Software Design Specifications

for

Lama

Version 1.0 approved

Prepared by your Software Engineering TAs

LamaDrama

25.03.2022

Table of Contents

Revision History

Name	Date	Release Description	Version
Felix Friedrich	25.3.2022	Template for Software Engineering Course in ETHZ.	0.2
	25.3.2022	Example SDS	1.0

Introduction

Purpose

<Identify the product whose software requirements are specified in this document, including the revision or release number. Describe the scope of the product that is covered by this SDS, particularly if this SDS describes only part of the system or a single subsystem.>

Document Conventions

<Describe any standards or typographical conventions that were followed when writing this SDS, such as fonts or highlighting that have special significance. For example, state whether priorities for higher-level requirements are assumed to be inherited by detailed requirements, or whether every requirement statement is to have its own priority.>

Intended Audience and Reading Suggestions

<Describe the different types of reader that the document is intended for, such as developers, project managers, marketing staff, users, testers, and documentation writers. Describe what the rest of this SDS contains and how it is organized. Suggest a sequence for reading the document, beginning with the overview sections and proceeding through the sections that are most pertinent to each reader type.>

Product Perspective

<Describe the context and origin of the product being specified in this SDS. For example, state whether this product is a follow-on member of a product family, a replacement for certain existing systems, or a new, self-contained product. If the SDS defines a component of a larger system, relate the requirements of the larger system to the functionality of this software and identify interfaces between the two. A simple diagram that shows the major components of the overall system, subsystem interconnections, and external interfaces can be helpful.>

Static Modeling

Package Common

The package common contains code for both client and server. This includes classes to encode the state of a Lama game as well as requests and responses for client-server-communication. Figure 1 shows an overview of the package common.

All classes additionally implement a from_json and write_into_json operation, that allows them to be written into and read from json strings, which will be used for encoding the game state into JSON format when sending messages between server and client.

Class card

Represents a card in the Lama game.

The class attributes are:
- value: int, card number from 1 to 6 or 7 for LAMA card

The class operations are:
- can_be_played_on: bool, checks if this card can be played on another according to the game rules

Class hand

Represents the hand of a player.

The class attributes are:
- cards: vector<card*>, hand cards

The class operations are:
- try_get_card: bool, tries to find a specific card in the hand

  Server only:
- setup_round: prepares hand for the next round (removes all cards from hand)
- add_card: adds card to hand
- remove_card: bool, attempts to remove card from hand

Class player

Represents a player of the Lama game.

The class attributes are:
- player_name: string, username chosen by the player
- has_folded: bool, if player has folded in the current round
- score: int, current number of minus-points
- hand: hand
- game_id: string, ID of the game that the player has joined (server may operate multiple games at the same time)

The class operations are:

- get_score: int
- fold: bool, player attempts to fold this round
- add_card: bool, attempts to add a card to the player's hand
- remove_card: bool, attempts to remove a card from the player's hand
- wrap_up_round: computes minus-points and updates player score according to rules
- setup_round: resets player state and removes all cards from his hand

Class discard_pile

Represents a discard pile, stores cards and offers a function to place cards.

The class attributes are:

- cards: vector<card*>, cards on discard pile

The class operations are:

- can_play: bool, returns if a card can be played on the discard pile
- setup_game: removes all cards from pile
- try_play: bool, attempts to place card (potentially from players hand) on top of the discard pile

Class draw_pile

Represents a draw pile, stores cards and offers option to draw cards.

The class attributes are:

- cards: vector<card*>, cards on draw pile

The class operations are:

- setup_game: fills pile with all cards of the game (shuffled)
- draw: bool, attempts to draw a card and to place it in a player's hand
- remove_top: card*, removes and returns the card on top of the draw pile

Class game_state

Holds the state of one game, executes and checks game state modifications.

The class attributes are:

- max_nof_players: int, set to 6
- min_nof_players, int, set to 2
- players: vector<player*>, players that joined the game
- draw_pile: draw_pile
- discard_pile: discard_pile
- is_started: bool
- is_finished: bool
- round_number: int
- current_player_idx: int, ID of the player who's turn it currently is

The class operations are:

- is_full: bool, returns if all slots in the game are taken (6 players joined)
- is_player_in_game: bool, checks if the requested player is in the game
- get_current_player: player*, returns player who's turn it is
- remove_player: bool, removes a player from the game if present
- add_player: bool, adds player to the game if possible
- start_game: bool, starts game if possible, initiates first round

- draw_card: bool, attempts to draw a card and place it in a given player's hand
- play_card: bool, attempts to place a card from a player's hand on the discard pile
- fold: bool, attempts to perform a fold for a given player
- update_current_player: determines the next valid player to make a move, updates it in game state
- setup_round: sets up round: updates round number, resets player's hands, resets the draw and discard pile and draws 6 cards for each player
- wrap_up_round: wraps up a round: computes and updates player scores, determines if game has ended and if not, determines next player to make a move

Class client_request

Base class for client requests to the server. For more details about the different requests, please refer to the Interface Modeling section of this document.

The class attributes are:
- player_id: string, ID of the player whose client makes the request
- game_id: string, ID of the game joined by the requesting client, if applicable
- type: enum RequestType, either join_game, start_game, play_card, draw_card or fold

The different request types may add additional fields and will be realized as subclasses of client_request, specifically as:
- join_game_request
- start_game_request
- play_card_request
- draw_card_request
- fold_request

Class server_response

Base class for server communication to the client. For more details about the different responses, please refer to the Interface Modeling section of this document.

The class attributes are:
- game_id: string, ID of the affected game
- type: enum ResponseType, either req_response (as answer to a client request) or full_state_msg (to communicate changes in the game state)

The different response types may add additional fields and will be realized as subclasses of server_response.

Class Diagram of Package Common

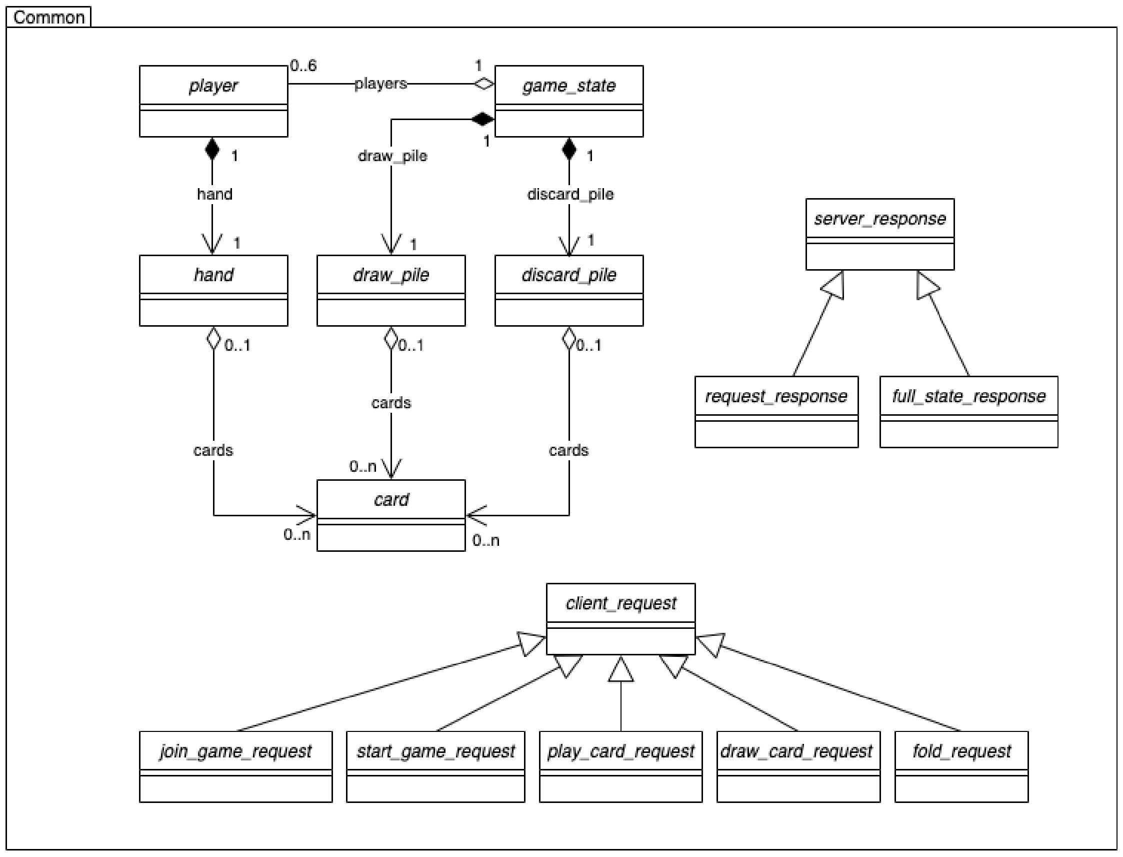

Figure 1. Class diagram of package Common.

Package Client

This package contains the classes and methods necessary to run the client and play a game of Lama. The code covers the GUI, user interaction with the client, sending the user input to the server and receiving server responses.

Class Lama

Provides an entry point to the client application. This class is derived from the wxApp class defined in the wxWidgets GUI library.

The class operations are:

- OnInit: application entry point

Class GameController

The GameController is the central controller of the client, it handles events, triggers the server connection, and deals with the interaction of the user with the GUI.

The class attributes are:

- gameWindow: main game window holding current panel
- connectionPanel: connection panel used to connect to the game
- mainGamePanel: the main game panel used in the game
- me: the player interacting with the client
- currentGameState: the latest game state

The class operations are:

- init: initializes all panels and displays the connection panel
- connectToServer: reads the user inputs on the connection panel and connects to game session
- updateGameState: saves the latest game state
- startGame: sends a start game request to the server and then starts a game if all conditions are fulfilled
- drawCard: sends a draw card request to the server
- fold: sends a fold request to the server
- playCard: sends a play card request to the server
- showError: prints an error message box to the client game window
- showStatus: shows the status specified in the status bar
- showNewRoundMessage: displays a dialog box showing the obtained points and a button to start the next round
- showGameOverMessage: displays a dialog box showing the winning/losing message and a button to leave the game

Class GameWindow

Main window of the application, responsible for the display of panels. This class is derived from the wxFrame class defined in the wxWidgets GUI library.

The class attributes are:

- currentPanel: the panel to be displayed in the game window
- statusBar: displays the connection status to the user

The class operations are:
- showPanel: switches view panel displayed
- setStatus: sets status to display in status bar

Class ConnectionPanel

Panel containing the connection GUI, displays input fields for the user to join a game with the client, derived from the wxPanel class defined in the wxWidgets GUI library.

The class attributes are:
- serverAddressField: holds the address of the server to connect to
- serverPortField: holds the port on the server to connect to
- playerNameField: holds the username picked by the player

The class operations are:
- getServerAddress: returns the address of the server to connect to
- getServerPort: returns holds the port on the server to connect to
- getPlayerName: returns the username picked by the player

Class MainGamePanel

Panel containing the in-game GUI, displayed when starting and playing a game in the client, derived from the wxPanel class defined in the wxWidgets GUI library.

The class operations are:
- buildGameState: removes existing GUI elements and builds latest game state GUI
- buildOtherPlayer: builds the GUI elements of the other player’s hand and labels
- buildCardPiles: builds the discard and draw piles
- buildTurnIndicator: builds the turn indicator
- buildThisPlayer: builds the hand and label of the player

Class ClientNetworkManager

Handles server-client communication on the client side.

The class attributes are:
- connectionSuccess: bool, indicates if successfully connected to host
- failedToConnect: indicates if failed to connect to host
- connection: TCP connector used to connect to the host and to initialize the response listener thread

The class operations are:
- init: creates a connection to a host
- sendRequest: sends a client request to the connected host
- parseResponse: parses a received server response for further processing

Class ResponseListenerThread

Listener thread to catch incoming server responses.

The class attributes are:
- connection: TCP connector on which the listener listens for incoming responses

The class operations are:

- Entry: threaded loop which deals with incoming server responses
- outputError: communicates error to the user

Class Diagram of Package Client

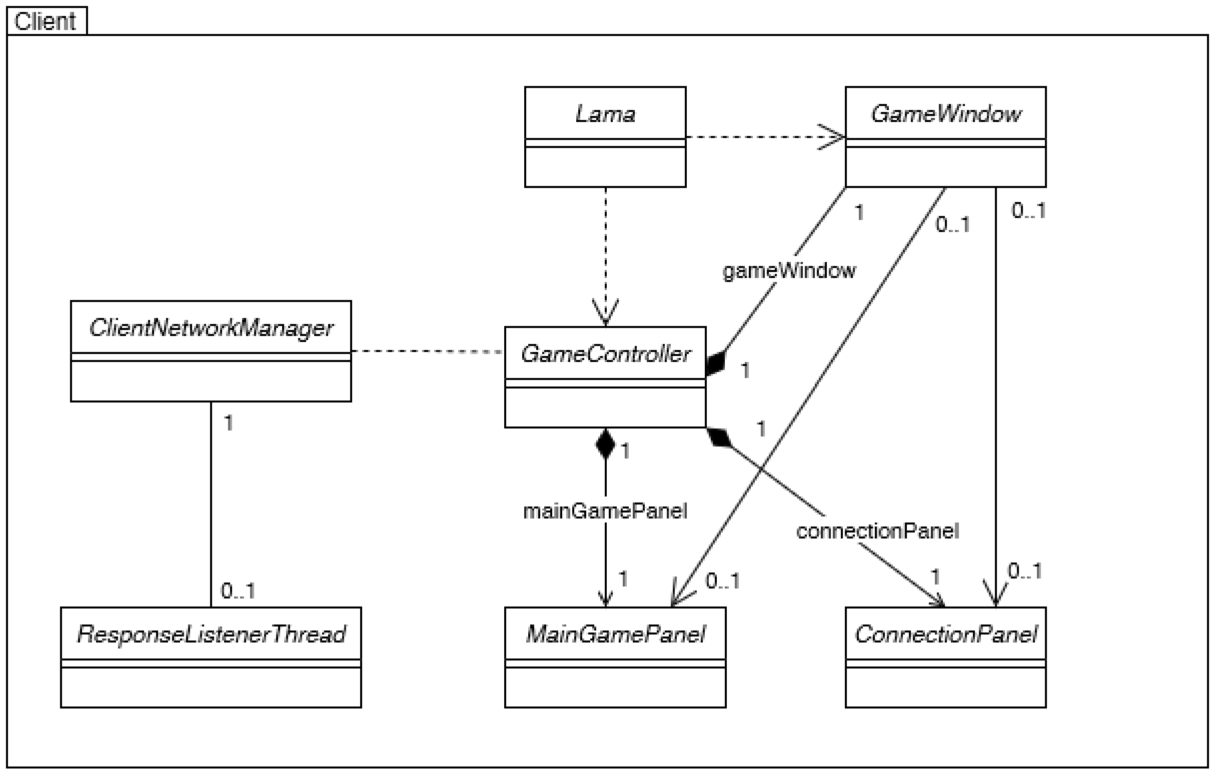

Figure 2. Class diagram of package Client.

Package Server

This package contains all classes and methods necessary to host Lama games, which includes managing the networking, game states and the players. These methods are meant to be run server-side. The package supports starting and maintaining game instances, enforcing the game rules, and keeping each client's game state up to date. An overview of the package server can be seen in Figure 3.

Class server_network_manager

Handles server startup, client requests and broadcasting information to all clients. After the startup the server will execute a listener loop and handle incoming requests from the clients.

The class attributes are:
- acc: a TCP acceptor socket for incoming connection requests
- player_id_to_address: maps the player ids to client addresses
- address_to_socket: maps the client addresses to TCP sockets

The class operations are:
- listener_loop: keeps the server running and catches incoming requests
- handle_incoming_message: receives a message and checks the contents
- read_message: parses a received message for further processing
- send_message: sends a message to a client
- broadcast_message: causes a send to all clients
- on_player_left: handles the event that a player quits the game

Class player_manager

Handles player management during a game instance.

The class attributes are:
- players: map, keeps track of the players and their names

The class operations are:
- try_get_player: retrieves a player form the player list
- add_or_get_player: adds new player to the player list or retrieves the existing player if present
- remove_player: removes the player form the player list

Class game_instance

Tool to maintain a game session. This includes keeping track of the game state and ensuring that the game state is kept up to date on the clients by passing the updated information to the server_network_manager instance.

The class attributes are:
- game_state: game_state

The class operations are:
- is_full: bool, returns if 6 players are in the game
- is_started: bool, returns if the game is underway
- is_finished: bool, returns if the game is terminated
- try_add_player: bool, if possible adds a player to the game
- try_remove_player: bool, if possible removes a player from the game
- start_game: bool, attempts to start the game

- play_card: bool, plays the chosen card and updates the game_state if it is according to the rules
- draw_card: bool, draws a card for the given player and updates the game_state
- fold: bool, a fold is declared for the given player and the game_state is updated

Class game_instance_manager

Manages game instances. Makes hosting multiple game instances is possible.

The class attributes are:
- games_lut: map, stores all game instances and their IDs

The class operations are:
- create_new_game:game_instance*, creates a new game instance
- try_add_player_to_any_game: bool, tries to add player to a non-full, non-started game instance
- try_get_game_instance: bool, tries to return a game instance given the ID
- find_joinable_game_instance : game_instance*, finds game instance that can be joined, creates new one if none exists
- try_remove_player: bool, removes a certain player from a game instance

Class request_handler

Handles received client requests of the types: join_game, start_game, play_card, draw_card or fold.

The class operations are:
- handle_request: request_response*, handles a client_request, changing the game state and returning a corresponding response

Class Diagram of Package Server

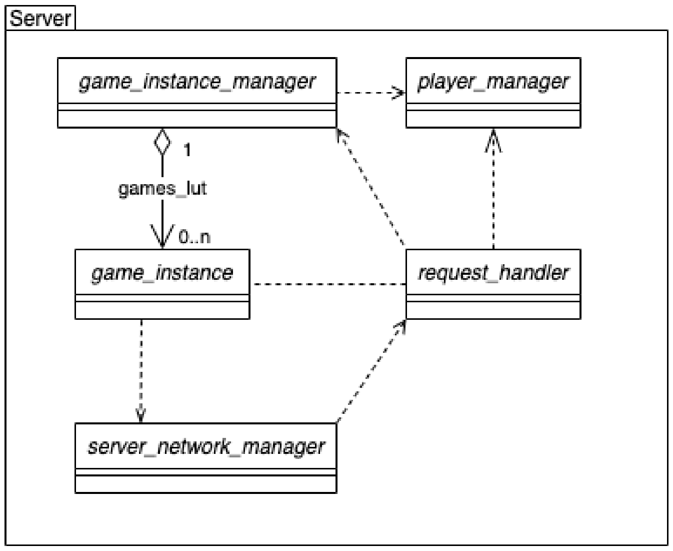

Figure 3. Class diagram of package Server.

Composite Structure Diagram

Figure 4 provides an overview of the Lama application structure: Both, the Client and the Server package, use parts of the Common package. Client App and Server App communicate through a TCP connection, whereas the user interacts with the Client App through the Client App's GUI and the hardware input devices provided.

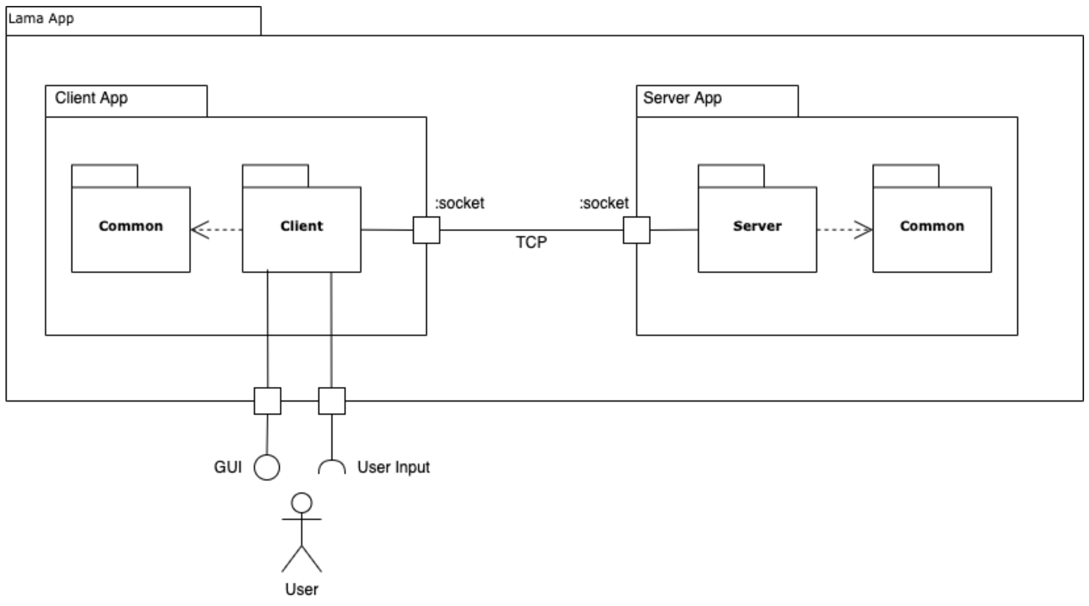

Figure 4. Composite structure diagram of Lama application

Sequence Diagrams

Sequence Join Game

Player tries to join a game.

The functional requirements related to this sequence are:

- FREQ-1: Game Server
- FREQ-2: Connection
- FREQ-3: Lobby
- FREQ-4: GUI
- FREQ-9: User Input

The scenarios which are related to this sequence are:

- SCN-1: Setting up a game

Scenario Narration:
The player clicks on 'connect' after filling out the required fields in the client. The connection request is sent to the server where it is checked. The server tries to assign the client to a game instance and reports back to the client.

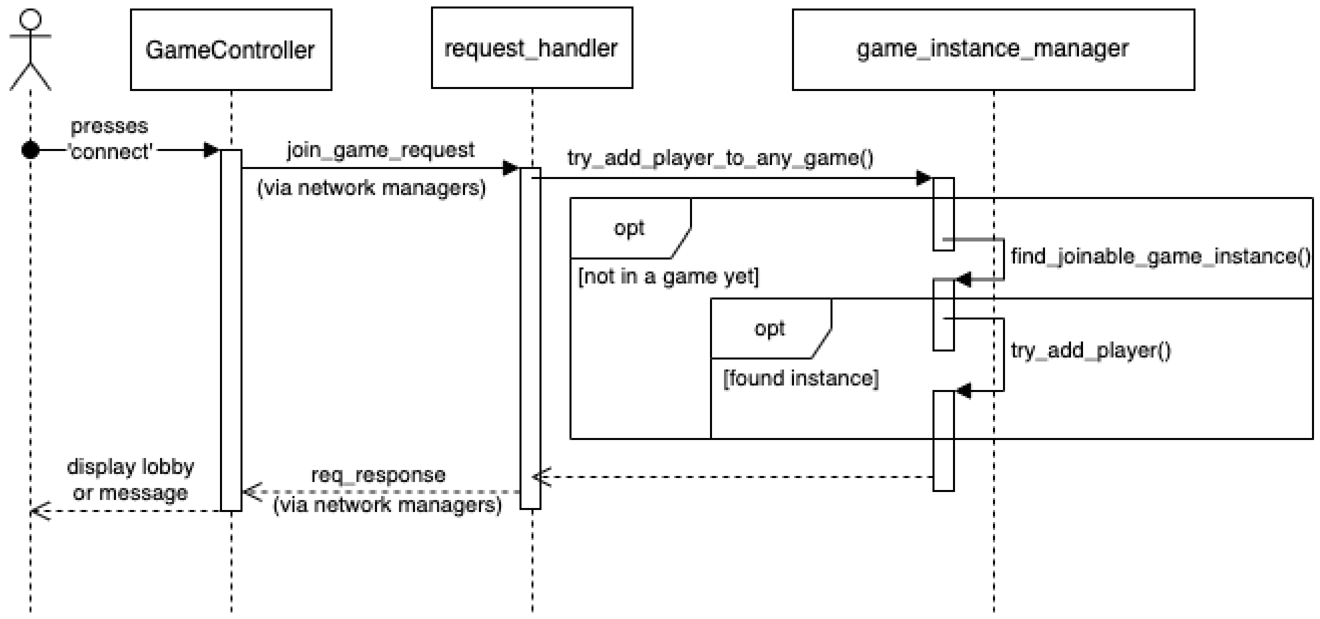

Figure 5. Sequence diagram of sequence Join Game.

Sequence Play Card

Player plays a card from his hand onto the discard pile.

The functional requirements related to this sequence are:

- FREQ-4: GUI

- FREQ-8: Make a Move
- FREQ-9: User Input

The scenarios which are related to this sequence are:

- SCN-3: Playing a card

Scenario Narration:
The player whose turn it is decides to play a card from his hand, he presses the chosen card. The request is sent to the server where the move is executed, and the card is added to the discard pile. Using the obtained response, the client updates his game state and displays the changes.

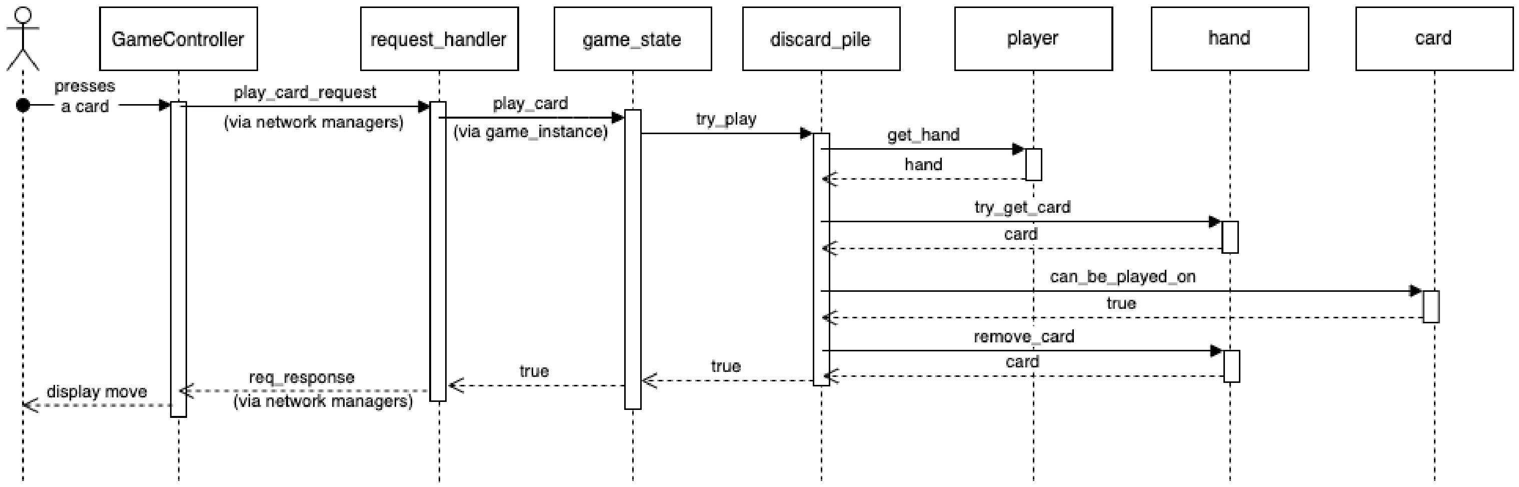

Figure 6. Sequence diagram of sequence Play Card.

Sequence End of Round

A fold move leads to the end of the current round.

The functional requirements related to this sequence are:

- FREQ-6: Round Start
- FREQ-8: Make a Move
- FREQ-10: Round End
- FREQ-11: Scores

The scenarios which are related to this sequence are:

- SCN-3: Playing a card
- SCN-4: End of game

Scenario Narration:
The Server receives a request from a player to fold. The round ends because no player is playing anymore. The scores are computed, but because the game is not over yet, a new round is initialized. The card piles are reset, and each player draws 6 cards. The game state is communicated to the client.

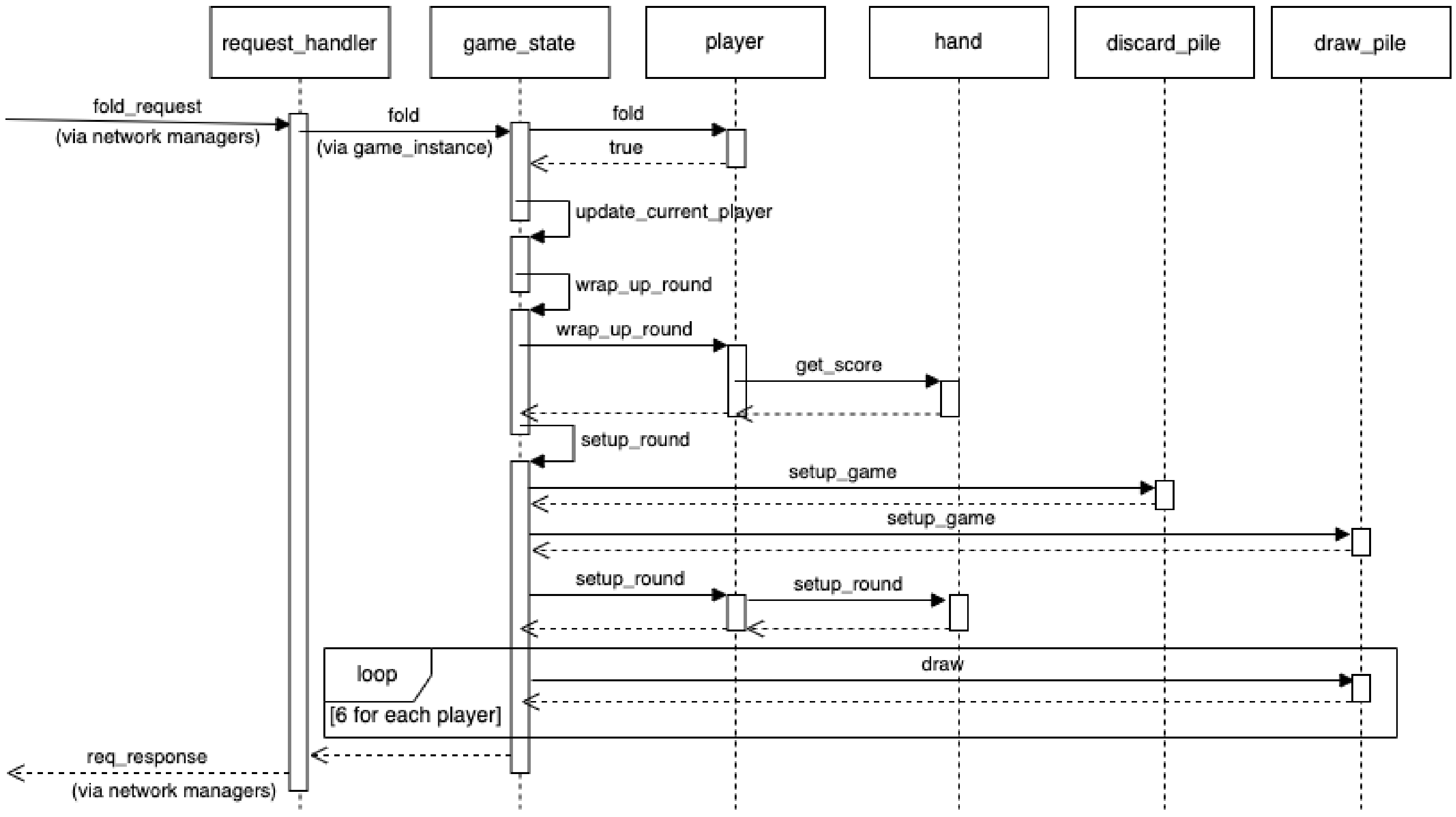

Figure 7. Sequence diagram of sequence End of Round.

Interface Modeling

Interface Client-Server

The purpose of this interface is the communication of the clients with the server. The interface allows client requests and server answers, game state updates and error communication.

Communication between: Client and Server, initiated by Client

Protocol: TCP

Communication modes: Request-Response and Broadcast of game state

The following message types are sent from the client to the server:

join_game_request

Purpose: Request to join a game as a player

Direction: Client to Server

Content:

- type: "join_game" (required)
- player_id: string (required)
- game_id: string
- player_name: string (required)

Format: as JSON string

Example:

```
{
      "type": "join_game",
      "player_id": "48e7ed2e",
      "game_id": "08a52fb2",
      "player_name": "player1"
}
```

Expected response: req_response

start_game_request

Purpose: Request to start a game

Direction: Client to Server

Content:

- type: "start_game" (required)

- player_id: string (required)
- game_id: string (required)

Format: as JSON string

Example:

```
{
    "type":"start_game",
    "player_id":"aa91a118 ",
    "game_id":"08a52fb2",
}
```

Expected response: req_response

play_card_request

Purpose: Request to play a card

Direction: Client to Server

Content:

- type: "play_card" (required)
- player_id: string (required)
- game_id: string (required)
- card_id: string (required)

Format: as JSON string

Example:

```
{
    "type":"play_card",
    "player_id":"aa91a118“,
    "game_id":"08a52fb2",
    "card_id":"90ab1dc6"
}
```

Expected response: req_response

draw_card_request

Purpose: Request to draw cards

Direction: Client to Server

Content:

- type: "draw_card" (required)
- player_id: string (required)
- game_id: string (required)
- nof_cards: int (required)

Format: as JSON string

Example:
```
{
    "type":"draw_card",
    "player_id":"aa91a118 ",
    "game_id":"08a52fb2",
    "nof_cards":1
}
```

Expected response: req_response

fold_request

Purpose: Request to fold in current round

Direction: Client to Server

Content:

- type: "fold" (required)
- player_id: string (required)
- game_id: string (required)

Format: as JSON string

Example:
```
{
    "type":"fold",
    "player_id":"efd43480",
    "game_id":"08a52fb2",
}
```

Expected response: req_response

The following message types are sent from the server to the client:

req_response

Purpose: Answer a request from a Client

Direction: Server to Client

Content:

- type: "req_response" (required)
- game_id: string (required)
- err: string
- success: bool (required)
- state_json: json string of state (required)

Format: as JSON string

Example:

```
{
    "type":"req_response",
    "game_id":"08a52fb2",
    "err":"",
    "success": true,
    "state_json":<json state encoding>
}
```

Expected response: none

full_state_msg

Purpose: Update Clients about game state changes

Direction: Server to Client

Content:

- type: "full_state_msg" (required)
- game_id: string (required)
- state_json: json string of state (required)

Format: as JSON string

Example:

```
{
    "type":"full_state_msg",
    "game_id":"08a52fb2",
    "state_json": <json state encoding>
}
```

Expected response: none