Software Design Specifications

for

Lama

Version 1.0 approved

Prepared by your Software Engineering TAs

LamaDrama

24.03.2022

Table of Contents

Revision History

Name	Date	Release Description	Version
Felix Friedrich	4.3.24	Template for Software Engineering Course in ETHZ.	0.2
	4.3.24	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 sever. This includes classes to encode a game of Lama as well as requests and responses for client-server-communication. Classes may have special functionality when used by the server.

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 generating messages between server and client.

Class card

Represents cards 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:
- get_value(): int
- can_be_played_on(other: card): bool, checks if this card can be played on another according to the game rules

Class hand

Stores the hand cards for a player.

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

The class operations are:
- get_nof_cards(): int
- get_cards(): vector<card*>
- try_get_card(c: card): bool, tries to find a specific card in the hand
- setup_round(): removes all cards from hand
- add_card(c: card): adds card to hand
- remove card(c: 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
- has_folded(): bool
- get_nof_cards(): int, returns number of cards that player is holding
- get_hand(): *hand
- get_player_name(): string
- fold(): bool, player attempts to fold this round
- add_card(c: card): bool, attempts to add a card to the player's hand
- remove_card(c: 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

The discard pile, stores cards and offers option to place cards.

The class attributes are:

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

The class operations are:

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

Class draw_pile

The 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:

- is_empty(): bool, returns if all cards have been drawn
- get_nof_cards(): int, number of cards left on the draw pile
- 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_started(): bool
- is_finished(): bool
- is_player_in_game(p: player): bool, checks if the requested player is in the game
- is_allowed_to_play_now(p: player): bool, checks if a specific player is allowed to make a move now
- get_players(): vector<player*>
- get_round_number(): int
- get_draw_pile(): draw_pile*
- get_discard_pile(): discard_pile*
- get_current_player(): player*, returns player who's turn it is
- 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
- remove_player(p: player): bool, removes a player from the game if present
- add_playe(p: player)r: bool, adds player to the game if possible
- start_game(): bool, starts game if possible, initiates first round
- draw_card(p: player): bool, attempts to draw a card and place it in a given player's hand
- play_card(p: player, c: card): bool, attempts to place a card from a player's hand on the discard pile
- fold(p: player): 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
- 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:

- req_id: string
- 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 class operations are:

- get_req_id(): string
- get_player_id(): string
- get_game_id(): string
- get_type(): enum RequestType

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 and 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 class operations are:

- get_game_id(): string
- get_type(): enum ResponseType

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

Class Diagram of Package Common

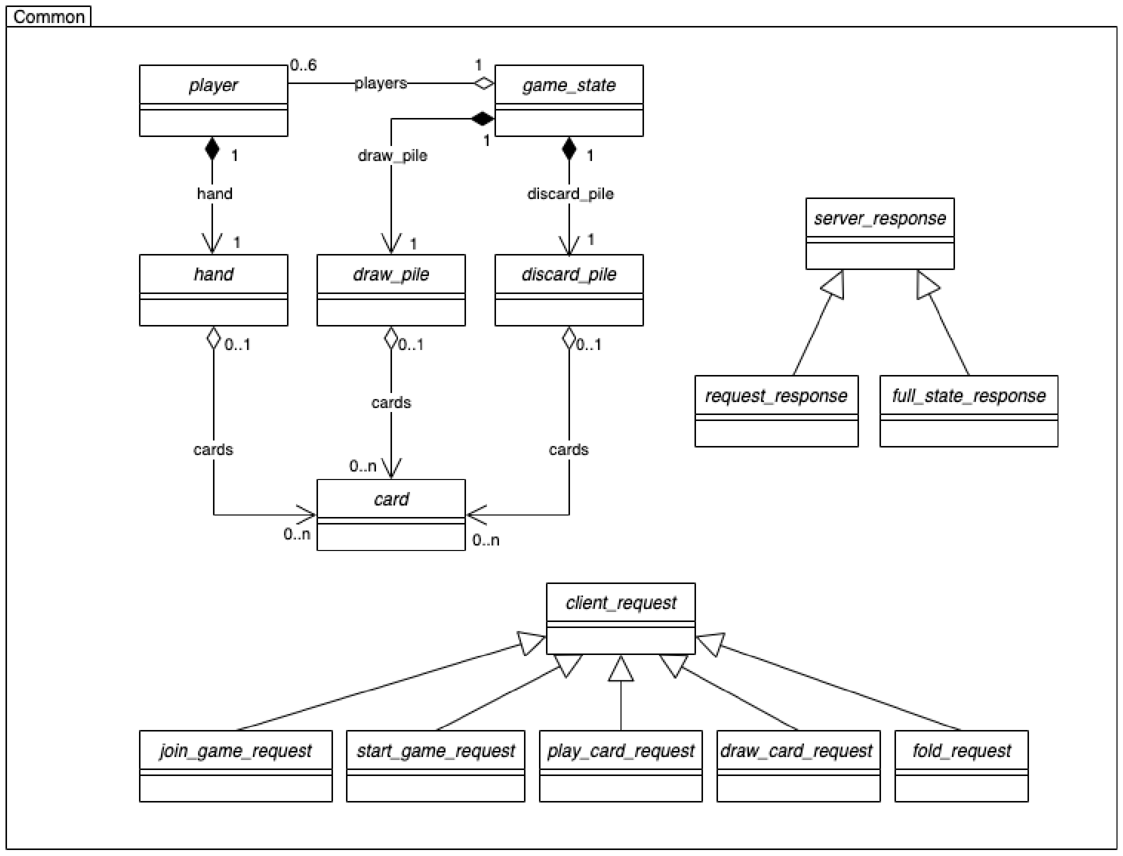

Figure 1. Class diagram of package Common.

Package Client

…

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. They start and maintain game instances, enforce the game rules, and keep each client's information up to date.

Class server_network_manager

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

The class attributes are:

- acc: tcp_acceptor, socket for incoming connection requests
- player_id_to_address: map[string, string] maps the player ids to client addresses
- address_to_socket: map[string, tcp_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(m: message): receives a message and checks the contents
- read_message(m: message): parses a received message for further processing
- send_message(m: message): sends a message to a client
- broadcast_message(m: message): causes a send to all clients
- on_player_left(p: player): handles the event that a player quits the game

Class player_manager

Handles player management during a game instance.

The class attributes are:

- lock: a lock to guarantee mutual exclusion and correctness when handling concurrent requests
- players: map, keeps track of the players and their names

The class operations are:

- try_get_player(p: player): retrieves a player form the player list
- add_or_get_player(p: player): adds new player to the player list or retrieves the existing player if present
- remove_player(p: player): removes the player form the player list

Class game_instance

Tool to maintain a game session. This includes track keeping 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 game state includes the round rules, card stacks, player cards, points, round number and the player's turn.

The class attributes are:

- game_state: game_state
- modification_lock: a lock to guarantee mutual exclusion and correctness when handling concurrent requests

The class operations are:

- get_game_state: game_state*
- 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
- start_game(): bool, attempts to start the game
- try_add_player(p: player): bool, if possible adds a player to the game
- try_remove_player(p: player): bool, if possible removes a player from the game
- play_card(c: card, p: player): bool, plays the chosen card and updates the game_state if it is according to the rules
- draw_card(c: card, p: player): bool, draws a card for the given player and updates the game_state
- fold(p: player): 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_lock: a lock to guarantee mutual exclusion and correctness when handling concurrent requests
- games_lut: map, stores all game instances and their IDs

The class operations are:

- create_new_game():game_instance*, creates a new game instance
- find_joinable_game_instance(): game_instance*, finds game instance that can be joined, creates new one if none exists
- try_get_game_instance(id: int): bool, tries to return a game instance given the ID
- try_get_player_and_game_instance(p: player, g: game_instance): bool, tries to find a given player in a given game instance
- try_add_player_to_any_game(p: player): bool, returns if a given player could successfully be added to a game instance
- try_add_player(p: player, g: game_instance): bool, tries to add a player to a given game instance
- try_remove_player(p: player, g: game_instance): 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(r: client_request): request_response*, handles a client_request, changing the game state and returning a matching response

Class Diagram of Package Server

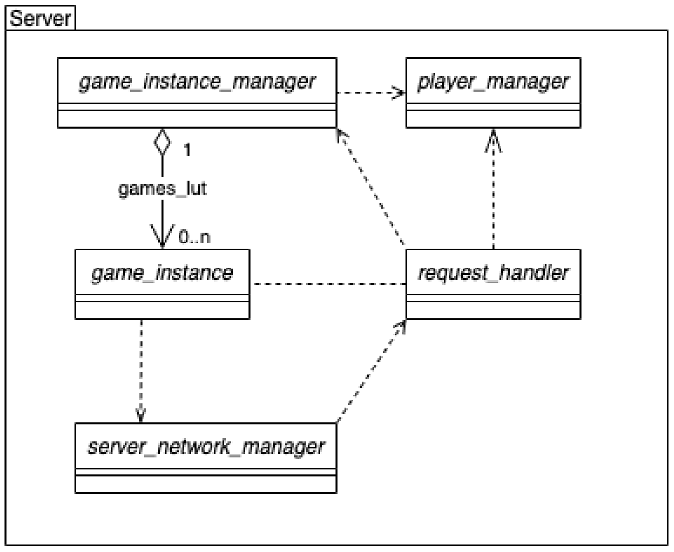

Figure 2. Class diagram of package Server.

Sequence Diagrams

Sequence Join Game

Player tries to join into game lobby.

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 into a game lobby and reports back to the client.

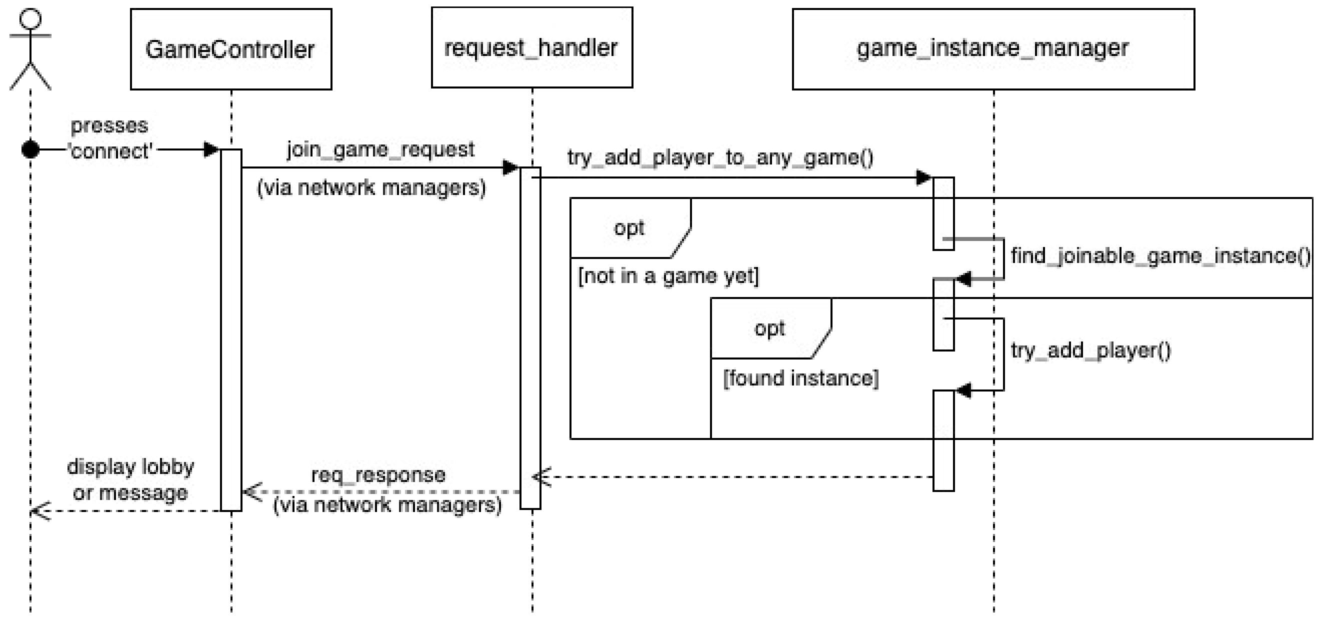

Figure 3. Sequence diagram of sequence Join Game.

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 who's 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.

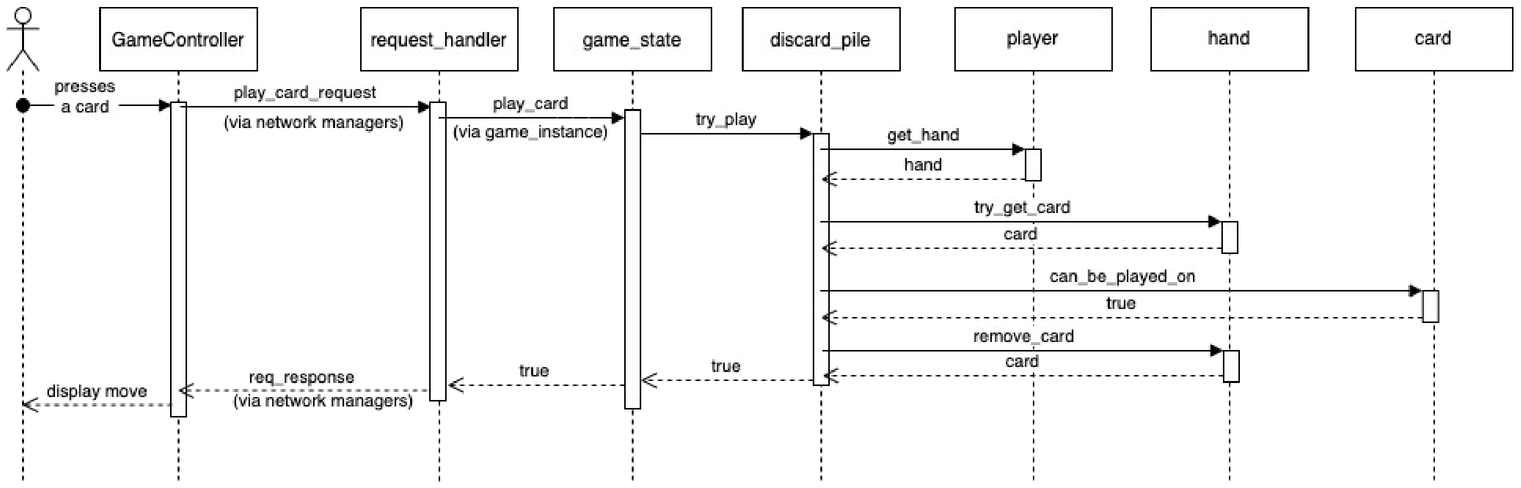

Figure 4. Sequence diagram of sequence Play Card.

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.

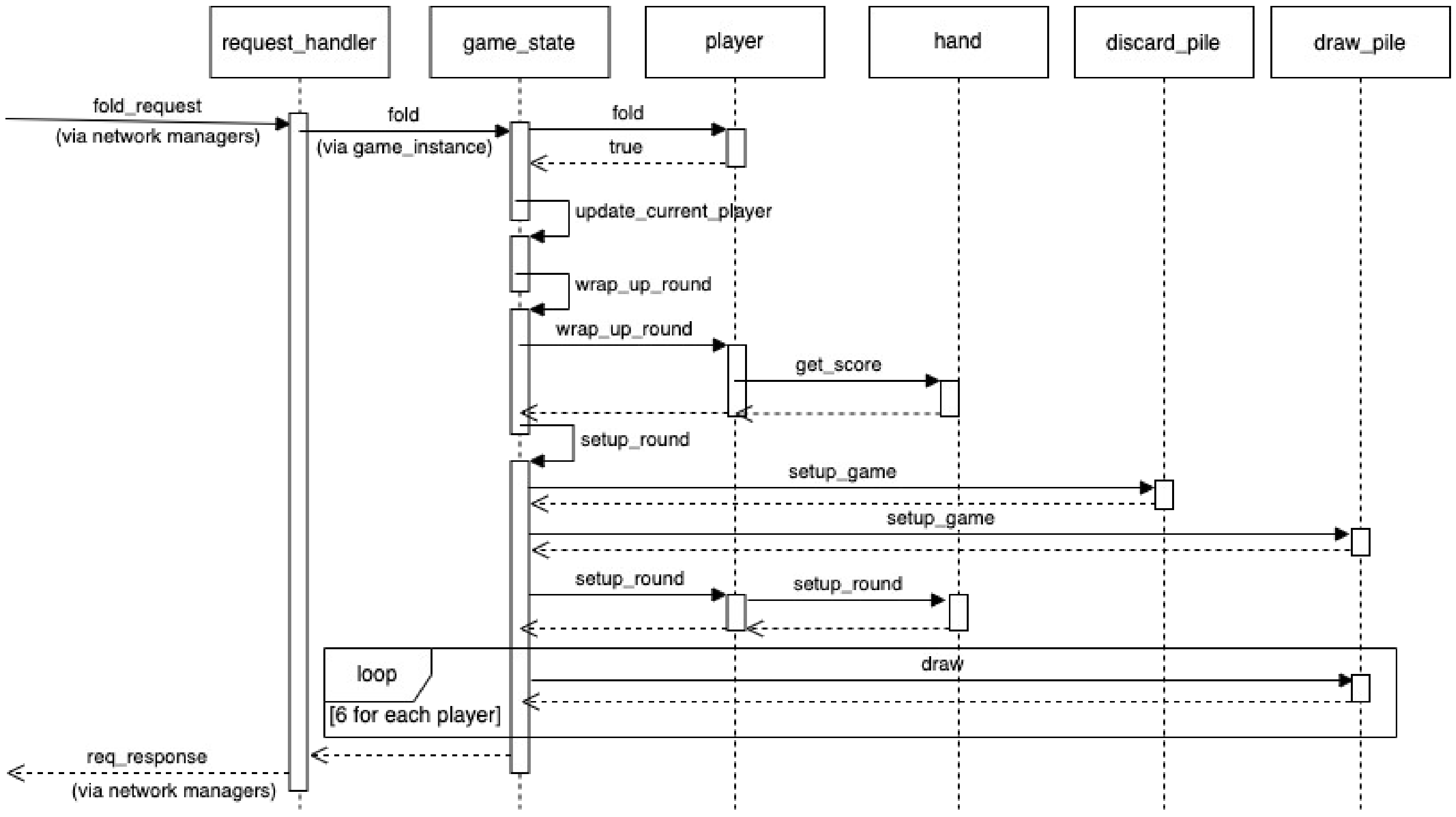

Figure 5. Sequence diagram of sequence End of Round.

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

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
- req_id: string (required)
- player_name: string (required)

Format: as JSON string

Example:
{"type":"join_game","player_id":"48e7ed2e-dc3d-49a5-ab11-8374ca9aaed6",
"game_id":"","req_id":"2d87932b-e8b1-4cdf-bc90-4a53fe34bfdf","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)
- req_id: string (required)

-

Format: as JSON string

Example:
{"type":"start_game","player_id":"aa91a118-fb3d-4f1f-bd65-f17ad38e1326","game_id":"08a52fb2-a289-41cd-b2e7-513cc2ce0db9","req_id":"c3d4eb03-4ceb-497f-8b39-9c260c5781e2"}

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)
- req_id: string (required)
- card_id: string (required)

Format: as JSON string

Example:
{"type":"play_card","player_id":"aa91a118-fb3d-4f1f-bd65-f17ad38e1326","game_id":"08a52fb2-a289-41cd-b2e7-513cc2ce0db9","req_id":"38461ed1-9a24-4bb3-93da-3f95e134ea38","card_id":"90ab1dc6-f563-40ba-abe5-15dd83422451"}

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)
- req_id: string (required)
- nof_cards: int (required)

Format: as JSON string

Example:

{"type":"draw_card","player_id":"aa91a118-fb3d-4f1f-bd65-f17ad38e1326","game_id":"08a52fb2-a289-41cd-b2e7-513cc2ce0db9","req_id":"7da10e0b-f6f1-4a80-bf5d-77aabcc52490","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)
- req_id: string (required)

Format: as JSON string

Example:
{"type":"fold","player_id":"efd43480-d142-4c80-873a-12448e070990","game_id":"08a52fb2-a289-41cd-b2e7-513cc2ce0db9","req_id":"fed9f657-678f-4de1-80f3-826dbbd7df52"}

Expected response: req_response

req_response

Purpose: Answer a request from a Client

Direction: Server to Client

Content:

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

Format: as JSON string

Example:
{"type":"req_response","game_id":"08a52fb2-a289-41cd-b2e7-513cc2ce0db9","err":"","req_id":"7b977bdf-c79a-49bb-a6f8-e3426c773e43","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-a289-41cd-b2e7-513cc2ce0db9","state_json" : <json state encoding>}

Expected response: none