Exploding-Kittens Final Report

Final Report for the Distributed Systems Course 2024/2025

alessandra.versari2@studio.unibo.it

margherita.balzoni@studio.unibo.it

May 18, 2026

The project is a distributed base implementation of the popular card game Exploding Kittens. Exploding Kittens is a card game for 2 to 4 players. On each turn, a player either plays action cards from their hand or draws cards from the deck. Draw an Exploding Kitten and you are out, unless you have a Defuse card to save yourself. The last player standing wins.

AI Disclaimer

During the preparation of this work, the authors used Copilot and Claude to help finding bugs, to assist with code completion and for document generation. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the final report/artifact.

1 CONCEPT

Exploding Kittens is a simplified digital version of the well-known card game **Exploding Kittens**. The desktop application features three main screens:

- **Welcome screen**, where the user can enter the game and select the number of players.
- **Game screen**, where the actual gameplay takes place. Here, the player can view all actions performed by other players in the log, the list of active players, and, of course, their own hand of cards.
- **Result screen**, which informs the player whether they have won or lost the game.

The application allows players to connect and join a match, up to the maximum number of players configured for the game. Players interact frequently with each other, as this is a card game in which the main objective is to disrupt opponents' strategies while avoiding drawing an Exploding Kitten card, in order to remain the last player alive.

No data is stored persistently; however, a significant amount of information is exchanged in real time during gameplay.

1.1 Game Rules and Cards

The objective of the game is to avoid drawing an *Exploding Kitten* card. A player who draws an Exploding Kitten and does not possess a suitable countermeasure is immediately eliminated from the game. The last remaining player wins the match.

The base version of the game includes the following card types:

- **Exploding Kitten**: A critical card that causes the immediate elimination of the player who draws it, unless the player is able to counter its effect using a Defuse card.
- **Defuse**: Allows a player to neutralize an Exploding Kitten. After playing a Defuse card, the player may place the Exploding Kitten back into the deck at a position of their choice.
- **Attack**: Ends the current player's turn and forces the next player to take two consecutive turns.
- **Skip**: Allows a player to end their turn without drawing a card from the deck.
- **Shuffle**: Randomly shuffles the remaining cards in the deck, making the future order of draws unpredictable.
- **See the Future**: Allows a player to secretly view the next cards that will be drawn from the deck.

- **Cat Cards**: Allows a player to stole a card from a chosen target. The stolen card is randomly chosen from the target hand. To play this card, the player need to have two cat cards in his hand.

2 REQUIREMENTS

This section describes the system requirements, divided into functional and non-functional requirements.

2.1 Functional Requirements

1. When a user starts the application, a welcome screen must be displayed. This screen allows the user to choose a nickname, select the number of players for the match (if it's the first player to join the match), and start the game.
2. Once the game has started, and until the required number of players has joined, the user is shown the game screen with a log message indicating: *"Waiting for other players..."*.
3. If a player disconnects for any reason, the remaining players must be able to continue the game without interruption. If only one player remains active, that player is automatically declared the winner. When a player disconnects, they are considered to have lost the game, and the remaining players are notified accordingly.
4. Each player must be able to play cards from their hand during their turn. Additionally, tooltips must be displayed on cards to improve the usability of the game interface.
5. The game ends when only one player remains alive.
6. A player's turn ends only when they use one of the following actions: Skip, Attack, or by drawing a card from the deck (see section "Game Rules and Cards").
7. The system must display the result of each action (card played, effect, etc.)
8. A player cannot play cards that are not in their hand.
9. A player cannot perform actions outside their turn.
10. Nickname must be unique within a match. Empty or invalid nicknames must be rejected.

2.2 Non-Functional Requirements

1. The game must be compatible with major operating systems, including Unix-based systems, Windows, and macOS.
2. The codebase must be maintainable and extensible in order to allow future improvements and feature additions.
3. The system must ensure low latency to provide a smooth user experience, while maintaining strong consistency between all connected players during gameplay.

2.3 Implementation

The system has been implemented in Java, which was selected due to its strong portability and suitability for building distributed applications.

The graphical user interface has been developed using Java Swing. Swing was chosen for its simplicity, stability, and full integration within the Java ecosystem.

The multi-agent aspects of the system have been implemented using JADE (Java Agent DEvelopment Framework). JADE enables the development of agent-based systems following the FIPA specifications, facilitating communication between autonomous agents and supporting a distributed approach to game logic management.

The chosen technologies were primarily driven by design requirements, such as modularity, distributed interaction, and maintainability, rather than by external constraints.

2.4 Relevant Distributed System Features

This section analyzes the relevance of distributed system characteristics with respect to the developed application, which is based on a multi-agent architecture implemented using JADE.

- **Transparency** - Transparency is partially relevant in the system. From the user perspective, distribution details such as agent communication and message passing are hidden, ensuring a simple and intuitive gameplay experience. However, from a development perspective, the distributed nature of the system is explicit and directly managed.
- **Fault tolerance and dependability** - Fault tolerance is relevant. If a player agent disconnects or fails, the system must allow the game to continue for the remaining players. In addition, a backup game server is used to ensure continuity of the game state in case the primary game server fails. This mechanism improves the overall reliability of the system by reducing the risk of losing the current match state and allowing recovery of the ongoing game session.
- **Scalability** - The system is designed to support a variable number of players up to a predefined limit. However, it is not intended to scale to large numbers of concurrent users or distributed servers. Future extensions could improve scalability, but it is not a primary design constraint.
- **Security and trust** - The application does not handle sensitive or personal data, and no authentication or authorization mechanisms are required. All users are assumed to be trusted participants within the game session.
- **Resource sharing** - Resource sharing is a core aspect of the system. All players interact with shared game resources such as the deck, cards, and game state. Proper synchronization is required to ensure consistency and correct turn-based behavior among agents.

- **Openness, interoperability, and heterogeneity** - The system is self-contained and does not interact with external services or heterogeneous components. All modules are developed using Java and JADE.
- **Evolvability and maintainability** - The system is designed following a modular agent-based architecture, allowing future extensions such as new card types, rules, or game modes with minimal changes to the existing codebase.
- **Performance and concurrency** - Performance and concurrency are important due to the real-time nature of the game. Multiple agents operate concurrently, and communication must be efficient to ensure a smooth user experience. Low latency and timely message delivery are required to maintain consistent game state across all players.

3 DESIGN

This chapter explains the strategies used to meet the requirements identified in the analysis.

3.1 Architecture

The project adopts a combination of architectural styles, separating concerns between the local application structure and the distributed system design.

Local architecture: MVC pattern

- The internal structure of the application follows the **Model-View-Controller (MVC)** pattern.
- The *Model* is implemented in the `game.model` package and contains the core game logic (e.g., cards, deck, players).
- The *View* is implemented in the `game.view` package and is responsible for the graphical user interface.
- The *Controller* role is implicitly handled by the agents (in the `remote` package), which process user actions and coordinate the game flow.
- This separation improves modularity, maintainability, and clarity of responsibilities.

Distributed architecture: Client-Server

- For the distributed part, the system follows a **client-server architecture**.
- A central *GameMaster agent* acts as the server, managing the game state, validating actions, and coordinating all players.
- Each player runs a *PlayerAgent*, which acts as a client interacting with the GameMaster.
- A peer-to-peer architecture was not adopted because the game requires a consistent and authoritative game state. The client-server model simplifies coordination by delegating all decisions to a single authoritative entity (the GameMaster).

The distributed system is implemented using **JADE**. JADE provides built-in support for agent lifecycle management, allowing agents to be dynamically created, started, suspended, and terminated without manually handling low-level concurrency. It offers a **high-level communication model** based on asynchronous message passing (ACL messages), which simplifies interaction between distributed components. The framework naturally supports a **behavior-based programming model**, allowing complex agent logic to be decomposed into modular and reusable behaviours (e.g., listening, handling turns, managing sub-agents). The use of JADE will be further explored in the following sections.

3.2 Infrastructure

The system is composed of a set of distributed components interacting through the JADE platform.

- **Infrastructural components**
  - Each player corresponds to a **client**, which runs a `PlayerAgent`.
  - The **server** is represented by the `GameMasterAgent`, which is responsible for managing the game state, coordinating players, and enforcing game rules.
  - Each client internally spawns additional **sub-agents** (e.g., HandManager and KittenDefense) to modularize responsibilities.
  - A **backup server agent** is also introduced to improve fault tolerance, allowing the game to continue in case the main server fails.
  - No **persistent storage** (e.g., databases) is used, as persistence was not required by the project specifications.
  - No dedicated components such as load balancers, caches, or message brokers are used, since JADE natively handles communication between agents.
- **Distribution of components**
  - For simplicity, both clients and server agents are executed on the **same machine** during development and testing.
  - However, the system is designed to be fully **distributed**, and components can be deployed on different machines across a network without changes to the application logic.
  - Each client runs in its own JADE container, while the server typically runs in the main container.
- **Component discovery and communication**
  - Components identify each other through JADE's **Agent Management System (AMS)** and **Directory Facilitator (DF)**.
  - The DF acts as a **service registry**, allowing agents (e.g., players) to dynamically discover the GameMaster agent.
  - Agents are uniquely identified by their **AID (Agent Identifier)**, which ensures unambiguous communication.
  - Communication between components is performed via **asynchronous ACL messages**, without requiring direct knowledge of network addresses.

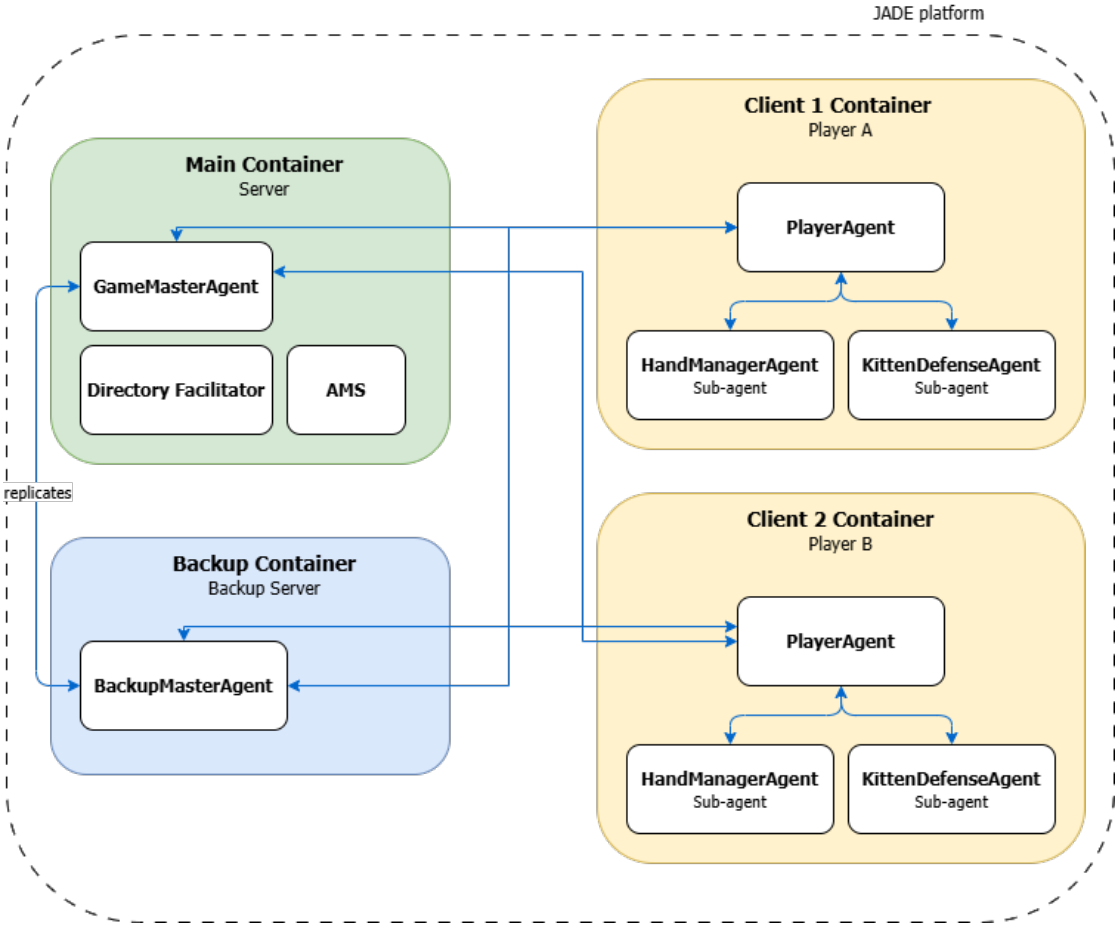

Figure 1: Infrastructure diagram

3.3 Modelling

The main domain entities of the system are:

- **Card**: represents a single game card with its type and effect.
- **Deck**: represents the collection of cards used during the game, including draw and discard operations.
- **Player**: represents a participant in the game, identified by a unique nickname and associated with an agent identifier.
- **GameState**: represents the global state of the match, including the list of active players, turn information, and game progression data.
- **Hand**: represents the set of cards held by a player at a given time.

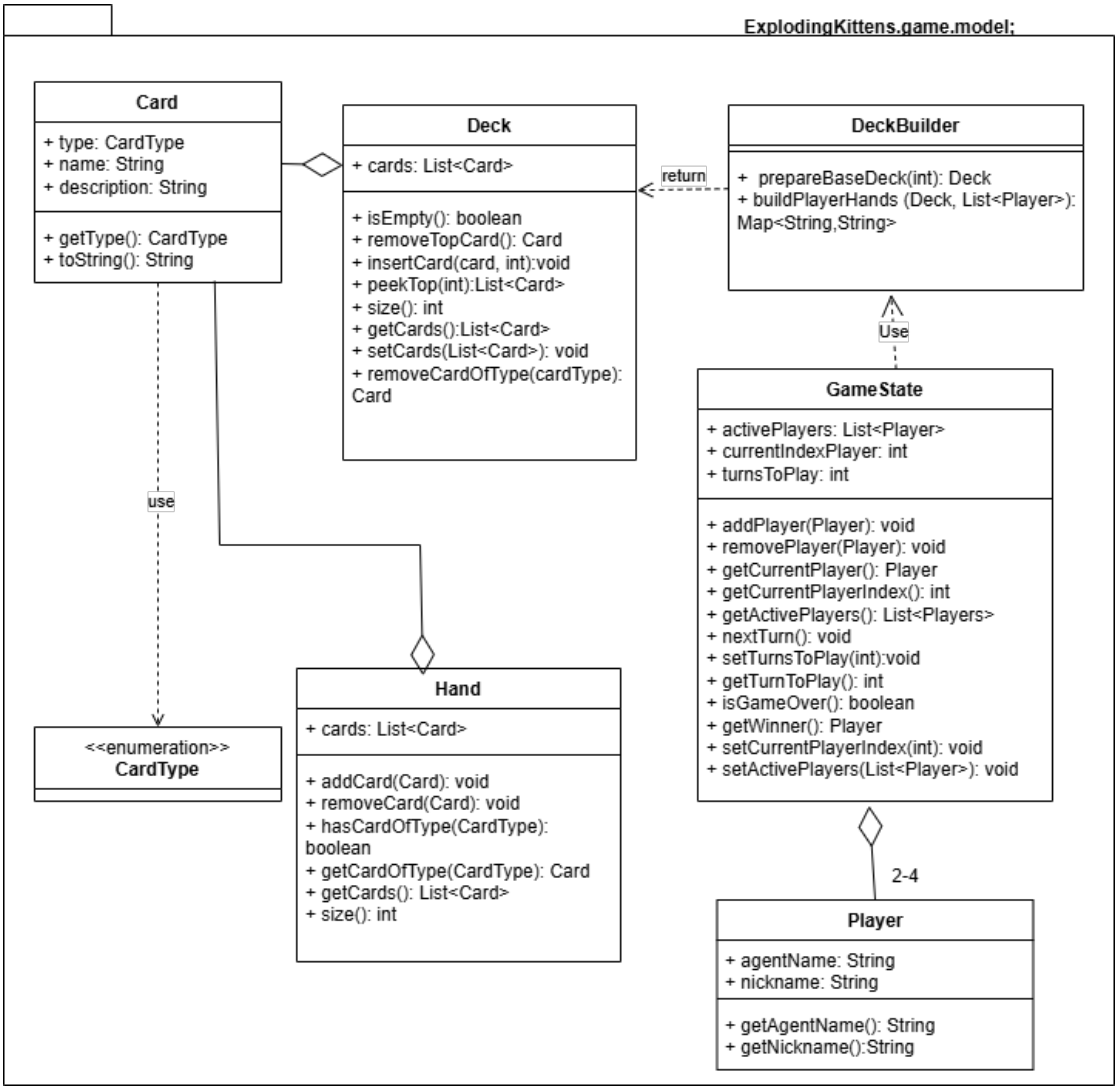

Figure 2: Class diagram about model package

The domain model is mapped onto a multi-agent infrastructure implemented using JADE, following a clear separation between global state management and local player state.

GameMasterAgent and GameState The **GameState** entity is managed by a central **GameMasterAgent**, which is responsible for maintaining the authoritative state of the game. This includes:

- the list of active **Player** objects
- the current **Deck**

- turn management and game progression logic.

Each **Player** entity in the GameState stores a reference to the corresponding JADE agent identifier (AID), allowing communication between the GameMaster and the player agents. To improve fault tolerance, a `BackupMasterAgent` is also deployed alongside the primary server. Both agents extend a shared abstract class, `AbstractMasterAgent`, which encapsulates all core game logic (lobby management, turn handling, card effects, state serialization).

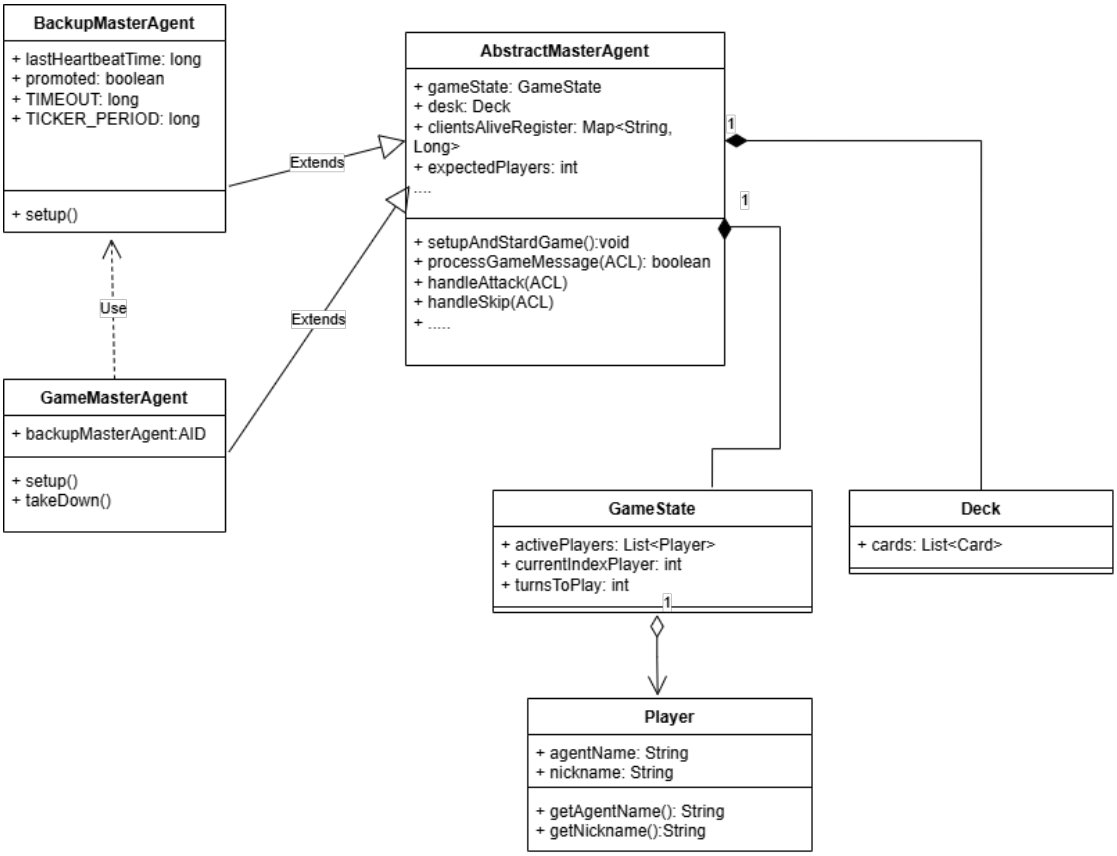

Figure 3: Class diagram about GameState and GameMasterAgent

PlayerAgent and local state Each player is represented by a dedicated **PlayerAgent**, which encapsulates the local state of the player. Upon creation and successful nickname validation, the PlayerAgent is registered in the GameState.

The PlayerAgent internally delegates responsibilities to two sub-agents:

- **HandManagerAgent**: responsible for managing the player's hand of cards, represented by the **Hand** entity. It handles drawing, playing, and updating cards in response to game events.

- **KittenDefenseAgent**: responsible for evaluating survival conditions when a danger event occurs (e.g., drawing a losing card). It coordinates with the PlayerAgent and HandManagerAgent to determine whether the player survives or is eliminated.

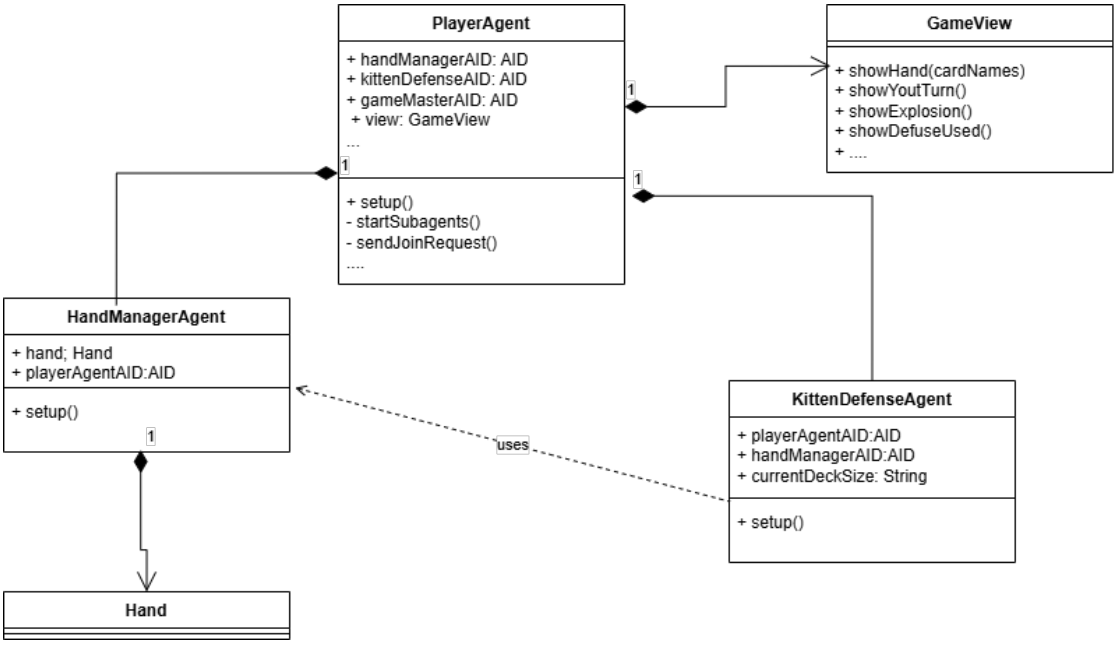

Figure 4: Class diagram about PlayerAgent and local state

Card and Deck management The **Deck** and **Card** entities are managed exclusively by the GameMasterAgent. The deck is modified only by centralized operations such as drawing or reshuffling, ensuring consistency across distributed clients.

3.3.1 System state overview

At runtime, the global system state consists of:

- the authoritative **GameState** maintained by the GameMasterAgent,
- the **Deck** and discard pile,
- the list of active **Player** entities,
- distributed local states, like hand and gameplay logic, inside each PlayerAgent and sub-agents (HandManagerAgent, KittenDefenseAgent).

This architecture ensures a clear separation between centralized game logic (GameMasterAgent) and distributed player-side logic, while maintaining consistency through message-based coordination.

3.3.2 Domain Events

Although the system does not implement a formal event-driven architecture, domain events are implicitly represented as state transitions triggered by agent message exchanges. The main domain events handled by the system are:

- **PlayerJoined**: a player successfully joins the game after nickname validation and is added to the GameState.
- **PlayerValidationCompleted**: the system validates whether a nickname is unique within the current match and if the number of players has not reached the maximum set.
- **GameStarted**: triggered when the number of registered players reaches the expected threshold and the GameMasterAgent initializes the game state.
- **CardDrawn**: a player draws a card from the deck.
- **PlayerEliminated**: a player is removed from the game after failing a survival condition.
- **TurnChanged**: the turn passes from one player to the next. The GameMasterAgent updates the active player and propagates the new turn to all PlayerAgents.
- **CardPlayed**: a player plays a card from their hand, triggering its effect and updating both local (Hand) and global (GameState) state.
- **GameEnded**: triggered when a single player remains and is declared winner.

3.3.3 Messages Exchanged

The multi-agent system relies on a set of message constants defined in `Messages.java` to coordinate communication among agents. Below, all messages are reported grouped by functionality.

- **Setup and Registration**

  `JOIN` Request sent by a `PlayerAgent` to join the lobby

  `JOINED` Confirmation that the player successfully joined the lobby

  `LOBBY_FULL` Error response indicating the lobby has reached its player limit

  `CHECK_NICK_AND_LOBBY:` Pre-join check sent by `NicknameCheckerAgent` to verify nickname availability and lobby status

  `VALID_HOST / VALID_GUEST` Confirmation of the player's role: host (first player, sets the number of players) or guest

  `INVALID` Error response indicating the chosen nickname is already in use

- **Turn Management**

`YOUR_TURN` Notifies the current player that it is their turn to act

`TURN:` Notifies all other players of whose turn it currently is

`NOT_YOUR_TURN` Error response sent when a player attempts to act outside their turn

- **State Synchronization**

  `PLAYER_LIST:` Broadcast containing the updated list of active players, sent after each join or elimination

- **Game Logic**

  `HAND:` Initial hand distribution sent by the `GameMasterAgent` to each player at game start

  `DRAW` Request sent by the active player to draw a card from the deck

  `DREW:` Confirmation sent by the `GameMasterAgent` specifying which card was drawn

  `DREW:EXPLODING_KITTEN` Specialisation of `DREW:` indicating the drawn card was an Exploding Kitten; also carries the current deck size

- **Player Actions**

  `PLAY:` Request to play a card from hand, followed by the card type (e.g. `PLAY:SKIP`)

  `CAT_CARD:` Request to play a Cat Card pair, followed by the target player's nickname

  `DEFUSE:` Request to use a Defuse card, followed by the position in the deck where the Exploding Kitten should be reinserted

- **Card Effects**

  `SKIP_OK` Confirmation that the Skip card was played and the current turn ended without drawing

  `ATTACK_OK` Confirmation that the Attack card was played; the next player must take two consecutive turns

  `SHUFFLE_OK` Confirmation that the Shuffle card was played and the deck reshuffled

  `SEE_THE_FUTURE:` Response containing the types of the top three cards in the deck, visible only to the acting player

  `YOU_STOLE:` Notifies the acting player of the card they stole from the target

  `STOLEN_FROM_YOU:` Notifies the target player that one of their cards was stolen, specifying which one

- **Hand Management**

  `ADD_CARD:` Instructs the `HandManagerAgent` to add a specific card to the player's hand

`REMOVE_CARD:` Instructs the `HandManagerAgent` to remove a specific card from the player's hand

`REFRESH_HAND` Instructs the `PlayerAgent` to request and re-render the current hand from `HandManagerAgent`

`REFRESH_HAND_AFTER_DEFUSE` Variant of `REFRESH_HAND` sent by `KittenDefenseAgent` after a Defuse card is consumed

`REQUEST_HAND` Request sent by the `GameMasterAgent` to a `PlayerAgent` to obtain the current serialized hand (e.g. to validate a card play)

`HAND_RESPONSE:` Response from `PlayerAgent` to `GameMasterAgent` containing the serialized hand

`GET_HAND` Request sent by `PlayerAgent` to `HandManagerAgent` to retrieve the current hand

`HAND_READY` Confirmation from `HandManagerAgent` that the initial hand has been loaded and is ready

`WINNER:` Broadcast sent when the game ends, containing the nickname of the winning player

`PLAYER_DISCONNECTED:` Broadcast notifying all players that a participant has disconnected and been treated as eliminated

`PLAYER_ELIMINATED_BROADCAST:` Broadcast notifying all players that a participant has been eliminated by drawing an Exploding Kitten without a Defuse

- **Exploding Kitten and Defuse Logic**

  `EXPLODING_KITTEN_DRAWN` Sent by `PlayerAgent` to `KittenDefenseAgent` to trigger the defuse evaluation procedure

  `PLAYER_ELIMINATED` Sent by `KittenDefenseAgent` to `PlayerAgent` to notify that the player has no Defuse and is eliminated; forwarded by `PlayerAgent` to `GameMasterAgent`

  `DEFUSED` Confirmation from `GameMasterAgent` that the Defuse was accepted and the Exploding Kitten reinserted in the deck

  `HAS_DEFUSE?` Query sent by `KittenDefenseAgent` to `HandManagerAgent` to check whether a Defuse card is available

  `HAS_DEFUSE:YES` Positive response from `HandManagerAgent`: the player holds at least one Defuse card

  `HAS_DEFUSE:NO` Negative response from `HandManagerAgent`: no Defuse card is available

  `USE_DEFUSE` Instruction from `KittenDefenseAgent` to `HandManagerAgent` to consume and remove the Defuse card from hand

- **UI Interaction Messages**

`SHOW_DEFUSE_USED:` Sent by `GameMasterAgent` to all players to display in the log that a specific player used a Defuse card

`SHOW_ELIMINATED` Sent by `KittenDefenseAgent` to `PlayerAgent` to trigger the elimination popup in the GUI

`ASK_DEFUSE_POSITION:` Sent by `KittenDefenseAgent` to `PlayerAgent` to prompt the user to choose the position in the deck where the Exploding Kitten should be reinserted; carries the current deck size

- **Failure Handling and Fault Tolerance**

`HEARTBEAT:` Sent periodically by `GameMasterAgent` to `BackupMasterAgent`, carrying the serialized game state; used to detect primary server failure

`HEARTBEAT_CLIENT` Sent periodically by each `PlayerAgent` to `GameMasterAgent` to signal that the client is still alive; used to detect player disconnections

`NEW_MASTER` Sent by `BackupMasterAgent` to all registered players upon promotion, instructing them to redirect future messages to the new master

3.4 Interaction

3.4.1 Interaction and Communication

All communication in the system is based on **asynchronous message passing** using JADE's ACL (Agent Communication Language) protocol. Each message carries a *performative* that characterizes the intent of the communication (e.g., `REQUEST`, `INFORM`, `CONFIRM`, `REFUSE`).

The main interaction adopted are:

- **Request-Response**: used when a component needs a reply from another (e.g., a player requests to join the lobby and waits for confirmation).
- **Publish-Notify**: used when the GameMaster broadcasts state updates to all players (turn changes, player list updates, game results).
- **Periodic Heartbeat**: used between the GameMaster and the BackupMaster to replicate state and detect failures, and between clients and the GameMaster to detect player disconnections.
- **Delegation**: used internally within a player's agent hierarchy, where the PlayerAgent delegates responsibilities to its sub-agents (HandManagerAgent and KittenDefenseAgent) via local message passing.

3.4.2 Player Registration Flow

The registration flow begins when a user launches the client application. Before joining the game, the client performs a preliminary **nickname and lobby validation** activating the NicknameCheckerAgent. This agent sends `NICKNAME_AND_LOBBY_CHECK` message to the GameMaster. The GameMaster replies with one of the following outcomes:

- `VALID_HOST`: the lobby does not yet exist and this player will create it, choosing the number of participants.
- `VALID_GUEST`: the lobby exists and there is still room for additional players.
- `LOBBY_FULL`: no more players can join.
- `INVALID_NICKNAME`: the chosen nickname is already in use.

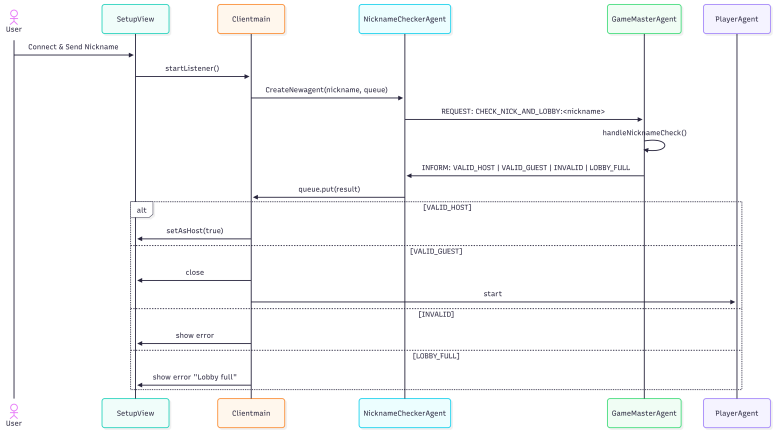

Figure 5: Sequence Diagram: Initial validation

Once validation is successful, the PlayerAgent sends a `JOIN` message to the GameMaster, including the desired number of players (set by the host). The GameMaster registers the player and replies with `JOINED`. When the expected number of players is reached, the game starts automatically.

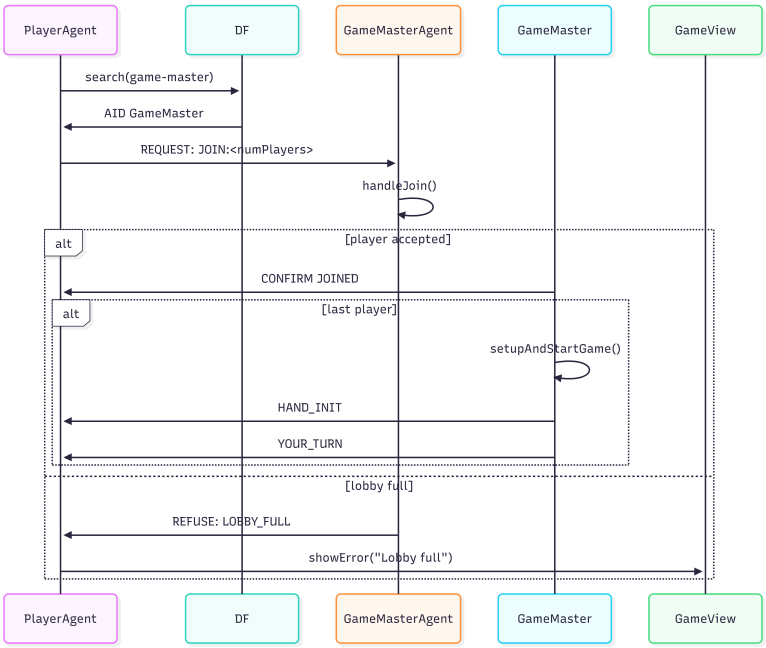

Figure 6: Sequence Diagram: Registration flow

3.4.3 Exploding Kitten and Defuse Flow

When a player draws an Exploding Kitten, a multi-step internal interaction is triggered within the player's agent hierarchy:

1. The PlayerAgent forwards `KITTEN_DRAWN` to the KittenDefenseAgent.
2. The KittenDefenseAgent queries the HandManagerAgent with `HAS_DEFUSE_ASK`.
3. If the HandManagerAgent replies `HAS_DEFUSE_YES`, the defuse procedure begins: the HandManagerAgent removes the Defuse card (`USE_DEFUSE`), and the player is asked to choose the position in which to reinsert the Exploding Kitten (`ASK_DEFUSE_POSITION`).
4. The chosen position is communicated to the GameMaster via `DEFUSE_PLAY:<position>`.
5. If the HandManagerAgent replies `HAS_DEFUSE_NO`, the player is eliminated: the KittenDefenseAgent sends `SHOW_ELIMINATED` to update the UI and `PLAYER_ELIMINATED` to notify the GameMaster, which removes the player and advances the turn.

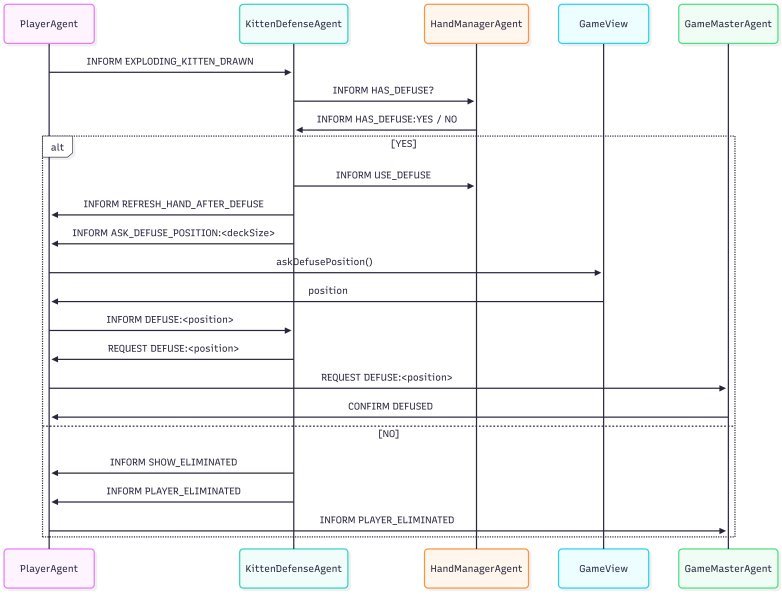

Figure 7: Sequence Diagram: Exploding kitten drawn

3.4.4 GameMaster Failure

The system implements a **heartbeat-based failover mechanism** to handle GameMaster failures.

During normal operation, the GameMaster periodically sends `HEARTBEAT:<serializedState>` messages to the BackupMaster. These messages carry a full serialization of the game state, allowing the BackupMaster to continuously reconstruct an up-to-date replica.

If the BackupMaster does not receive a heartbeat within the defined `TIMEOUT` threshold, it assumes the GameMaster has failed and initiates the **promotion procedure**:

1. The BackupMaster promotes itself and updates its registration in the Directory Facilitator from `backup-master` to `game-master`.
2. It broadcasts a `NEW_MASTER` message to all registered players (retrieved from the DF).
3. Each PlayerAgent updates its internal reference to the GameMaster and, if the game was already in progress, resumes the current turn; otherwise, it re-sends a `JOIN` request to the new master.
4. The BackupMaster resumes game management from the last replicated state.

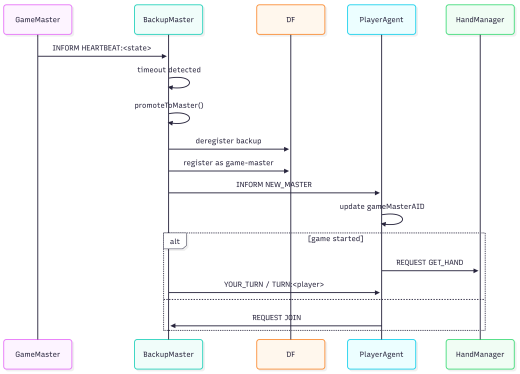

Figure 8: Sequence diagram of GameMaster failure

3.4.5 Client Disconnection Handling

Players periodically send `HEARTBEAT_CLIENT` messages to the GameMaster. The GameMaster stores the timestamp of the last received heartbeat for each player and periodically checks for timeouts via a `TickerBehaviour`.

If a player exceeds the `PLAYER_TIMEOUT` threshold without sending a heartbeat, the GameMaster:

1. Removes the player from the active player list.
2. Broadcasts `PLAYER_DISCONNECTED:<nickname>` to all remaining players.
3. If the disconnected player was the current one, advances the turn to the next player.
4. If only one player remains, announces the winner via `WINNER:<nickname>`.

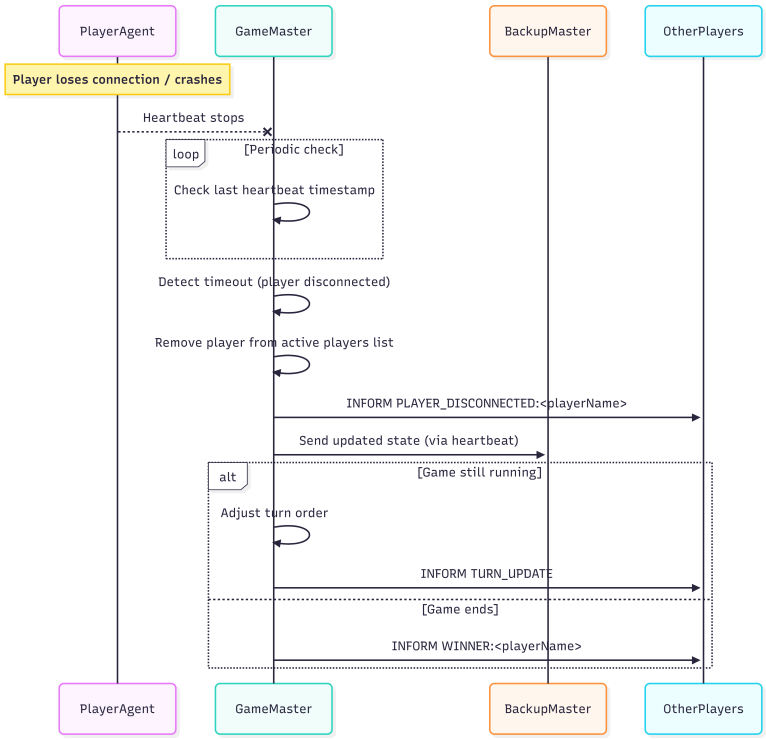

Figure 9: Sequence Diagram: Client Failover

3.5 Behaviour

3.5.1 JADE Behaviour Model

In JADE, agent logic is decomposed into **behaviours**, which are modular units of execution scheduled by the agent's internal thread. The system makes use of three main types of behaviours:

- **CyclicBehaviour**: executes repeatedly and indefinitely.
- **OneShotBehaviour**: executes exactly once and then terminates.
- **TickerBehaviour**: executes periodically at a fixed time interval.

3.5.2 GameMasterAgent

The GameMasterAgent is the most complex stateful component of the system. It transitions through the following operational phases:

1. **Lobby phase**: managed by a `LobbyBehaviour` (CyclicBehaviour). The agent waits for `JOIN` and `NICKNAME_AND_LOBBY_CHECK` requests. Each valid JOIN adds

a player to the GameState and triggers an immediate state synchronization with the BackupMaster. When the expected number of players is reached, the LobbyBehaviour is removed and the game starts.

2. **Game phase**: managed by a `HandleActionBehaviour` (CyclicBehaviour). The agent processes all incoming game messages (DRAW, PLAY, DEFUSE, HEARTBEAT_CLIENT, etc.) and updates the GameState accordingly.

3. **Heartbeat sending**: a `TickerBehaviour` running in parallel periodically sends the serialized game state to the BackupMaster.

4. **Disconnection checking**: after game start, a second `TickerBehaviour` periodically checks client heartbeat timestamps and removes inactive players.

3.5.3 BackupMasterAgent

The BackupMasterAgent operates in two distinct phases:

1. **Backup phase**: the agent passively listens for `HEARTBEAT` messages from the GameMaster via a `CyclicBehaviour`. Each received heartbeat updates the internal replica of the game state through `reconstructState()`, and resets the heartbeat timestamp. A parallel `TickerBehaviour` periodically checks whether the last heartbeat exceeded the `TIMEOUT` threshold (10 seconds).

2. **Promoted phase**: if the timeout is exceeded, the agent promotes itself to GameMaster. It updates its DF registration, broadcasts `NEW_MASTER` to all players, and begins processing game messages using the same logic inherited from `AbstractMasterAgent`. From this point, the agent behaves identically to a GameMasterAgent, managing lobby, game flow, and disconnection detection.

3.5.4 PlayerAgent

The PlayerAgent is an agent that transitions through the following phases:

1. **Registration phase**: a `OneShotBehaviour` searches the DF for the GameMaster and sends a `JOIN` request. A subsequent `CyclicBehaviour` waits for `CONFIRM` or `REFUSE`.

2. **Waiting phase**: after a successful JOIN, the agent waits for the game to start (i.e., for the `HAND_INIT` message). A parallel `TickerBehaviour` continuously sends `HEARTBEAT_CLIENT` messages to the GameMaster to signal liveness.

3. **Game phase**: managed by `MainListenerBehaviour` (CyclicBehaviour). The agent dispatches incoming messages to the appropriate handlers based on the sender:
   - Messages from the GameMaster are routed to `dispatchFromGameMaster()`, which updates the UI and coordinates with sub-agents.

- Messages from the HandManagerAgent are routed to `dispatchFromHandManagerAgent()`.
- Messages from the KittenDefenseAgent are routed to `dispatchFromKittenDefenseAgent()`.

4. **Failover handling**: if a `NEW_MASTER` message is received, the agent updates its GameMaster reference and either re-sends a JOIN (if the game had not started) or resumes the current turn.

3.5.5 HandManagerAgent

The HandManagerAgent is a **stateful** sub-agent responsible for maintaining the local hand of cards. It runs a single `CyclicBehaviour` that processes the following messages:

- `HAND_INIT`: initializes the hand with the distributed cards and notifies the PlayerAgent with `HAND_READY`.
- `ADD_CARD` / `REMOVE_CARD`: updates the hand by adding or removing a specific card.
- `GET_HAND`: serializes and returns the current hand to the PlayerAgent.
- `HAS_DEFUSE_ASK`: checks whether the hand contains a Defuse card and replies accordingly.
- `USE_DEFUSE`: removes the Defuse card from the hand.

The agent is entirely **reactive**: it only acts in response to messages and holds no autonomous logic.

3.5.6 KittenDefenseAgent

The KittenDefenseAgent manages the survival procedure when a player draws an Exploding Kitten. It transitions through the following behaviour sequence:

1. **ListenForKittenBehaviour** (CyclicBehaviour): waits for a `KITTEN_DRAWN` message from the PlayerAgent.
2. **WaitForDefuseCheckBehaviour** (CyclicBehaviour): after querying the HandManagerAgent, waits for `HAS_DEFUSE_YES` or `HAS_DEFUSE_NO`.
3. **UseDefuseBehaviour** (OneShotBehaviour): if a Defuse is available, removes it and requests the player to choose the reinsertion position.
4. **WaitForExplodingPositionBehaviour** (CyclicBehaviour): waits for the chosen position and forwards the `DEFUSE_PLAY` message to the PlayerAgent, which in turn sends it to the GameMaster.

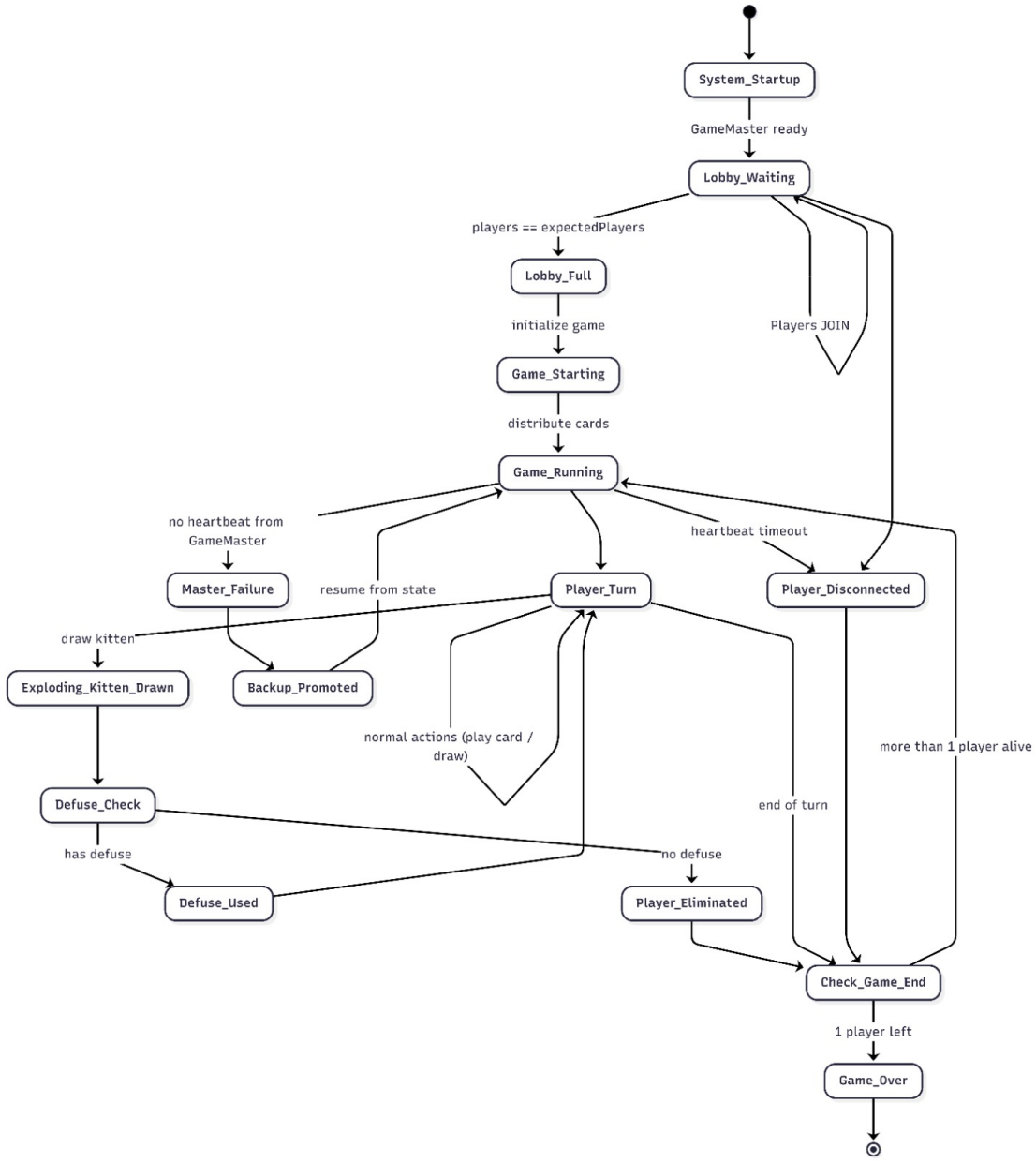

Figure 10: State Diagram

3.6 Data and Consistency Issues

No persistent storage is used in the system. The entire game state is maintained in memory by the GameMasterAgent, which is responsible for updating and managing the global state through interactions with the PlayerAgents.

Data sharing is necessary to ensure that all components (GameMaster, BackupMaster, and Player agents) have a consistent view of the game and can correctly coordinate their actions.

The shared data includes:

- The global game state (active players, turn order, number of remaining turns)
- The deck configuration (remaining cards and their order)
- Player-related information (nicknames, agent identifiers, connection status)
- Other gameplay information (pending actions, current turn, special card effects)

This information is exchanged through message passing:

- Between GameMaster and Player agents, to notify game events and enforce game rules
- Between GameMaster and BackupMaster, to replicate the full game state via heartbeat messages
- Between Player agent and his sub agents.

Consistency is ensured by adopting a centralized control model, where the GameMaster acts as the single source of truth. All state modifications are performed by the GameMaster and propagated to other components, avoiding concurrent conflicting updates.

3.7 Fault-Tolerance

The system implements a fault-tolerance strategy for both server-side and client-side failures, based on **heartbeat**, **timeout**, and **state replication** mechanisms.

- **Data replication**
  - *Why*: Replication is required to ensure game continuity in case of a failure of the primary server (*GameMaster*).
  - *How*: The GameMaster periodically sends heartbeat messages to the *BackupMaster*, including a full serialization of the game state (active players, deck, current turn, etc.).

    The BackupMaster continuously reconstructs the state using this information (`reconstructState`). In case of a primary failure, the backup is already synchronized and can seamlessly continue the game without loss of information.
- **Heart-beating and timeout**
  - *Why*: Heartbeat mechanisms allow the system to detect failures of both servers and clients, preventing the system from stalling.

- *Among which components*:
  - Between GameMaster and BackupMaster (server-side fault detection)
  - Between clients (players) and GameMaster (client-side fault detection)
- *How*:
  - The GameMaster periodically sends heartbeat messages to the BackupMaster containing the current game state.
  - The BackupMaster keeps track of the timestamp of the last received heartbeat; if it exceeds a predefined threshold (`TIMEOUT`), the BackupMaster automatically promotes itself to the new GameMaster.
  - Clients periodically send heartbeat messages to the server, which stores the last received timestamp for each player.
  - A periodic check identifies timeouts: if a client does not send a heartbeat within a given threshold (`PLAYER_TIMEOUT`), it is considered disconnected.

  No explicit retry mechanism is implemented, as the system is designed to react to failures by removing or replacing inactive components.

3.8 Availability

System availability is primarily ensured through the presence of a BackupMaster agent, which acts as a failover mechanism for the primary GameMaster.

The BackupMaster continuously receives heartbeat messages from the GameMaster containing a serialized version of the current game state. This allows it to maintain an up-to-date in-memory replica of the system state. In case of failure of the primary server, the BackupMaster promotes itself to GameMaster and continues the game without interruption. This mechanism significantly improves availability by avoiding a single point of failure.

3.9 Security

The system does not implement explicit security mechanisms such as authentication, authorization, or cryptographic protocols. This design choice is motivated by the nature of the application, which not handle sensitive data.

4 IMPLEMENTATION

4.1 Network Protocols

Since the system is built on top of the JADE, the underlying network communication is entirely managed by the platform itself.

Since all containers run on the same host during development, JADE transparently selects **Java RMI** as the inter-container transport protocol. Should the system be deployed across multiple machines, JADE would automatically switch to **IIOP** without any changes to the application code, as the transport selection is handled entirely by the platform.

This means that no explicit choice of application-level protocol was required: JADE abstracts away the transport layer, exposing only the FIPA-compliant ACL messaging model to the developer.

4.2 In-Transit Data Representation

All inter-agent communication is performed via **FIPA ACL messages**, whose content field is serialized as plain strings following a structured, human-readable format agreed upon at the application level.

No binary serialization format (e.g., Protocol Buffers) or structured markup language (e.g., JSON or XML) was adopted, since JADE's built-in content languages and the relatively simple nature of the exchanged data did not justify the additional complexity. Should the system be extended with richer payloads, a migration to JSON-encoded ACL content would be a natural and low-effort improvement.

4.3 Databases

No persistent storage layer was introduced in the system. The project specifications did not require persistence across sessions, and all game state is held in memory by the `GameMasterAgent` for the duration of a single game. As a consequence, no database technology (relational or otherwise) was adopted, and no query language (SQL or NoSQL) was needed.

4.4 Authentication

The system does not implement explicit authentication mechanisms. Agent identity is managed natively by JADE through the **Agent Management System (AMS)**, which assigns each agent a unique **AID** (Agent Identifier) upon registration. Since all agents are instantiated programmatically within a controlled JADE platform, the AMS guarantee of unique identity is considered sufficient for the scope of this project.

No external authentication protocols were adopted, as the system is not exposed to untrusted external clients and operates in a closed, developer-controlled environment.

5 VALIDATION

5.1 Automatic Testing

The system was validated through an extensive suite of automated tests implemented using **JUnit 5**. The testing strategy combines *unit tests*, *integration tests*, and *distributed tests* based on the JADE multi-agent framework.

- ***Individual components*** of the game logic were tested in isolation using standard JUnit unit tests. The main classes tested include `Deck`, `Hand`, `GameState`, and `DeckBuilder`.

  Each test verifies the correctness of state manipulation operations (e.g., adding/removing cards, turn rotation, deck updates) and ensures consistency of internal data structures.

  Corner cases were explicitly considered, such as drawing from an empty deck, limited "See the Future" visibility, etc.

  These tests ensure correctness of the core game rules in complete isolation from the distributed layer (Requirements covered: game rules consistency, deck correctness, hand management, win condition validation). They are fully automated using JUnit assertions and executed via Maven (`mvn test`).

- ***Integration between components*** was tested using JADE agents simulating real players and the Game Master.

  Tests verify:

  - JOIN and GAME START communication flow
  - message exchange using ACL protocol

  Additionally, fault tolerance scenarios were tested, including:

  - GameMaster crash and BackupMaster takeover
  - heartbeat-based liveness detection
  - player disconnection handling

  These tests validate correctness of distributed interaction and robustness under failure conditions. To obtain automation, agents are dynamically created inside JUnit tests using JADE `AgentContainer`. Synchronization is achieved using blocking queues and shared test registries. Requirements covered: distributed communication, fault tolerance, game start synchronization, player lifecycle management.

- A full ***end-to-end*-test** of an entire Exploding Kittens match (from player join to game completion) was not implemented.

  The main reason is the high complexity and non-deterministic nature of the game logic, which makes it difficult to simulate a complete match in an automated and reproducible way.

Instead, the system was validated through **scenario-based integration tests**, which cover complete execution paths of critical functionalities, such as:

- player joining and game initialization
- game start when the required number of players is reached
- disconnection handling and automatic win assignment
- GameMaster failure and BackupMaster takeover

These tests execute the system in a production-like JADE runtime environment, ensuring that all major components interact correctly under realistic conditions.

5.2 Acceptance test

In addition to automated testing, a set of **manual acceptance tests** was performed to validate the system from an end-user perspective and to verify complete execution flows in realistic conditions.

The following end-to-end scenarios were manually executed:

- complete game lifecycle (player join, game start, turn progression, and game termination)
- lobby behavior and player registration
- correct execution of game flow and turn management
- usage of cards during real gameplay sessions
- system behavior under failure conditions (GameMaster crash and recovery)
- BackupMaster takeover and continuation of the game session

These tests effectively covered **end-to-end system behavior**, since they involved all system components interacting together in a realistic execution environment.

Full automation of end-to-end scenarios involving JADE agents is challenging due to several inherent characteristics of multi-agent systems. In particular:

- JADE agents communicate asynchronously through message passing, making execution order and timing non-deterministic
- system behavior depends on concurrent interactions among multiple agents, leading to emergent and hard-to-predict outcomes
- timing-related mechanisms (e.g., heartbeats and timeouts) introduce variability in test execution
- some agents may interact with graphical user interfaces, which can block automated execution in non-headless environments

- simulating a full game requires implementing complex decision-making logic for agents, effectively introducing an additional artificial AI layer
- long-running interactions reduce the practicality of fully automated regression testing

For these reasons, manual end-to-end validation was preferred to ensure correctness of the overall system behavior and user experience.

6 Deployment

This section describes how to install and run the system from scratch, including required software and configuration steps.

- **Installation and execution**
  - Download the latest release of the project as a compressed archive and extract it locally.
  - To start the system, one user must act as the **host** and launch the server components, while the other users run the clients.
  - **Step 1 — Start the main server (host only):**
    - On Windows: double-click `ServerMain.bat`
    - On Linux/Mac:

      ```
      chmod +x ServerMain.sh
      ./ServerMain.sh
      ```

  - **Step 2 — Start the backup server (host only):**
    - On Windows: double-click `BackupServerMain.bat`
    - On Linux/Mac:

      ```
      ./BackupServerMain.sh
      ```

  - **Step 3 — Start the clients:**
    - Each player (2 to 4) launches the client on the same machine:
      - Windows: `ClientMain.bat`
      - Linux/Mac:

        ```
        ./ClientMain.sh
        ```

  - **Expected outcome:**
    - The server starts and waits for incoming players.
    - Each client opens a graphical interface where the user can choose a nickname and the number of players (only for the first player joining the match)
    - Once all players are connected, the game starts automatically.
- **Software requirements**
  - **Java 21** (or higher) must be installed on all machines.
  - No additional libraries or external services are required at runtime, as all dependencies are bundled in the distributed application.

- **Configuration**
  - No manual configuration files are required.
  - No environment variables need to be set.
  - All players must be connected to the **same network**, since the system relies on local network communication.
  - The provided scripts must remain in the same directory as the `.jar` file.
- **Containerization**
  - The system does not use containerization technologies such as Docker.
  - Deployment is performed through platform-specific scripts (`.bat` and `.sh`) for simplicity.

7 USER GUIDE

7.1 Starting the System

To launch the game, the following components must be started in order:

1. **ServerMain** - upon startup, the JADE GUI will appear, which can be used to monitor the agents and to test the fault tolerance of the system.

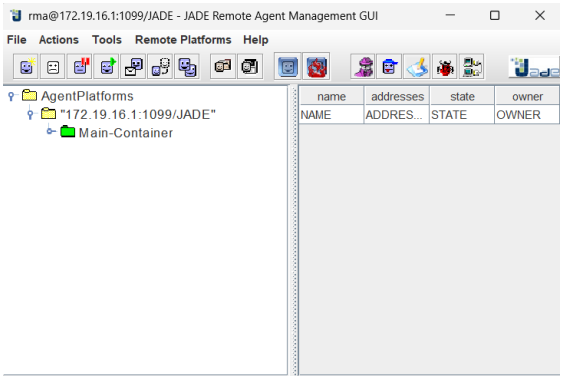

Figure 11: GUI JADE

2. **BackupServerMain** - the backup server agent registers itself and stands by to take over in case the main server fails.

3. **Clients** - each client corresponds to one player. The **first** client to connect will be prompted to select the number of players expected for the match (between 2 and 4).

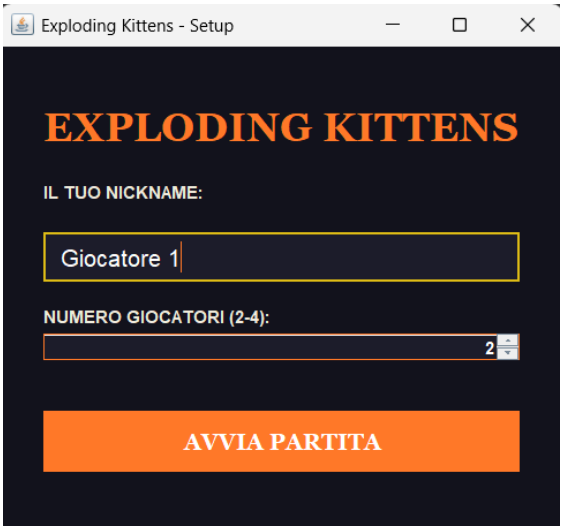

Figure 12: Client 1 Setup view

7.2 Player Registration

Every player must enter a **unique nickname** upon joining. The initial window will notify the user if the lobby is already full or if the chosen nickname is already taken by another player.

Nickname uniqueness is enforced for usability reasons: certain cards, such as *Cat Card*, require the acting player to select a target among the opponents. If two players share the same nickname, unambiguous target selection would not be possible.

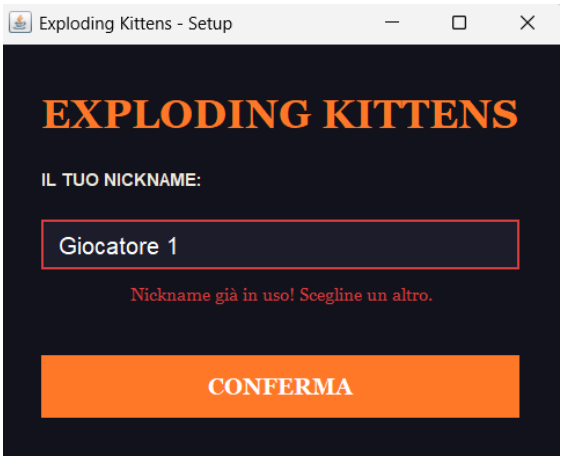

Figure 13: Nickname Error

Figure 14: Lobby Error

Once registration is successful, the player will see either:

- a **waiting screen**, informing them that the game has not yet started because not enough players have joined;
- or directly the **game screen** with their hand of cards, if the required number of players has already been reached.

7.3 Gameplay

Once all players have joined, the game begins. On each turn, the **active player** can either:

- **play** one or more of the cards in their hand
- or **draw** a card from the deck to end their turn.

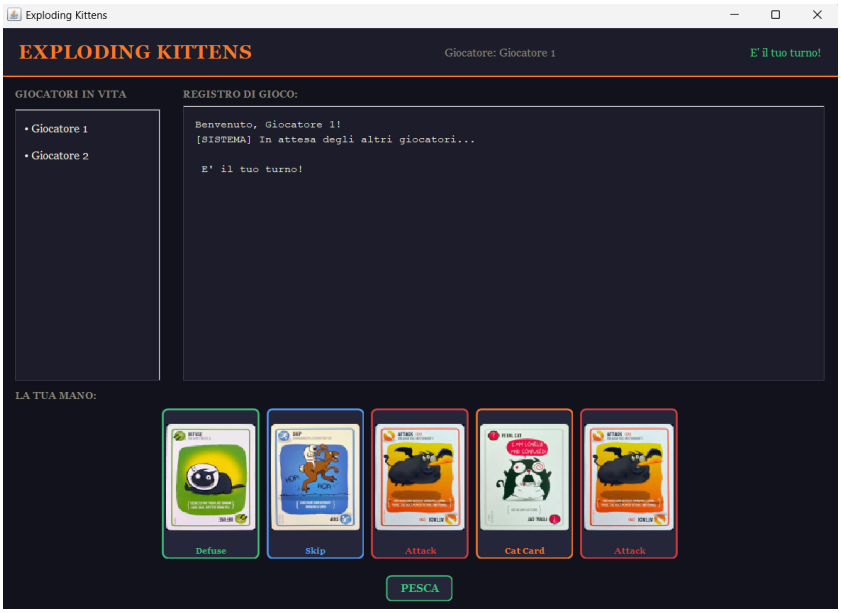

Figure 15: Active player

All other players are in a passive state: the draw button is disabled and no card can be played outside of one's own turn. The **left panel** of the game window displays the list of players still alive, updated in real time throughout the match.

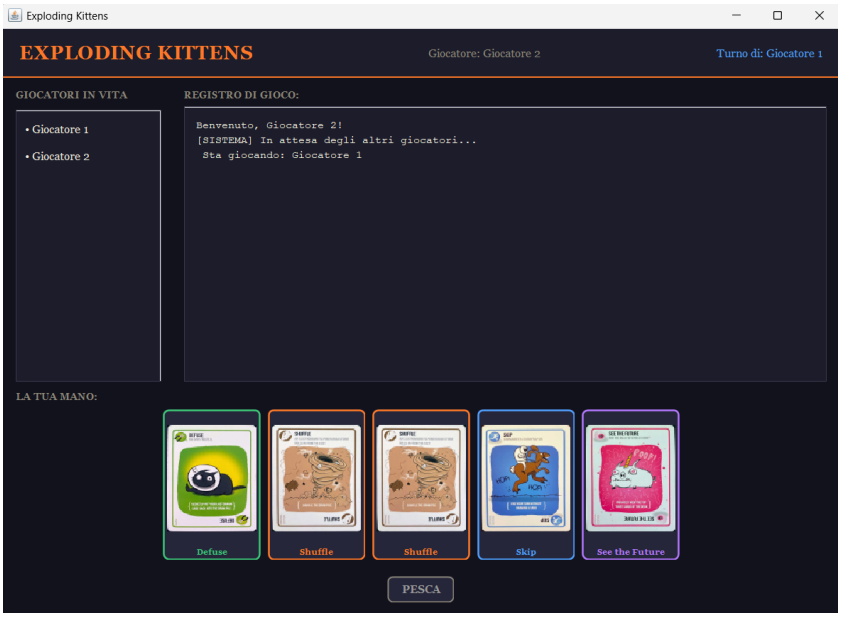

Figure 16: Player outside turn

7.4 Card Effects

See the Future - When a player plays a *See the Future* card, the top three cards of the deck are shown in the game log. This information is visible only to the player who played the card.

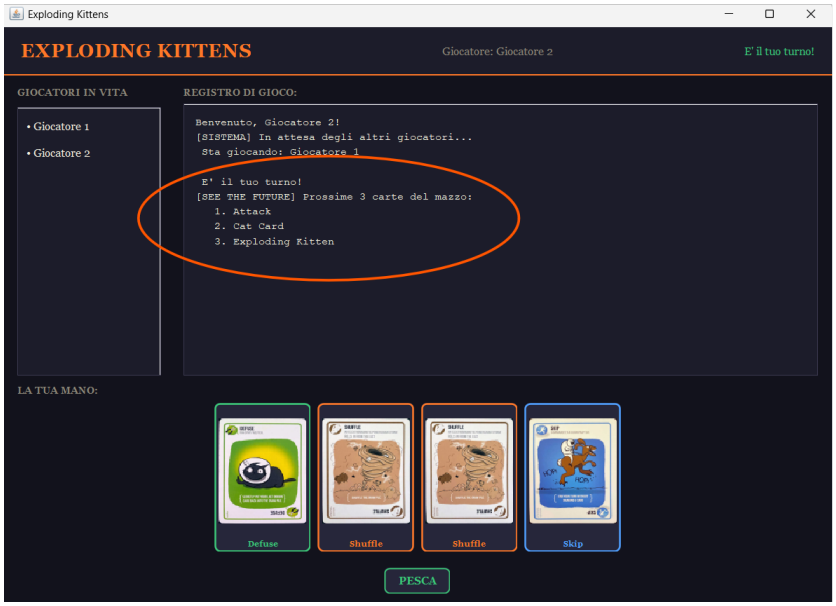

Figure 17: See the future effect

Cat Card - A *Cat Card* can only be played if the player holds **at least two** Cat Cards in hand. The system checks this condition before applying the effect:

- If the player holds two or more Cat Cards, a dialog window appears allowing them to select the opponent from whom to steal a card. After confirming the selection, a randomly chosen card is removed from the target's hand and added to the acting player's hand.
- If the player does not hold enough Cat Cards, they are notified that the card cannot be played.

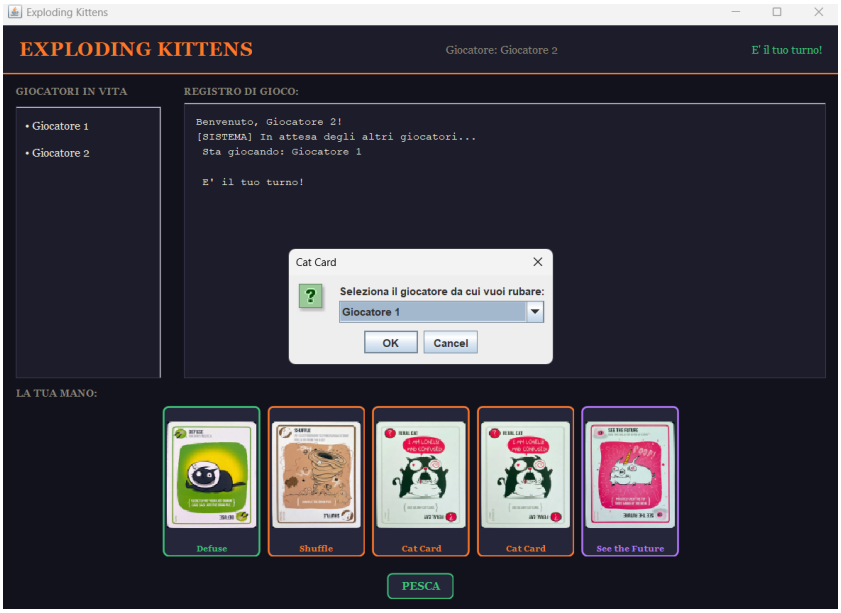

Figure 18: Choosing target

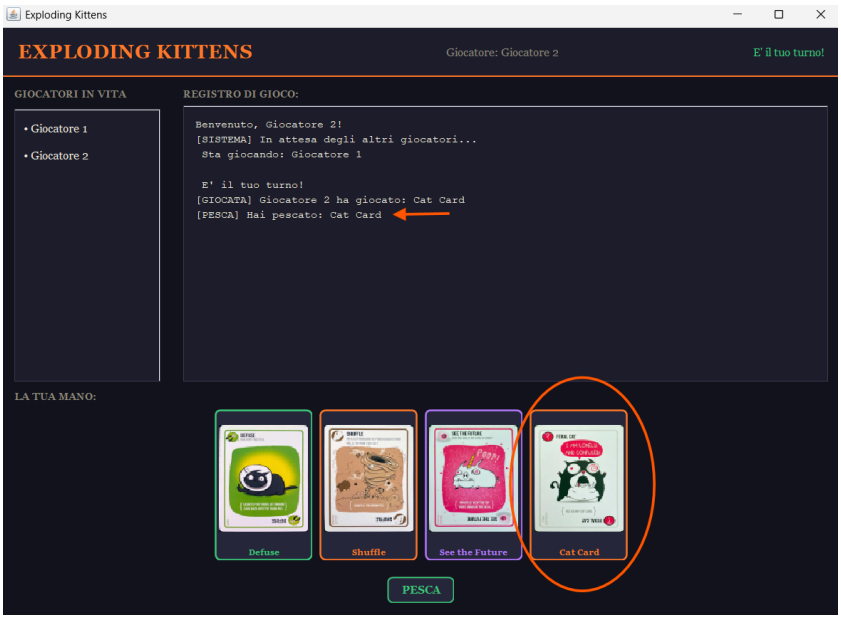

Figure 19: Card stolen using Cat cards

Attack - Playing an *Attack* card immediately ends the current player's turn. The next player in line must then play **two consecutive turns** before passing.

Shuffle - Playing a *Shuffle* card reshuffles the deck. To verify that the effect is applied correctly, one can play a *See the Future* card to observe the top of the deck, then play *Shuffle*, and finally draw a card to confirm that the drawn card differs from the one previously seen at the top (unless the deck happens to contain duplicate cards).

Skip - Playing a *Skip* card ends the current player's turn immediately, without drawing a card from the deck.

Defuse - The *Defuse* card is played **automatically** when a player draws an *Exploding Kitten*. Every player starts the game with at least one Defuse card in hand.

If a player draws an *Exploding Kitten* and holds a *Defuse*, they survive and must choose a position in the deck (from 0 to *deckSize*) in which to reinsert the *Exploding Kitten*. The used *Defuse* card is removed from their hand.

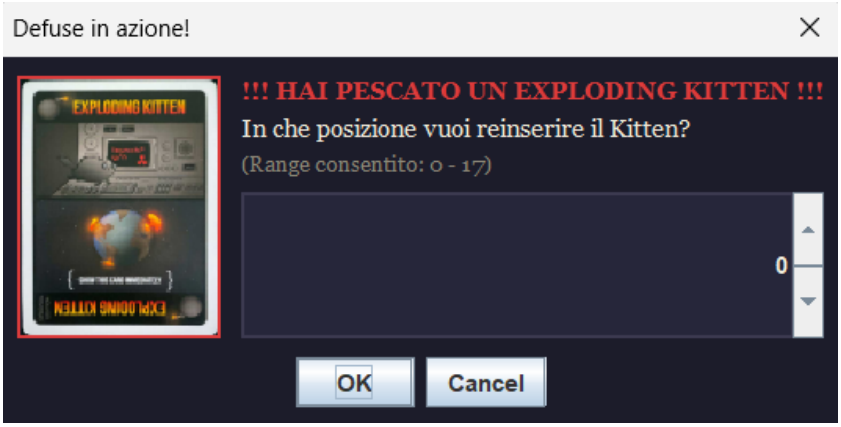

Figure 20: Defuse effect after drawing Exploding Kitten card

If the player has no *Defuse*, they are **eliminated** from the game and the *Exploding Kitten* is removed from the deck permanently.

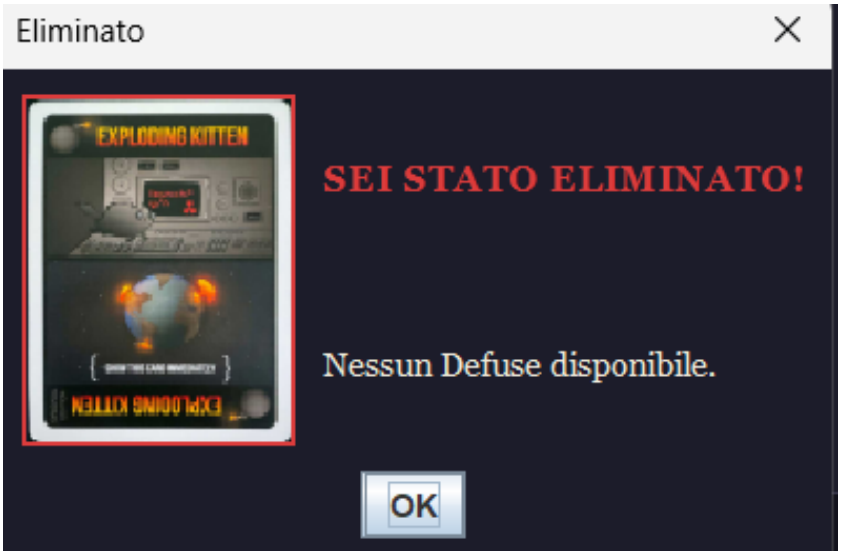

Figure 21: Player eliminated

7.5 Winning Condition

The last player remaining alive wins the match.

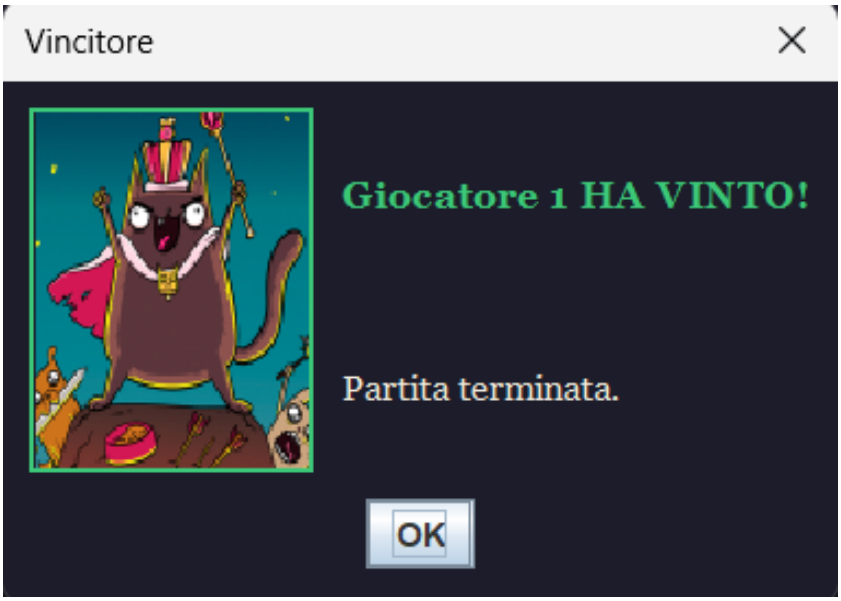

Figure 22: Winner player

7.6 Player Disconnection

If a player disconnects during the game, they are treated as eliminated. The remaining players continue the match normally. If only one player remains after a disconnection, that player is immediately declared the winner.

7.7 Testing Fault Tolerance

To test the fault tolerance mechanism, the JADE GUI (launched together with `ServerMain`) can be used to manually kill the `GameMasterAgent`. Within a few seconds, all connected players should see a notification in their game log informing them that the backup server has taken over. After that, the game resumes normally and players can continue their match as before.

> *Note:* the choice to explicitly notify players in the log when the backup takes over is intentional, and is meant solely to simplify testing and verification. In a production scenario, this notification could be suppressed entirely: the failover would still occur transparently within a few seconds, with no visible interruption to the players.

8 SELF-EVALUATION

8.1 Alessandra Versari

Role. My primary areas of contribution were the `BackupServerAgent`, the fault tolerance mechanisms, and the `PlayerAgent`. I also worked on the game view (`GameView`), and on the shared game model. In practice, however, the development was highly collaborative throughout the project: both members of the group contributed to all parts of the codebase, and the distinction in roles reflects areas of primary responsibility rather than exclusive ownership.

Strengths. Overall, I am satisfied with the result. The game interface, while intentionally simple, is intuitive and easy to use. The in-game log effectively communicates all relevant events to the player, making the state of the match transparent at all times. I am particularly pleased with the fault tolerance implementation: both client-side and server-side failure scenarios are handled, and the system recovers gracefully in most cases.

Weaknesses. The main aspect I am less satisfied with is the latency introduced by the failover mechanism: when a client or the server crashes, the remaining players experience a few seconds of inactivity before the game can resume. This is an inherent consequence of the heartbeat-based detection approach, and could be mitigated with a more reactive failure detection strategy. Given more time, I would have invested further effort in refining the GUI, expanding the set of available cards to make the game more engaging, and increasing the overall test coverage of the system.

8.2 Margherita Balzoni

Role. My primary areas of contribution were the automated test suite, the core game logic, the `GameMasterAgent`, and the registration flow (`SetupView`). As with all parts of the project, development was collaborative and both members contributed across the entire codebase; the areas listed above reflect where I took primary responsibility.

Strengths. I am generally satisfied with the final product. The game logic is correctly implemented and the interaction flow (from registration to gameplay) is smooth and easy to follow. The log provides clear, real-time feedback on every action taken during the match, which also proved useful during testing and debugging. The fault tolerance support, developed jointly, works reliably across the scenarios we tested.

Weaknesses. The test suite, while functional, could be significantly expanded: the distributed and asynchronous nature of the JADE platform makes automated testing challenging, and some interaction scenarios remain covered only by manual testing. Given more time, I would have strengthened the test coverage, improved the visual

quality of the interface, and enriched the card set to make the gameplay more varied and replayable.

9 FUTURE WORKS

There are several possible improvements and extensions that can be considered for the system:

- First, the **graphical user interface** could be significantly enhanced. In the current implementation, the GUI has been intentionally simplified in order to focus on the design and implementation of the distributed system aspects within the available time constraints. A more polished interface, including animations, better visual feedback for game events, and improved usability, would greatly enhance the overall user experience.
- Another important extension concerns the **game content** itself. The system could be enriched by adding additional cards from the official game and its expansions. This would make gameplay more varied, balanced, and engaging.
- Additionally, **fault tolerance mechanisms** could be enhanced by improving failure detection and introducing state recovery strategies for interrupted games.
- Finally, the introduction of **artificial players** (AI agents) with different strategies could allow games to be played even without human participants, as well as providing a useful tool for testing and balancing the game dynamics.

10 REFERENCES

- JADE documentation
- JADE network protocol