SOEN6441 Project Document: Risk-like Strategy Game

---

1. Introduction

This document outlines the design and implementation of a Java-based strategy game inspired by "Risk." The project consists of two main modules:

- **Map System Module**: Manages continents, countries, and their adjacency, handles map creation, editing, and file I/O.
- **Game Play Module**: Manages the game logic, including the game start-up phase and the main game loop phase.

The application follows a Model-View-Controller (MVC) architecture to separate concerns and ensure modularity, scalability and maintainability.

- **Model**: Represents the data and business logic.
- **View**: Handles the user interface (console-based)
- **Controller**: Manages user input and updates the model and view accordingly.

---

2. Project Structure

```
src/
├──mapeditor/
│    ├── controller/
│        ├── MapEditorController.java
│    ├── model/
│        ├── Map.java
│        ├── Continent.java
│        ├── Country.java
│    ├── view/
│        ├── MapEditor.java
├──gameplay/
│    ├── engine/
│        ├── GameEngine.java
│    ├── phase/
│        ├── StartUp.java
│        ├── OrderCreation.java
│        ├── OrderExecution.java
│        ├── TournamentPlay.java
│    ├── model/
│        ├── Order.java
│        ├── Player.java
│        ├── DeployOrder.java
│        ├── AdvanceOrder.java
```

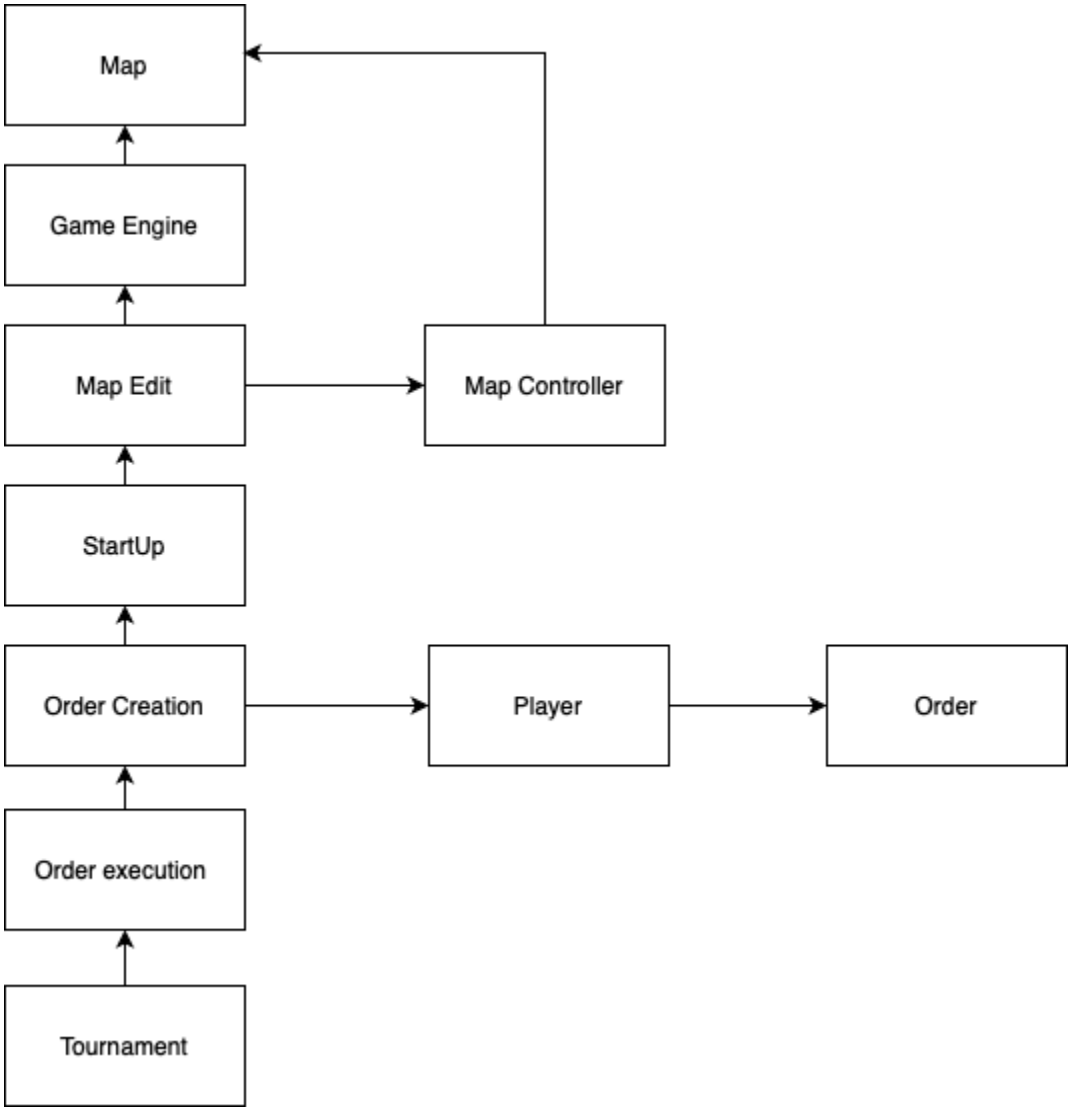
```
│       ├── Card.java
│       ├── Play.java
│       ├── MainPlay.java
│       ├── cards
│           ├── AirliftOrder.java
│           ├── BombOrder.java
│           ├── DiplomacyOrder.java
│           ├── BlockadeOrder.java
│   ├── strategy/
│       ├── PlayerBehavior.java
│       ├── PlayerBehaviorBase.java
│       ├── AggressiveBehavior.java
│       ├── BenevolentBehavior.java
│       ├── CheaterBehavior.java
│       ├── HumanBehavior.java
│       ├── RandomBehavior.java
└── utils/
    ├── MapFileHandler.java
    ├── Helper.java
    ├── InvaildMapException.java
    ├── FileManager.java
    ├── ConquestMapFileAdapter.java
    ├── ConquestMapFileManager.java
    ├── DominationMapFileAdapter.java
    ├── DominationMapFileManager.java
test/
├──Testing files
docs/
├──javadoc/
├──Risk project document - TYRCL.pdf
map/
├──map1.txt
├──canada.map
```

---

3. Architecture Diagram

The project is designed as a modular game system, where different components interact seamlessly to create a gaming experience. The game has been divided into distinct modules, each responsible for specific functionalities, while all modules rely on a central game engine to handle the overall presentation and coordination during the game process.

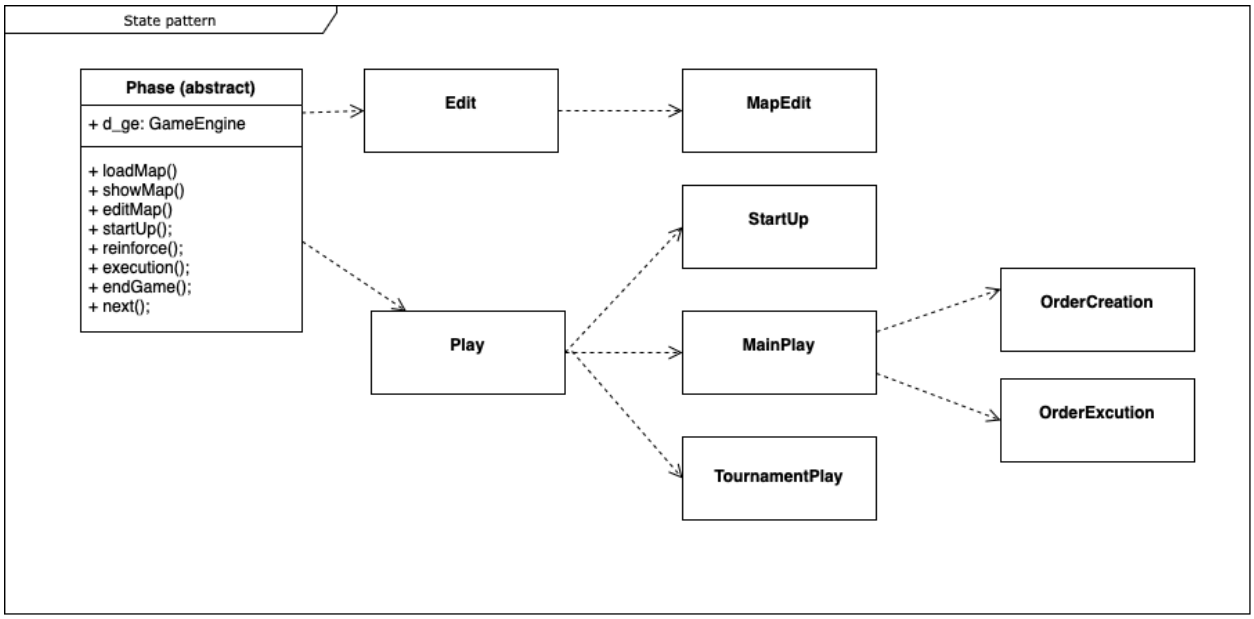
- The **Player Class** generates orders based on player input or decisions.
- These orders are sent to the **Order Creation Module**, which processes and validates them.
- The **Map Controller** updates the game map based on the executed orders, ensuring the game world reflects the changes.
- The **Game Engine** orchestrates the presentation, rendering the updated map and player actions to provide a cohesive gaming experience.
- The **Tournament Class** proceeds with options provided by the command and run the game without any user interaction and shows the results at the end.



4. Detailed Design

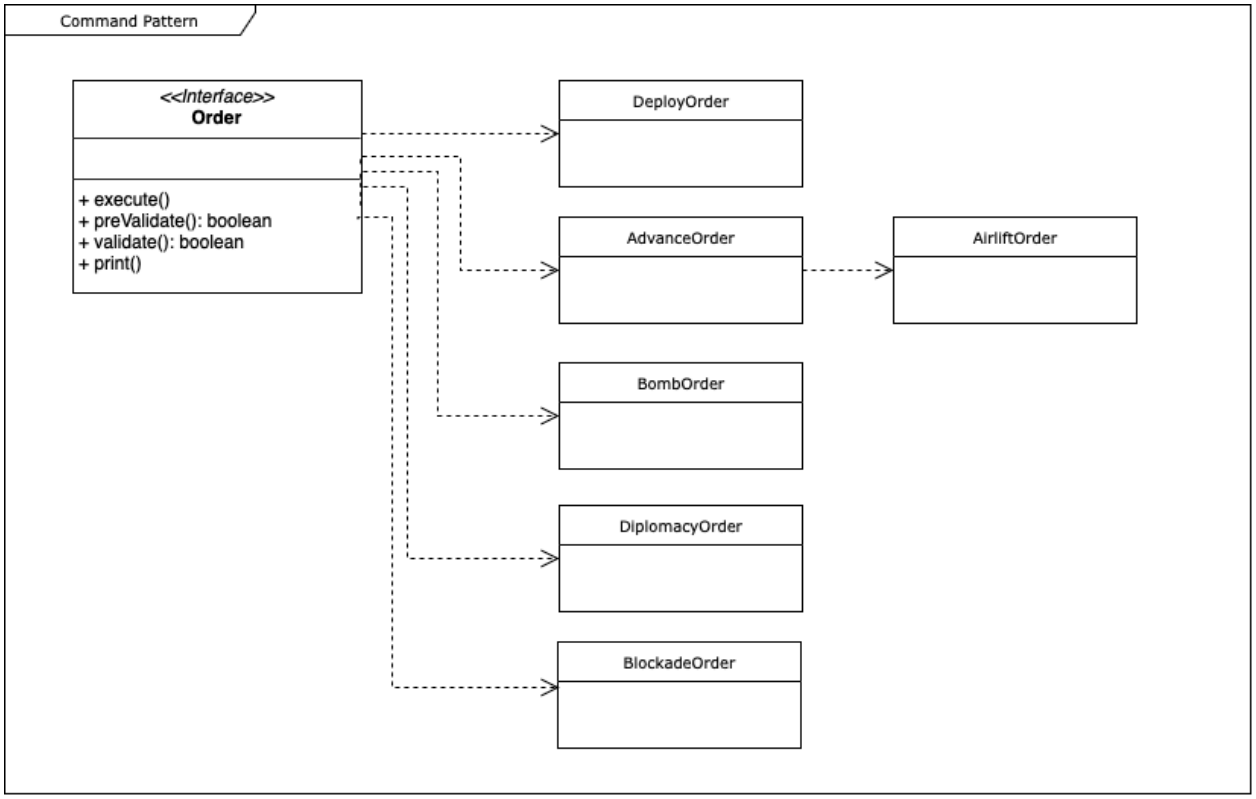
4.1 Game phases

The design leverages the State Pattern to manage the different phases of the application, which can be a game or a simulation. The State Pattern allows the application to change its behavior dynamically based on its current state (phase). Each phase is represented by a class, and the transitions between phases are managed by the context class.



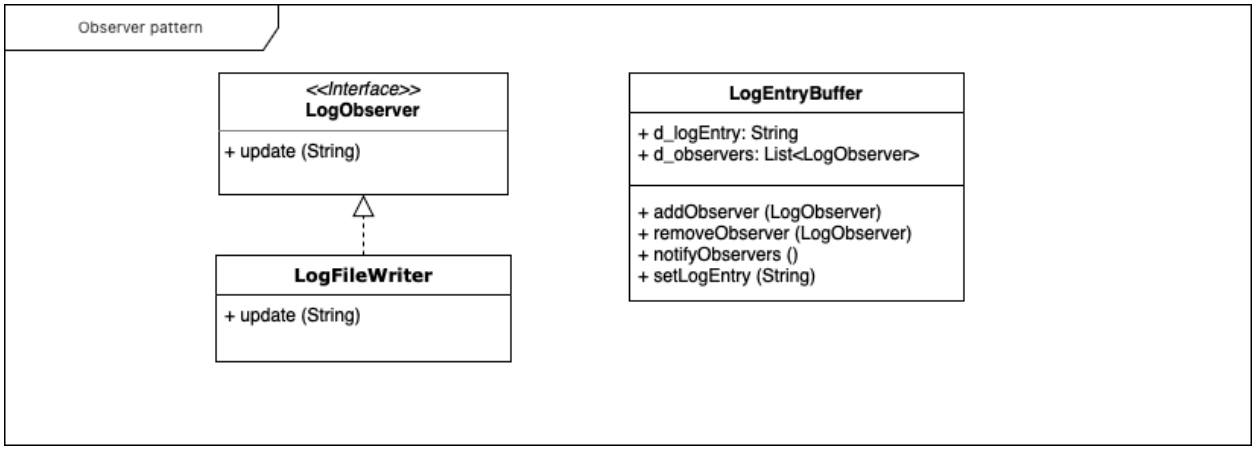
4.2 Orders

The design leverages the Command Pattern to encapsulate orders as objects, ensuring a clean separation between the request and its execution. At its core, an Order interface defines a common execute() method, which all concrete orders (e.g., AdvanceOrder, DeployOrder) implement to provide their specific behavior. For orders with similar functionality, such as AirliftOrder and AdvanceOrder, inheritance is used to reuse code and simplify implementation. This approach promotes modularity, allowing new orders to be added without modifying existing code, and ensures all orders can be executed uniformly through the same interface.



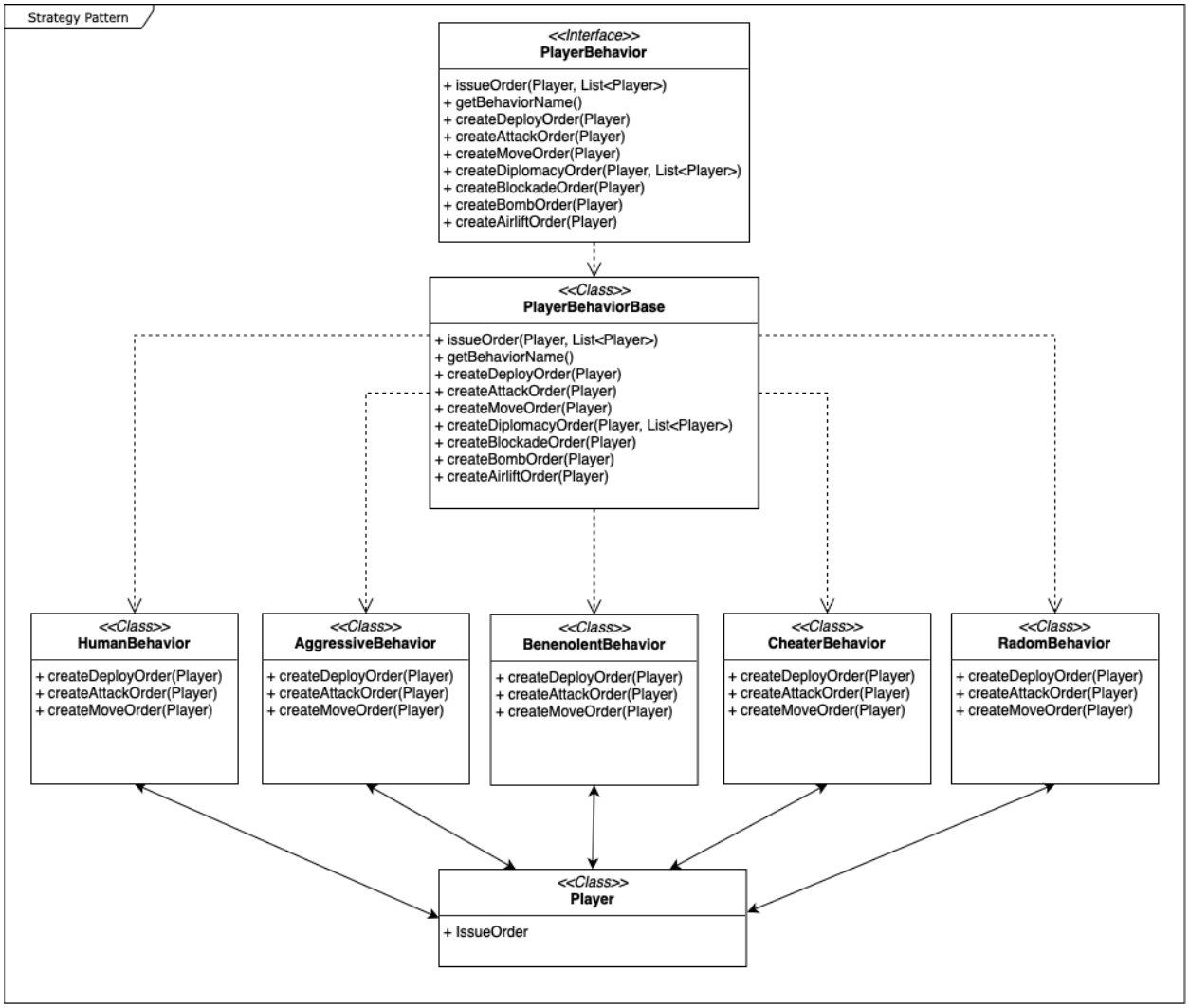
4.3 Game Log

A LogWriter that utilizes the observer pattern can be designed to allow multiple log observers (listeners) to be notified whenever a log entry is written. The LogWriter acts as the subject, maintaining a list of registered observers, such as file loggers, console loggers, or network loggers. When the LogWriter writes a log message, it iterates through its list of observers and invokes their update method, passing the log message as an argument. This decouples the logging logic from the observers, enabling flexible and extensible logging systems where new observers can be added or removed dynamically without modifying the core LogWriter implementation.



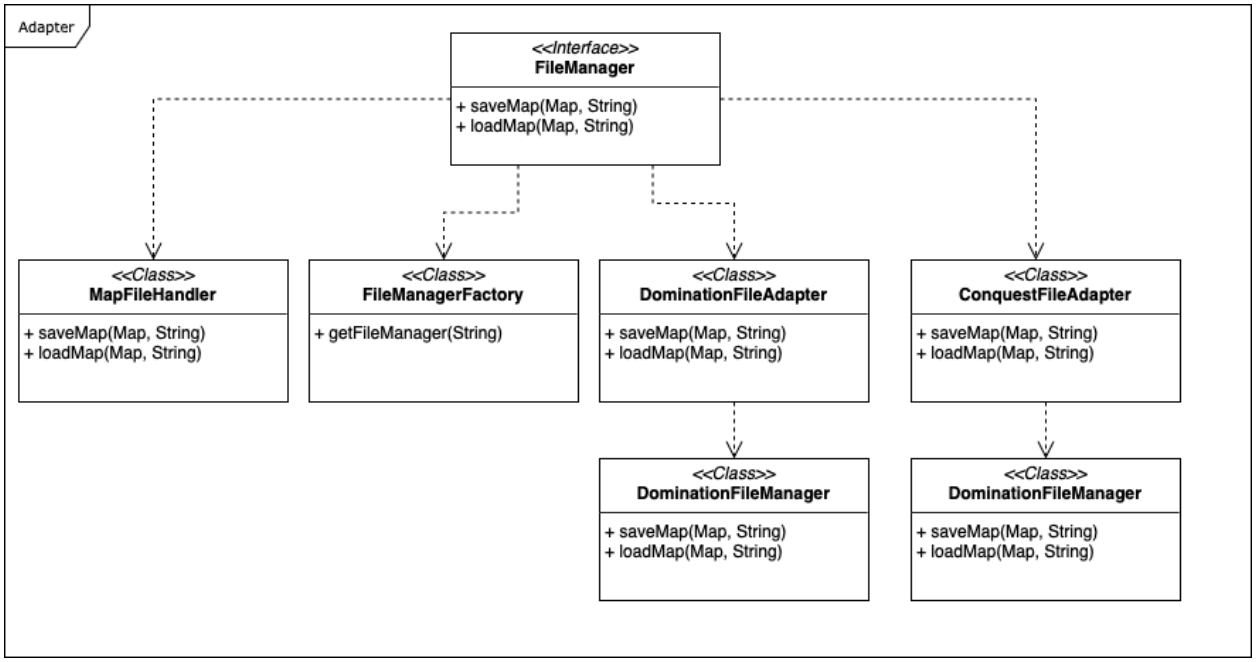
4.4 Player Behaviors

This design uses the Strategy Pattern to encapsulate different order-issuing behaviors (human, aggressive, benevolent, random, Cheater) in separate strategy classes. The Player class delegates order creation to its current strategy, allowing flexible behavior changes at runtime while keeping the core player logic clean and extensible. Each strategy implements deploy, attack, move and card orders differently based on its tactical approach.



4.5 File Manager

This design uses the Adapter Pattern to create a unified FileReader interface that can read different map file formats (domination, conquest) by wrapping their specific implementations in adapters. A factory selects the appropriate adapter based on the file extension, allowing the client to read files without knowing the underlying implementation details. This makes the system extensible (new formats can be added easily) and decouples the client code from concrete file readers.



---

5. Flow of Control with GameEngine

5.1 Initialization

The GameEngine initializes the map, players and phases.

5.2 Game Start-Up Phase:

The GameEngine calls the StartUp Phase to:

- Load the map
- Add players.
- Assign countries to players.

5.3 Main Game Order Creation Phase.

The GameEngine enters a loop and calls the MainLoopPhase to:

- Assign reinforcements.
- Issue orders.
- Execute orders.

5.4 Main Game Order Execution Phase.

The GameEngine enters a loop and calls the MainLoopPhase to:

- Assign reinforcements.
- Issue orders.
- Execute orders.

---

6. File Format Specification

6.1 Domination File Format

Example `map.txt`:

```
[continents]
1 Europe 5      # ID 1
2 Asia 7        # ID 2

[countries]
1 France 1    # Belongs to continent 1 (Europe)
2 China 2     # Belongs to continent 2 (Asia)

[borders]
1 2           # Country 1 borders country 2
2 1           # Country 2 borders country 1
```

6.2 Conquest File Format

Example `map.map`:

```
[Continents]
1 Europe=3      # ID 1
2 Asia=2        # ID 2

[Territories]
1,France,Europe,2    # Belongs to (Europe), borders territory 2
2,China,Asia,1     # Belongs to (Asia) , borders territory 1
```

---

7. Key Features

- **Map Validation**: Ensures playability via connectivity checks.
- **Round-Robin Order Creation**: Each player creates orders one by one in a turn. Players can create multiple orders in a turn but can create only one order at a loop.
- **Round-Robin Order Execution**: First come first serve order processing.
- **Reinforcement Calculation**.
- **Attack/occupy Mechanism.**
- **AutoPlay.**
- **Tournament Mode.**

---

8. Testing Plan

1. **Unit Tests**
    - Map validation.
    - File I/O for saving/loading maps.
    - Game engine.
    - Different orders.
    - Different phases.
    - Different players with different strategies.
    - Tournament.
2. **Integration Tests**
    - Full game loop with reinforcement assignment, order creation and order execution.