

Mesos — Final Project Report

Software Engineering — A.Y. 2025/2026

Giuseppe Pio Granatiero, Andrea Migliavacca, Simone Leonetti, Luca Ialuna

1 Implemented features

The project implements the board game **Mesos** as a distributed client–server application following the MVC pattern:

Complete Rules + TUI + GUI + RMI + Socket + 2 Advanced Features (FA)

- **Complete rules:** full game logic (no expert variant), supporting matches with the full set of character cards and any allowed number of players.
- **Two interfaces:** a textual interface (TUI) and a graphical interface (GUI, JavaFX), selectable at client startup.
- **Dual network technology:** both Socket (TCP) and RMI, selectable at startup; the server supports matches mixing clients that use different technologies.
- **FA1 — Game leaderboard on DB:** the server persists the history of completed matches on an external database and reports each player’s ranking among matches with the same player count.
- **FA2 — Multiple matches:** the server manages several concurrent matches through a lobby layer that lets players create or join open matches.

2 Work division

The group worked on feature-based branches (not personal ones). Before each merge, a member who did *not* implement the feature was assigned to write its tests, ensuring shared ownership of the codebase.

- **Giuseppe Pio Granatiero** — Model: `core`, `parser`, `cards` (`buildings.effects`, `events`); lobby manager handling multiple matches (FA2). View: interface design, GUI (setup, utilities, main menu).
- **Andrea Migliavacca** — Controller; network; View: GUI (lobby and in-game actions); Model: `player`.
- **Simone Leonetti** — Model: `board`; database (FA1); View: TUI.
- **Luca Ialuna** — Model: `player`, `resources` (JSON and pictures); View: design wireframes and UI.

3 Design choices and patterns

- **Pure polymorphism** for game components (cards), avoiding type checks on concrete card classes.
- **State pattern** to manage the finite-state machine of a match (game phases and turn flow).
- **Factory pattern** for the parser that builds game objects from JSON resources.
- **Command pattern** for the network layer, encapsulating client/server interactions as messages.
- **Strategy pattern** for the special effects of the cards.

- **Facade pattern** for **LobbyManager**, exposing a single entry point to the model.
- **Mediator pattern** for **GuiManager**, coordinating GUI components.

External libraries:

- **Gson** — in the parser, to deserialize the JSON asset files into model objects.
- **PostgreSQL** — relational database backing the game leaderboard (FA1).
- **Mockito** — mocking framework for unit tests.
- **H2** — in-memory database used for integration tests on the persistence layer.

4 Communication protocol

We implemented both supported network technologies: **Socket (TCP/IP)** and **RMI**. Both are exposed behind a common interface, so the server handles every client uniformly and can host matches in which players connect through different technologies.

5 Test coverage

Overall test coverage of the project, measured with IntelliJ IDEA:

	Class	Method	Branch	Line
Coverage	65%	52%	34%	39%
	(98/149)	(454/864)	(543/1575)	(1425/3630)

Testing effort concentrated on the model, where most of the game logic resides. Each test name and comment specifies the tested functionality and the components involved.