ETH zürich

FS 2021

barefoot

Software Design Specifications

for

digital UNO

Version 0.4 approved

Prepared by Achart Marie-Louise; Mazzoleni Alice; Heer Damian; Schaufelberger Andreas; Alija Valon; Gutmann Katharina

Cereal Killers

01.04.2021

Table of Contents

Revision History

Name	Date	Release Description	Version
Felix Friedrich	3/21/21	Template for Software Engineering Course in ETHZ.	0.2
Cereal Killers	3/21/21	First draft to be reviewed	0.3
Cereal Killers	4/1/21	Final version	0.4

1. Introduction

1.1 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.>

1.2 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.>

1.3 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.>

1.4 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.>

2. Static Modeling

2.1 Package Server

The package server contains code relevant for the server.

2.1.1

2.1.2 Discard_Pile

The class attributes are:

ATTRIBUTE: std::stack<CARD> cards (store all cards of discard pile in vector)

The class operations are:

OPERATION: CARD get_top_card(), get top card of pile important if discard pile needs to be reshuffled
OPERATION: string to_json(), serialize discard_pile
OPERATION: void from_json(string), deserialize cards held in Discard_Pile
OPERATION: void add_card(CARD), adds played card to discard pile
OPERATION: void set_cards(std::vector<CARD>), set pile of card, important if one wants to reshuffle
OPERATION: bool played_card_valid(CARD), check if card can be played
OPERATION: void add_hand(Hand), add the hand at the bottom if player wants to leave

2.1.3 Draw_Pile

The class attributes are:

ATTRIBUTE: std::stack<CARD> cards (store all cards of draw pile in vector)

The class operations are:

OPERATION: string to_json(), serialize discard_pile
OPERATION: void from_json(string), deserialize value of card
OPERATION: void push_card(CARD), adds played card to Draw_Pile
OPERATION: void pop_card(CARD), removes card of Draw_Pile and return it
OPERATION: void reshuffle(), put elements of cards in random order
OPERATION: void set_cards(std::vector<CARD>), set pile of card, important if one wants to reshuffle

2.1.4 Game_State

The class attributes are:

ATTRIBUTE: Draw_Pile draw_pile
ATTRIBUTE: Discard_Pile discard_pile
ATTRIBUTE: std::vector<Player> players (array of all players currently playing)
ATTRIBUTE: int current_player (array index of which players turn it is)
ATTRIBUTE: COLOR color_to_be_matched (specifies the color to be matched, important if one plays a wild card)

The class operations are:

OPERATION: void set_current_player(int) (set array index of player which is currently playing)
OPERATION: string to_json(), serialize Game State
OPERATION: void from_json(string), deserialize Game State

2.1.5 Game_Controller

The class attributes are:
ATTRIBUTE: Game_State game_state

The class operations are:
OPERATION: Response execute_request(GameState, Request) handles the request, update the GameState and returns an answer

2.1.6 Response

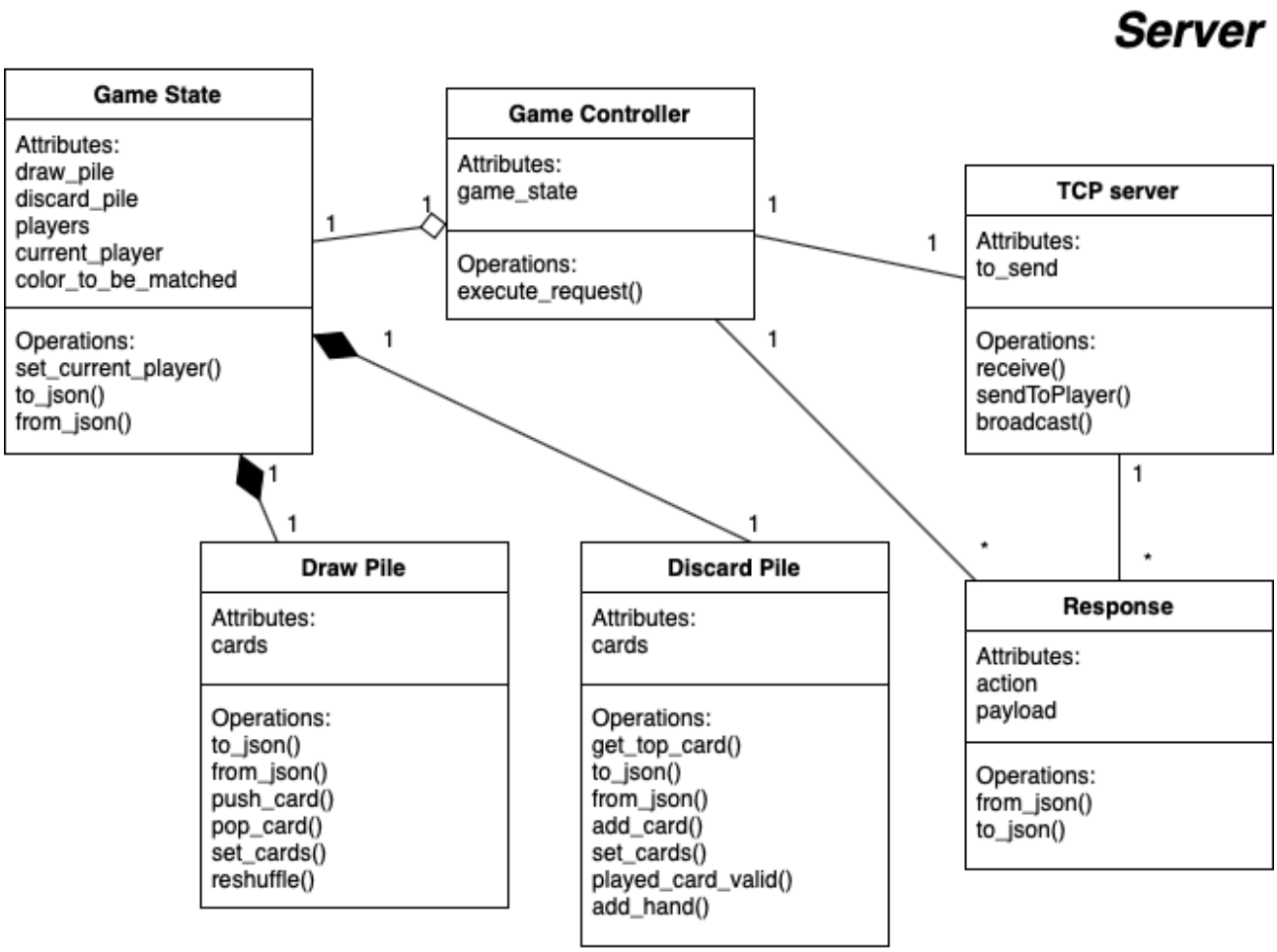
The class attributes are:
ATTRIBUTE: RESPONSE action
ATTRIBUTE: void* payload implementation specified payload to handle requests

The class operations are:
OPERATION: void from_json(string), deserialize request
OPERATION: string to_json(), serialize request

2.1.7 TCP_Server

The class attributes are:
ATTRIBUTE: queue<string> to_send;

The class operations are:
OPERATION: static void receive(PLAYER, callback), adds a lambda callback for receiving messages
OPERATION: static void sendToPlayer(PLAYER, string msg) tries to send the string to the player
OPERATION: static void broadcast(string msg) tries to send the messages to all connected players



2.2 Package Client

The package client contains code relevant for the client.

2.2.1 Player

The class attributes are:

ATTRIBUTE: int unique_player_id (player_id for unique identification)
ATTRIBUTE: Player_State player_state
ATTRIBUTE: bool is_active (if player exited is_active is set to false)

The class operators are:
OPERATOR: void set_is_active(bool) (server sets player to inactive if he exits)

2.2.2 Player_State

The class attributes are:
ATTRIBUTE: Hand hand (cards player has in hand)

ATTRIBUTE: unsigned int number_of_cards (should be displayed for all players)
ATTRIBUTE: bool players_turn (true if it is the players turn, one can only do actions if true)
ATTRIBUTE: bool has_won; (true if player has won, one cannot do any actions if true)

The class operations are:

OPERATION: void set_players_turn(bool) (server will set players turn to true if it is its turn)
OPERATION: void set_player_won(bool) (server will set has_won to true if player has won)
OPERATION: void set_number_of_cards(int) (update number of cards)

2.2.3 Hand

The class attributes are:

ATTRIBUTE: std::vector<CARD> cards (store all played cards in vector)

The class operations are:

OPERATION: void add_card(CARD) (server will update hand when cards will be drawn)
OPERATION: void remove_card(CARD) (server will remove played cards)
OPERATION: void from_json(string), deserialize Hand
OPERATION: string to_json(), serialize value of Hand

2.2.4 Player_Controller

The class attributes are:

ATTRIBUTE: Game_State game_state
ATTRIBUTE: wxFrame graphic_user_interface (setup frame)

The class operations are:

OPERATION: void draw_card(int unique_player_id) (if players turn is true it initializes a Request of type draw and sends it to the server)
OPERATION: void play_card(CARD, int unique_player_id) (if players turn is true it initializes a Request of type play and sends it to the server)
OPERATION: void exit(int unique_player_id) (initialize a Request of type exit and sends it to the server)
OPERATION: void start_game() (initializes a Request of type start_game and sends it to the server)
OPERATION: void handle_response(response) (update the play_state)
OPERATION: void select_color() (send color to be matched if one has played a wild card)

2.2.5 Request

The class attributes are:

ATTRIBUTE: int unique_player_id
ATTRIBUTE: REQUEST action
ATTRIBUTE: void* payload implementation specified payload to handle requests

The class operations are:

OPERATION: void from_json(string), deserialize request

OPERATION: string to_json(), serialize request

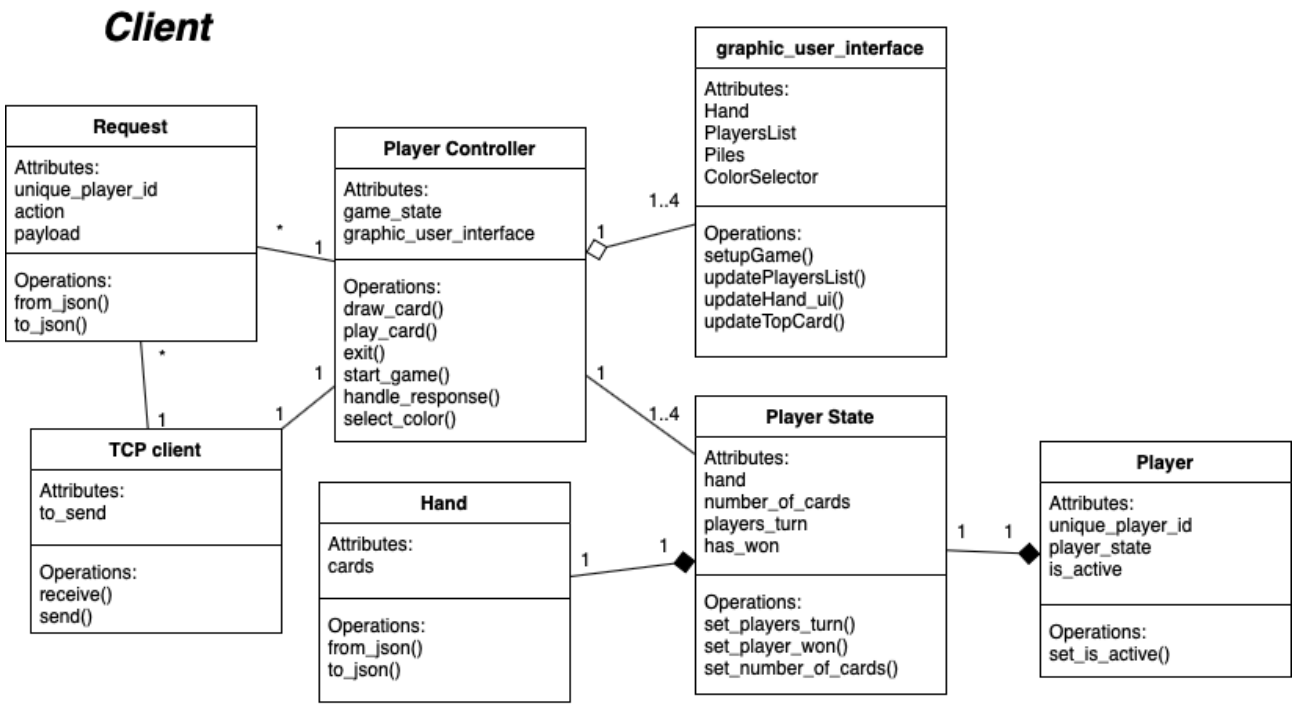
2.2.6 TCP_CLIENT

The class attributes are:

ATTRIBUTE: queue<string> to_send;

The class operations are:

OPERATION: static void receive(callback), adds a lambda callback for receiving messages
OPERATION: static void send(string msg) tries to send the string to the server



2.3 Package Common

The package common contains code relevant for both client and sever.

2.3.1 Card

The class attributes are:

ATTRIBUTE: color, specifies color of card (green, blue, yellow, red, none for wild cards)
ATTRIBUTE: value, specifies number of card (0-9, none for action cards)
ATTRIBUTE: action, specifies type of action card (wild, wild_draw_4, draw_2, reverse, skip, none for nonaction cards)

The class operations are:

OPERATION: int get_value() getter

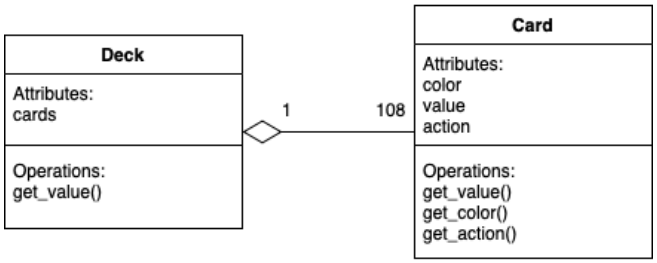
OPERATION: COLOR get_color(), getter
OPERATION: ACTION get_action(), getter

2.3.2 Deck

The class attributes are:
ATTRIBUTE: std::unordered_map<CARD, unique_ptr<Card>> cards, store all 108 cards of a deck into a map for easy referencing of cards cards

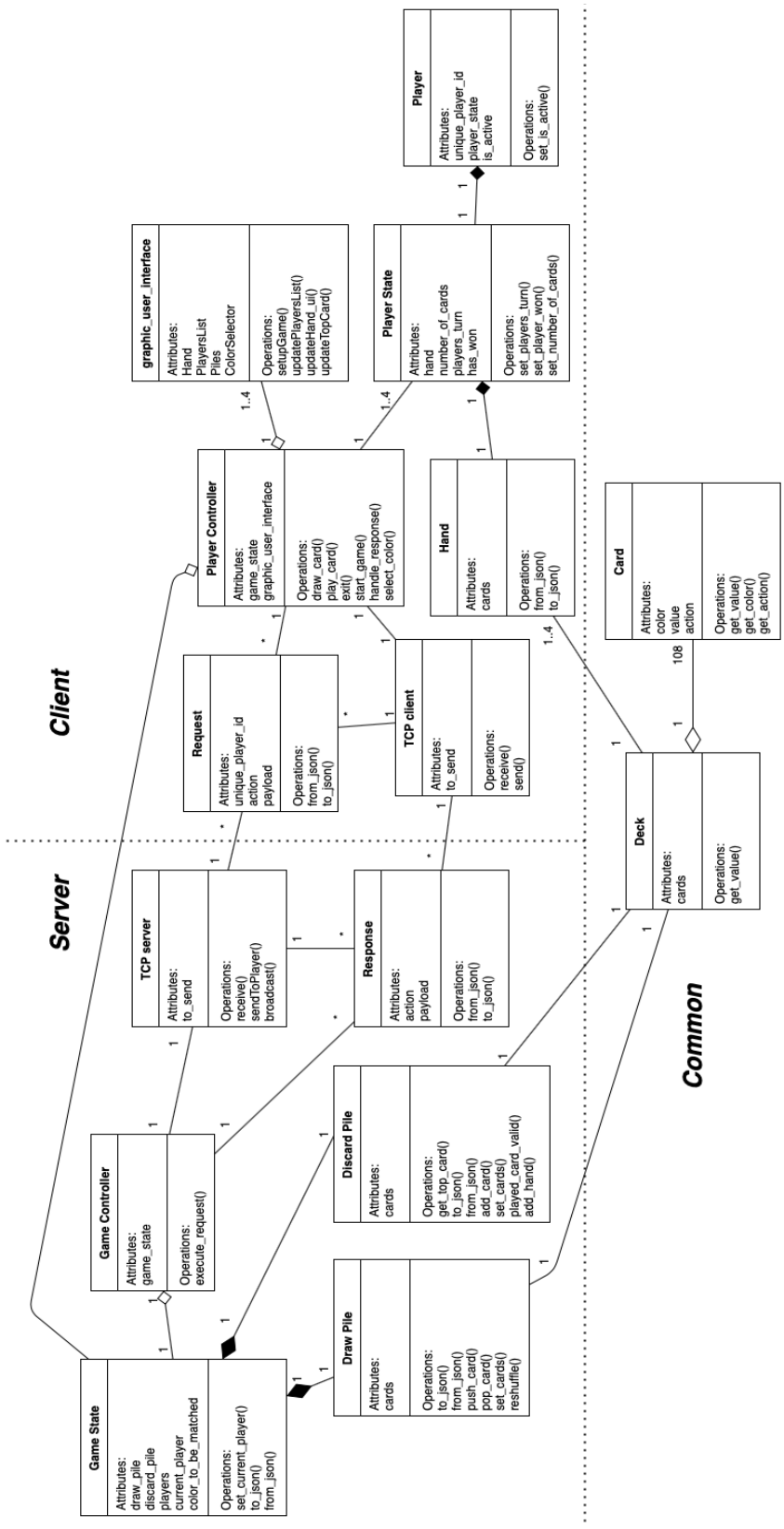
The class operations are:
OPERATION: const Card& get_value(CARD)

Common



2.4 Composite Structure Diagram

2.4 Composite Structure Diagram



3. Sequence Diagrams

3.1 Sequence Deck Reshuffle

The functional requirements related to this sequence are:

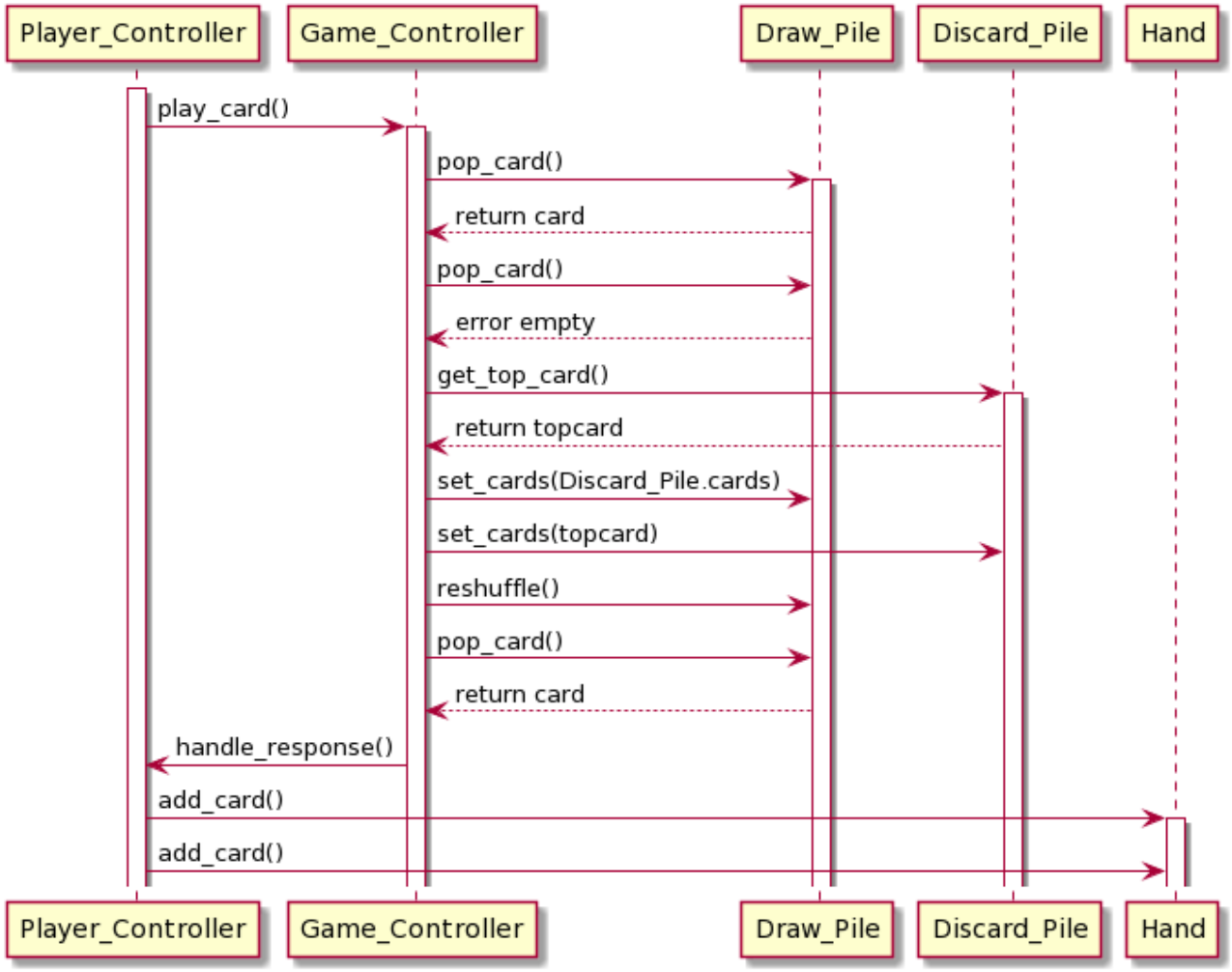
FREQ-3: Graphic User Interface
FREQ-7: Ruleset of Uno
FREQ-10: Play/Draw
FREQ-15: Highlight of current player
FREQ-16: Draw cards
FREQ-17: Timeout

The scenarios which are related to this sequence are:

SCN-1: Deck Reshuffle

Scenario Narration:

The user will draw 2 cards. After drawing the first card the deck is empty. Reshuffling the cards of the discard pile and then drawing another card is required.


Player_Controller
Game_Controller
Draw_Pile
Discard_Pile
Hand
play_card()
pop_card()
return card
pop_card()
error empty
get_top_card()
return topcard
set_cards(Discard_Pile.cards)
set_cards(topcard)
reshuffle()
pop_card()
return card
handle_response()
add_card()
add_card()
Player_Controller
Game_Controller
Draw_Pile
Discard_Pile
Hand

3.2 Sequence Diagram One Player exits but more than two players are still in the game

The functional requirements related to this sequence are:

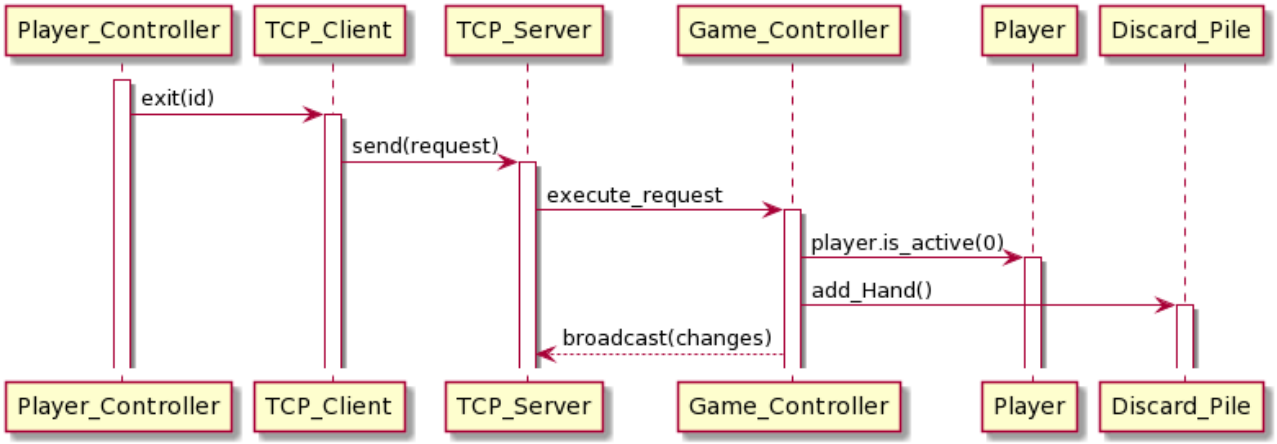
FREQ-5: Displaying user-id
FREQ-9: Exit game

The scenarios which are related to this sequence are:

SCN-3: Only Player exists but more than one player are still in the game

Scenario Narration:

A player exits. The other players will continue the game.



3.3 Sequence Play Wild Card

The functional requirements related to this sequence are:

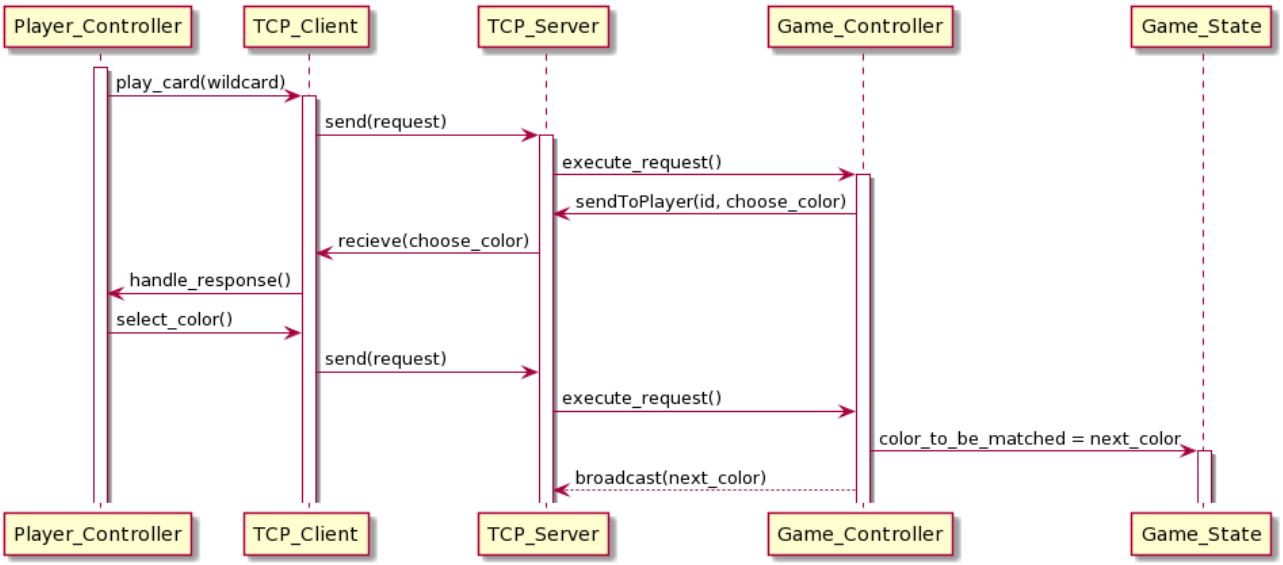
FREQ-7: Ruleset of Uno
FREQ-18: Colour selection

The scenarios which are related to this sequence are:

SCN-5: Player plays Wild Card

Scenario Narration:

A Player plays a Wild card to choose the color. Window pops up where he can choose the color of the card, which will then be applied.



4. Interface Modeling

Messages passed between the client and the server are c++ objects (of class request), with the underlying TCP network protocol, allowing serialization and deserialization to/from JSON file string from/to c++ objects. Details to be found here: https://lec.inf.ethz.ch/se/2021/project/phase2.html

4.1 Main Interface

Purpose: Starts and ends the game, allows the transfer of request and response between players and server.

Communication between: TCP_Server <-> TCP_Client

Protocol: TCP

Communication modes:

- game_intitalization{join_game, setup_game, start_game}
- game_playing{draw_request, play_request, exit_request, game_update_respond, change_players_turn_respond, select_color_request, select_color_respond}

4.1.1 play_request

Purpose: Send key of card to be played

Direction: Client to Server

Content:

type: string (required)
unique_player_id: integer (required)
card: CARD

Format: as JSON string

Example: `{"type": "play_request" "unique_player_id": 2, "card": yellow_8 }`

Expected response: error_message if invalid card was played, server updates game_state and broadcasts message game_update_respond, server sends message change_players_turn response to next player

4.1.2 game_update_respond

Purpose: send some important information of game_state such that user interface can be updated

Direction: Server to Client

Content:

type: string (required)
players: std::vector<Player> (required)
current_player: int (highlight of current player)
color_to_be_matched: COLOR
card: CARD (top card of discard pile)

Format: as JSON string

Example: `{"type": game_update, "players": std::vector<Player>, "current_player": 2, "color_to_be_matched": yellow, "card": yellow_0}`

Expected response: every player updates the game state according to the information in the game_state_response

4.1.3 select_color_request

Purpose: Server asks player which color should be played next (after wildcard).

Direction: Server to Client

Content:

type: string (required)
unique_player_id: integer (required)

Format: as JSON string

Example: `{"type": select_color_request, "unique_player_id": 2 }`

Expected response: clicks onto desired color sends a selected_color_response