HS 2024

Software Design Specifications

for

Durak

Version 0.1 approved

Prepared by Eric Holenstein, Noah Rutz, Thomas Waldmann, Vito Maiocchi and Danil Poluyanov

Tubbeli Trupp

14.10.2024

Table of Contents

Revision History

Name	**Date**	**Release Description**	**Version**
Felix Friedrich	6.11.24	Template for Software Engineering Course in ETHZ.	0.2
Tubbeli Trupp	6.11.24	Durak SDS	1.0

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 Common

This package contains the classes and methods used by both the server and the client. It is mainly responsible for Networking aspects.

Class Card

Util class which describes the 52 cards of the game with two enums.

The class attributes are:

- enum suit
- enum rank

The class operations are:

toInt()
fromInt()

Class Networking Handler

Handles all network connections. For each client there are two connections for each direction of communication. On the Server a separate thread listens to each inbound connection and updates a message queue. With receiveMessage() one can fetch the next Message. Sending will happen directly. On the Client it is the other way around receive is not an extra thread and sending has a queue. This is because the client Events are triggered by clicks and are not synchronous.

The class attributes are:

- List of connections
- Open sockets
- Active threads

The class operations are:

- openSocket(port)
- openConnection(ip, port)
- sendMessage(Message)
- Message receiveMessage()

Base Class Message

Base class for all Messages that can be easily serialized for network communication.

The class attributes are:

- Message type

The class operations are:

- toJSON()
- fromJSON()

All next classes in this package a derived classes from the base class Message, they all have the same member functions and attributes.

Class Illegal Move Notify

Notifies if the move is illegal.

The class attributes are:

- Error message

The class operations are:

Inherited from base class

Class Card Update

Contains information about all the cards that a client should know about.

The class attributes are:

- Amount of cards per opponent (can be 0)
- Amount of cards on draw pile
- Trump card (can be NULL)
- Trump suit
- Cards in the middle
- Hand

The class operations are:

Inherited from base class

Class Player Update

Contains the information of the players. (Each player has an integer ID)

The class attributes are:

- Name of each player
- Amount of players
- Durak of last game: uint

The class operations are:

Inherited from base class

Class Battle State Update

Contains the information about the Battle State (player ID and their role)

The class attributes are:

- Defenders
- Attackers
- Idle

The class operations are:

Inherited from base class

Class Available Action Update

Contains what Actions are available to you.

The class attributes are:

- Pass On

- Okay
- Pick Up

The class operations are:

Inherited from base class

Class Game State Update

Contains the meta information about the game.

The class attributes are:

- Enum current Meta state (lobby, game, spectator, game over, Durak screen)

The class operations are:

Inherited from base class

Class Play card Event

Contains the information about the card and the place of a card that some player wants to play.

The class attributes are:

- Card
- Enum slot

The class operations are:

Inherited from base class

Class Client Action Event

Contains information about which Action a player wants to perform.

The class attributes are:
Action (Enum: Okay, Pass on, Pick up)
The class operations are:

Inherited from base class

Class Join Event

Contains information about the players.

The class attributes are:

- username

The class operations are:

Inherited from base class

Class Disconnect Event

Dummy Message only for the Network Manager. Useful to profit from the Message queue in the Network Class.

The class attributes are:
- none

The class operations are:

none

Class Diagram of Package Common

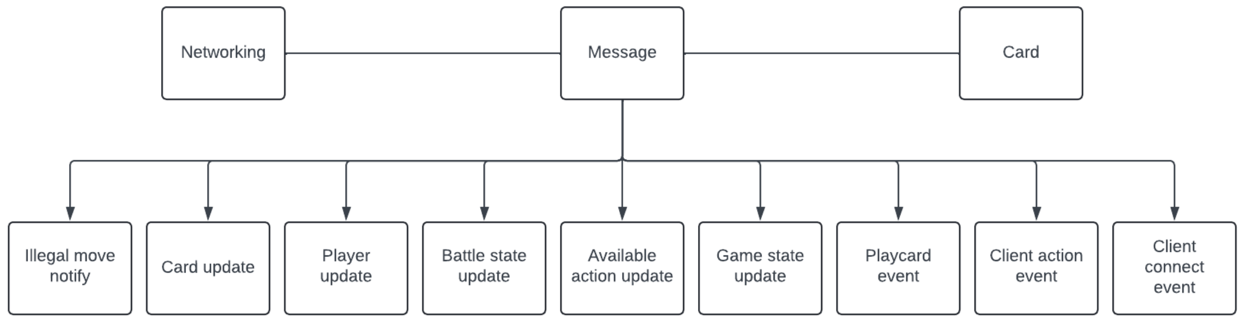


Figure 1: Class diagram of package common

2.2 Package Client

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

Class Node

Abstract Base Class for all nodes

The class attributes are:
- MinWidth, minimum width of the node
- MinHeight, minimum height of the node

The class operations are:
- Draw, draws this node to the screen
- SendClickEvent, where (x,y) is clicked
- SetClickEventCallback, sets a callback function which is executed if clicked
- CallForAllchildren calls a function for all children

Class Tree Node

Base Class for all nodes with children. Helps with click callback

The class attributes are:
- none

The class operations are:

- Inherited from Node

Class Leaf Node

Base Class for all nodes without children

The class attributes are:
- none

The class operations are:
- Implements callforallchildren as nothing

Class Image Node

Displays an image

The class attributes are:
- Image, width, height

The class operations are:
- Inherited from Leaf Node
- Draw draws image

Class Text Node

Displays text

The class attributes are:
- Text format
- Text content

The class operations are:
- Inherited from Leaf Node
- Draw text

Class Box Node

Displays colored box

The class attributes are:
- color

The class operations are:
- Inherited from Leaf Node
- Draw draws box

Class Buffer Node

Creates a buffer around the node

The class attributes are:
- MinWidth, minimum width of the Buffer Node

- MinHeight, minimum height of the Buffer Node
- BufferType, enum for different buffertypes
- Buffersize, the size of the buffer

The class operations are:

- Inherited from Tree Node
- SetBufferSize, sets the Buffer with buffertype and buffer size

Class Master Node

Node at the top of the node tree. Inherited from tree node. Represents the screen
This is where everything that is displayed is created.

The class attributes are:

- various elements displayed on screen (child nodes)

The class operations are:

- inherited from Tree Node
- draw: draw the screen
- handle Message: update the components according to server specifications

Class Linear stack Node

Stacks the nodes linear to each other with a direction and type

The class attributes are:

- minWidth, minimum width of the Linear Stack Node
- minHeight, minimum height of the Linear Stack Node
- stackDirection, enum for the direction of the stack (horizontal, vertical)
- stacktype, enom for the stacktype (compact, spaced)

The class operations are:

- inherited from Tree Node
- setStackType, sets the stacktype, with a stack direction and type

Class Card stack Node

Leaf node that displays two stacked cards in a way where both are visible.

The class attributes are:

- minWidth, minimum width of the Card Stack Node
- minHeight, minimum height of the Card Stack Node
- The two cards in question

The class operations are:

- Inherited from Leaf Node
- Draw draws the two cards

OpenGL

Class Image

Renders images to screen with OpenGL

The class attributes are:

- Image (texture)

The class operations are:

- Draw (draws the image at some size and location)

Class Text

Draws text to screen with OpenGL

The class attributes are:

- Format
- Text

The class operations are:

- Draw: draws the text

Class Box

Draws a colored box with OpenGL

The class attributes are:

- none

The class operations are:

- Draw box (of color and size location)

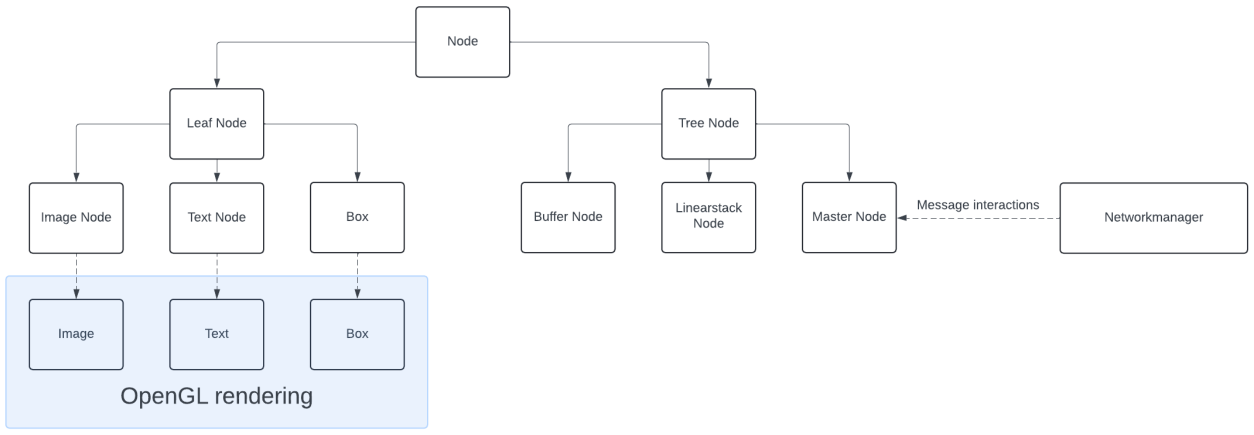


Figure 2: Class diagram of package client

2.3 Package Server

The package server contains all necessary classes and methods to run the server successfully. It controls the main logic of the game.

Class Server

Contains a main loop and is responsible for receiving and handling all network communication. Everything that is not part of the actual game logic is handled here. Like a lobby and keeping track of connected players.

The class attributes are:
- players: unordered_map<uint (playerid), <undeclared type>>, keeps track of players connected to server and their ids
- Game current_game

The class operations are:
- startServer(): initiates the server socket and starts listening for incoming messages.
- mainLoop(): keeps the server running and receives messages from network handler.
- handleIncomingMessage(): receives a message and checks its contents.
- broadcastMessage(): sends a message to all clients.
- onPlayerExit(): handles the case when a player exits the game.
- createGame(): creates a game once enough players joined and are ready
- tryAddSpectator(): bool, tries to add a player as a spectator
- removePlayer(): bool, removes a player from the player list.
- tryGetPlayer(): bool, retrieves a player from the player list.
- addPlayer(): bool, adds a new player to the player list
- handleDisconnection(): if player disconnects.

Class Game

Starts the first battle and a new battle after a battle has ended. Receives messages from the server and passes them on accordingly to the Battle Class to handle. Also handles general information required between battle instances

The class attributes are:
- players: std::list<player_id>, stores a list of players, which will be used to keep track of the turns (attackers, defenders, spectators)
- active_players: std::tuple<player_id, player_id, player_id>, stores a triple of players which designate #1 the attacker, #2 the defender and #3 the co-attacker.
- current_battle: Battle&, stores a reference to the currently active battle instance
- card_manager: CardManager, stores the card manager object.

The class operations are:

- createGame(): bool, initializes the game by shuffling the deck, dealing cards, determining the trump card and thus suit, setting the first attacker and calling makeFirstBattle()
- makeFirstBattle(): bool, creates the first battle and passes the special rules (int max_attacks, bool pass_on_legal)
- createBattle(): bool, attempts to create new battle, passing the current attackers and defender
- isStarted(): bool, returns if there is an ongoing battle i.e. if there is an instance of current_battle
- endGame(): bool, when the last battle is over the current game instance is deconstructed and passes the information of who is the loser/'Durak'. Provides the client with the possibility to start a new game

- resetGame(): &game_t, resets the current game to a fresh one, if the server demands it
- updateTurnOrder(): bool, updates active_players before every new battle
- handleClientActionEvent(): bool, handles client action events appropriately
- handleClientCardEvent(): bool, handles player card action.

Class Battle

The Battle class manages the card interactions between players during a round. This includes handling turns, playing cards, and determining the outcome of each battle.
Sends out updates on the current battle state to the players.

The class attributes are:

- battle_state: std::vector< std::tuple<id, int> >, every player has their corresponding state (attacking, defending, spectating)
- defended_flag: bool
- battle_field: std::vector< std::tuple<card, card> > #max_attacks x 2 matrix designating the 'Battlefield' i.e. the six slots where the attacking and defending cards are stored for a given game
- card_manager: CardManager ptr, stores a pointer to the current card manager object
- max_attacks: int, designating the number of maximal attacking cards. Generally, this will be six but for the first move it is only five.
- attacks_to_defend: int, shows how many attacks there are left to defend, if 0 then all attacks are defended.

The class operations are:

- Battle(first_battle, CardManager* card_manager): constructor for Battle
- handleActionEvent(): bool, handles client action events appropriately
- handleCardEvent(): bool, handles player card action.
- attack(): helper function for the handle card event, only players in the correct battle state can use this
- defend(): helper function for the handle card event, only players in the correct battle state can use this
- successfulDefend(): bool, determines if an attack was successfully defended or not, setting the defended_flag to true for every single attack, flag is reset after the battle is done
- passOn(): checks if it is possible to pass on the cards to the next player, if yes, then it updates the battle state.
- isValidMove(): Returns whether a move requested by a player is valid or not.

Class CardManager

Keeps Track of all Cards in the game. Send out Card updates to all players when they change.

The class attributes are:

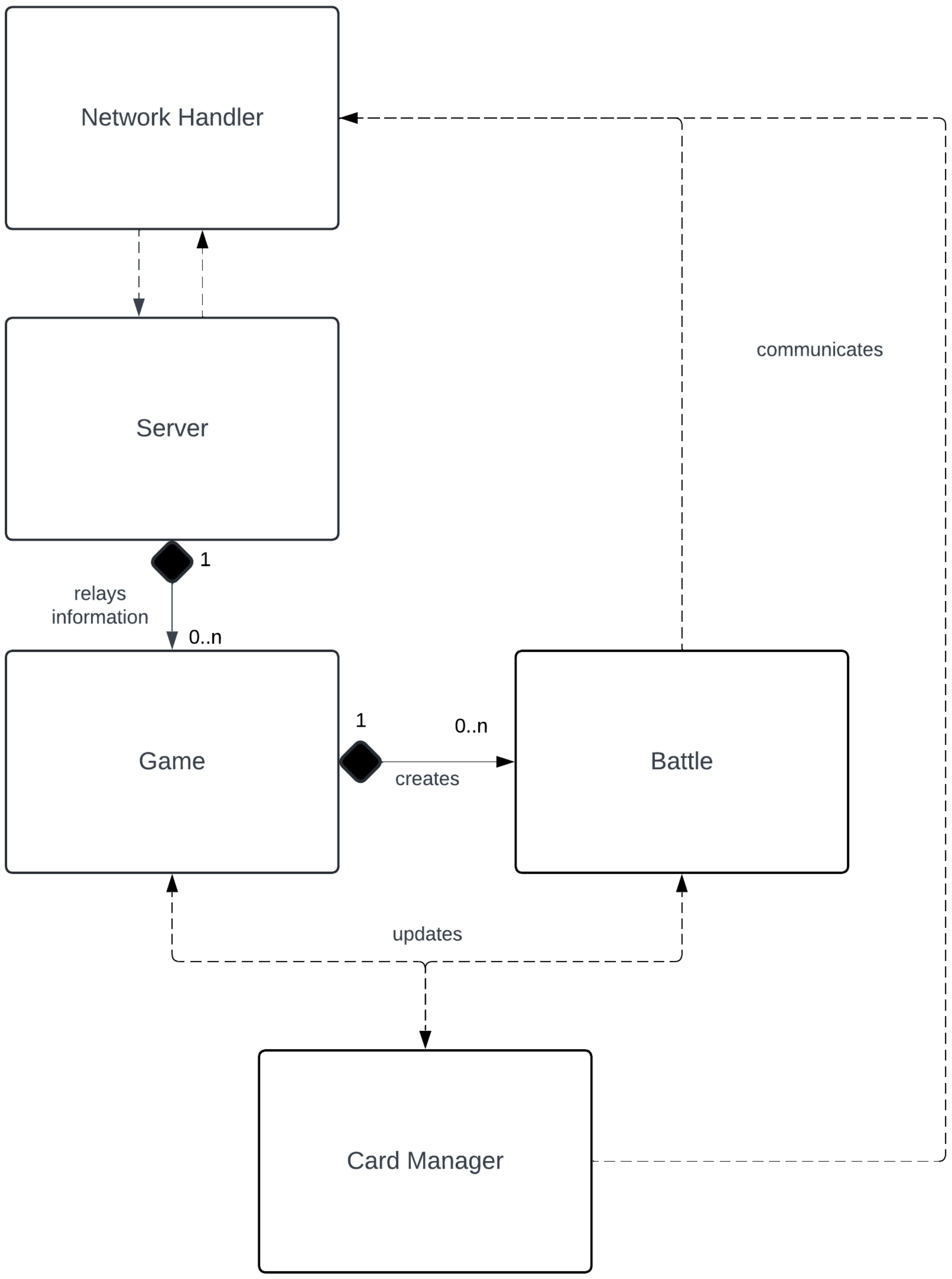
- deck: std::deque<Card>, Represents the deck with cards being dealt from one end of the queue
- last_card: *Card, Pointer to the bottom card of the deck. Once this card has been distributed this pointer is set to NULL
- number_cards_in_deck: unsigned int Number of cards in the deck, can be zero
- endgame: bool, true if last_card=NULL
- discarded_cards: std::vector<Card>, stores all cards that have been discarded
- number_discarded_cards: unsigned int, number of cards that are discarded
- trump: enum, trump suit
- player_hands: for each player stores the cards they are currently holding
- player_number_of_cards: Stores how many cards each player is holding

The class operations are:

- compareCards(): Implements comparison of two cards. A card is either greater than, lesser than or not comparable to another card. This function is needed to determine whether a move is valid or not. This function also considers the current trump suit
- playCard(): If a move is deemed valid, this function executes that move and updates the hand of the player involved (and his card count). This function also updates the cards lying in the middle
- clearMiddle(): This function is called when all attacks in a battle are successfully defended. All cards from the middle are moved to the discard pile.
- pickUp(): This function is called if the defender cannot defend the attack & he presses the “pick up” button. It assigns all the cards in the middle to the defending players’ hand and updates card counts
- shuffleCards(): at the beginning of the game shuffle the cards and deal 6 cards to each player
- determineTrump(): After cards have been shuffled, the bottom card of the deck will determine the trump suit. This function sets the trump suit accordingly and makes sets the pointer last_card to point to the last card in the deck.
- distributeNewCards
- getPlayerHand(unsigned int PlayerID): returns a copy of the player hand corresponding to the ID.
- getMiddle(): returns a copy of the cards in the middle (battlefield)
- getNumberActivePlayers(): Checks how many players still have cards in their hands. This function is used to check if the game is supposed to end or if a new battle will start.

Class Diagram of Package Server

Figure 3: Class diagram package server



2.4 Composite Structure Diagram

Figure 4 gives an overview of the structure of the Durak app. The Client and Server packages both use parts of the Common package. The Client and Server apps communicate through a TCP connection, whereas the user interacts with the client app through the client app's GUI and hardware input devices.

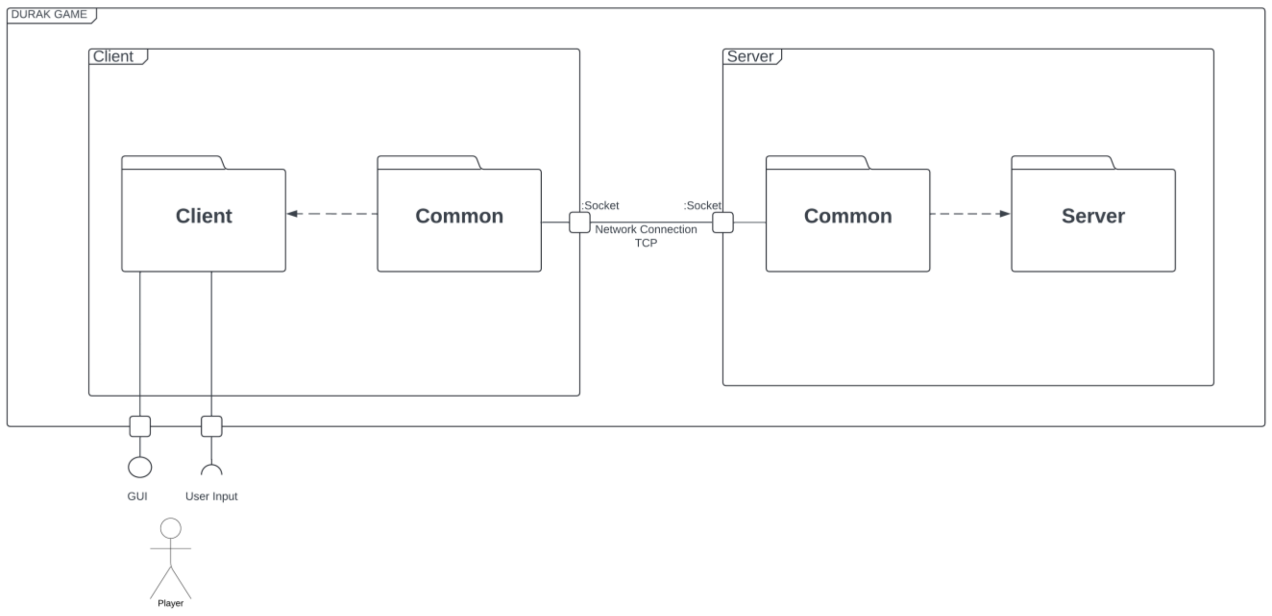


Figure 4. Composite Structure Diagram

3. Sequence Diagrams

3.1 Sequence Setting Up Game

This sequence is about starting the server, all players starting the clients, connecting to the server and joining the lobby.

The functional requirements related to this sequence are:

- FREQ-1: Game Server
- FREQ-2: Connection
- FREQ-3: GUI
- FREQ-4: Lobby
- FREQ-13: User Messages

The scenarios which are related to this sequence are:

- SCN-1: Setting up a game

Scenario Narration:

Player 1 starts the server application, and players 1, 2 and 3 start their client applications. All players fill in the required information, and press 'CONNECT', this connects each individual client to the server, which is hosted by player 1. The players now see the lobby, and everyone who is in it.

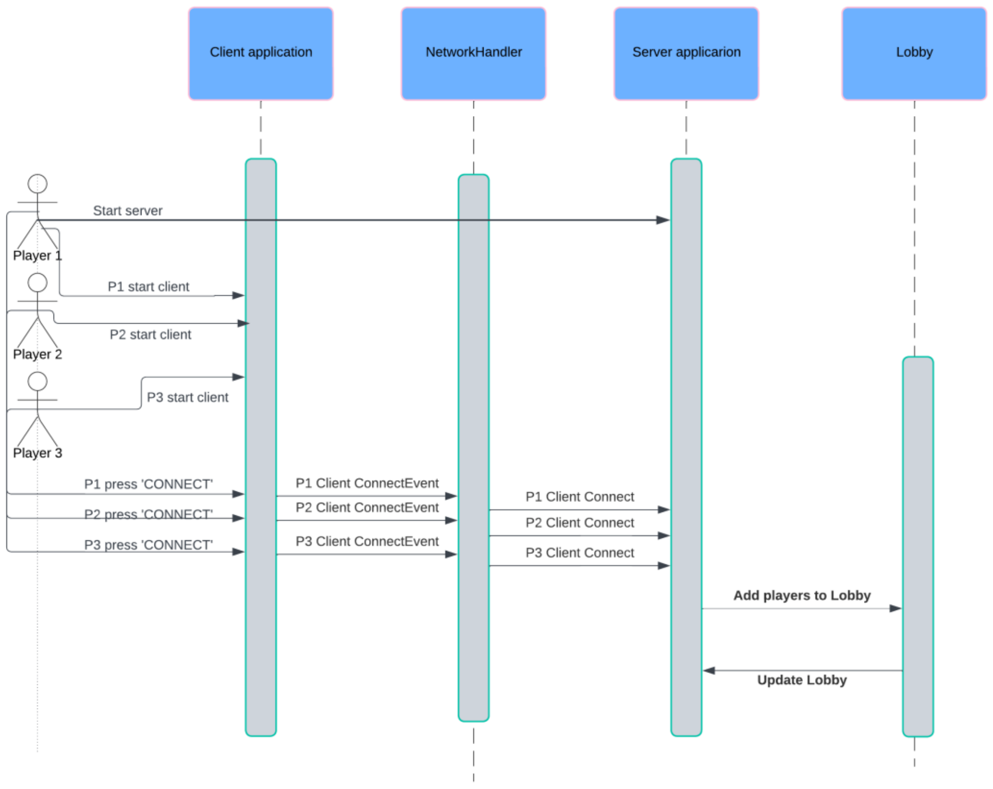


Figure 5: Sequence Diagram for setting up game

3.2 Sequence Start Game

This sequence is about starting a game, once five players have joined the lobby and pressed 'Ready'.

The functional requirements related to this sequence are:

- FREQ-1: Game Server
- FREQ-2: Connection
- FREQ-3: GUI
- FREQ-5: Game Start
- FREQ-6: First Move
- FREQ-13: User Message
- FREQ-14: Fairness

The scenarios which are related to this sequence are:

- SCN-2: Starting a game

Scenario Narration:

The lobby is filled with 5 players and the settings have been set. Everyone has pressed the 'Start Game' button and a new game is initialized. The server class passes the player ids and the game class shuffles a deck, determines the trump card (and trump suit), draws six cards for every player and determines the first attacker.

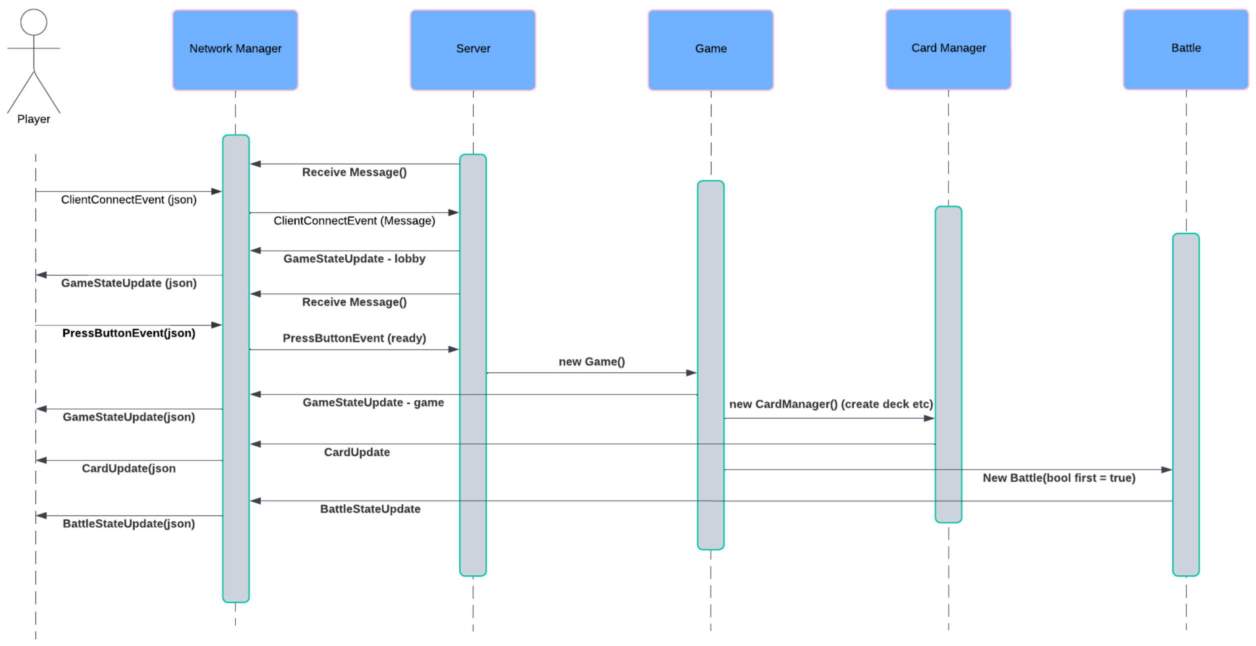


Figure 6: Sequence Diagram for starting a game

3.3

Sequence Battle

This sequence is about creating a new Battle, passing the battle on, playing a card, attacking, defending and ending the battle.

The functional requirements related to this sequence are:

- FREQ-1: Game Server
- FREQ-2: Connection
- FREQ-3: GUI
- FREQ-6: First Move
- FREQ-7: Turn-Based Gameplay with Time Limits
- FREQ-8: Attacking

- FREQ-9: Defending
- FREQ-10: Observing
- FREQ-11: After Turn End
- FREQ-13: User Message
- FREQ-14: Fairness

The scenarios which are related to this sequence are:

- SCN-3: Starting a battle by ending a battle
- SCN-4: Playing the wrong card

Scenario Narration:
In the current game ongoing, a new battle is created with the battle constructor. The attacking player clicks on the '4 of hearts' then on the battlefield. A message is sent from the client to the server, in server it is transferred to Game (this is all described in the sequence above, to focus better on the interactions between Game, Battle and CardManager we also left out the messages that are passed to the server and to the client). From Game the message is transferred to Battle, and there it checks for the game logic and calls the CardManager for the correct function to execute. Battle checks how many players still have cards, if only one then the battle is ended with the battle destructor. If all cards are defended or picked up, the battle is ended with the battle destructor. After a battle is ended, Game checks for the amount of players that still have cards, this needs to be checked in order to either start a new battle or end the game because only one player still has cards.

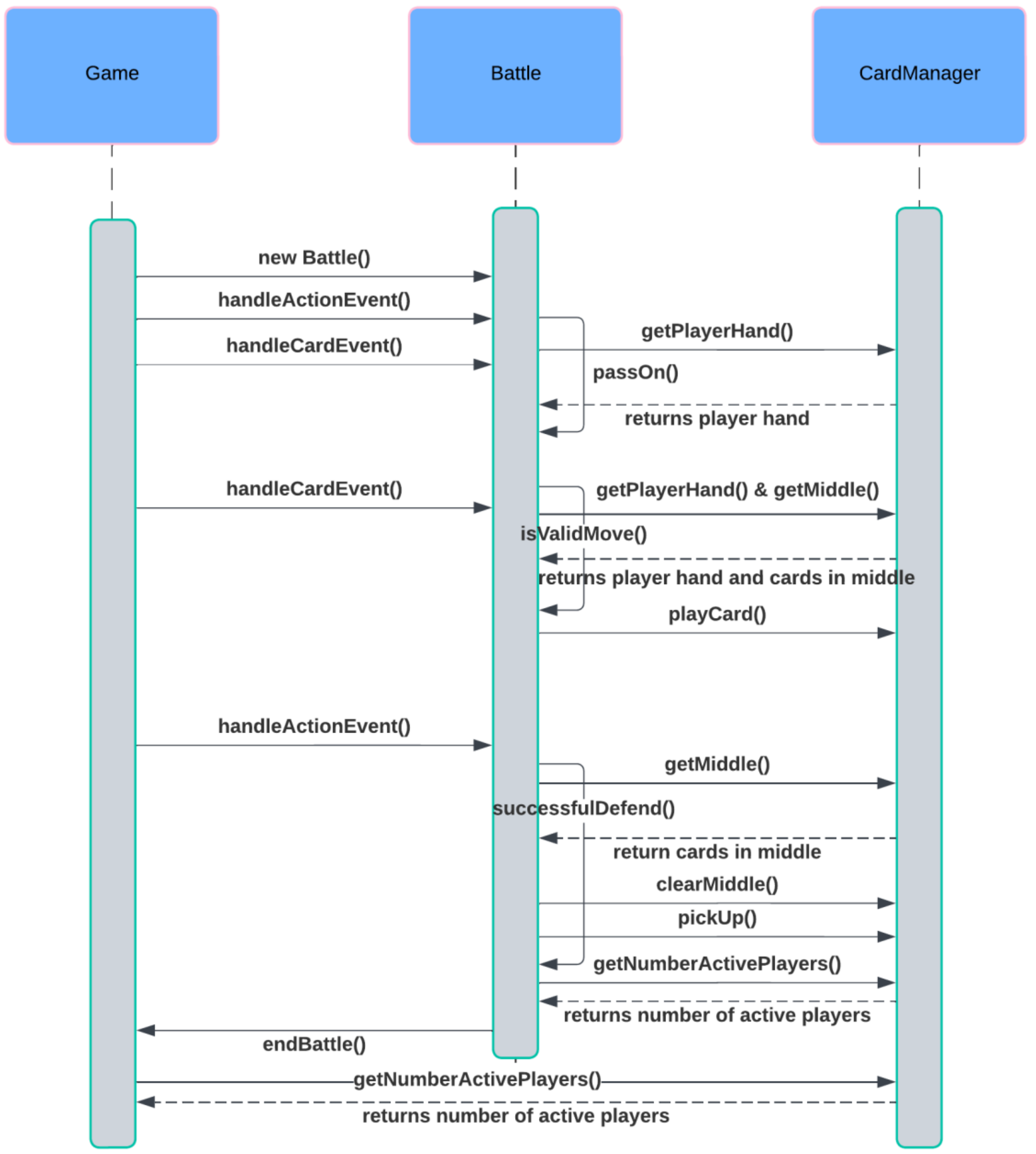


Figure 7: Sequence Diagram for a battle

3.4 Sequence Endgame

This Sequence is about the end of a game which includes the last attacks, ending the game and determining the Durak.

The functional requirements related to this sequence are:

- FREQ-1: Game Server
- FREQ-2: Connection
- FREQ-3: GUI
- FREQ-7: Turn-Based Gameplay with Time Limits
- FREQ-8: Attacking
- FREQ-9: Defending

- FREQ-11: After Turn End
- FREQ-12: Game End
- FREQ-13: User Messages
- FREQ-14: Fairness

The scenarios which are related to this sequence are:
SCN-5: Endgame

Scenario Narration: Klara attacks Frank with her last card (this part has been simplified below so that the sequence diagram doesn't look too clustered). After the turn has ended, the current battle calls the function numberActivePlayers in Card Manager to check how many players still have cards in their hands. If this number is less than two, the ongoing battle ends early. In this case the function returns that there is only one player (namely Frank) that still has cards. The destructor for battle is called and the current ongoing battle is ended. After the battle has ended, numberActivePlayers is again called from the current instance of "Game". Again the function returns that there is only 1 active player. As a consequence, the function endGame is called. endGame determines the Durak (loser) of the game and sends a message to client containing the player ID of the Durak.

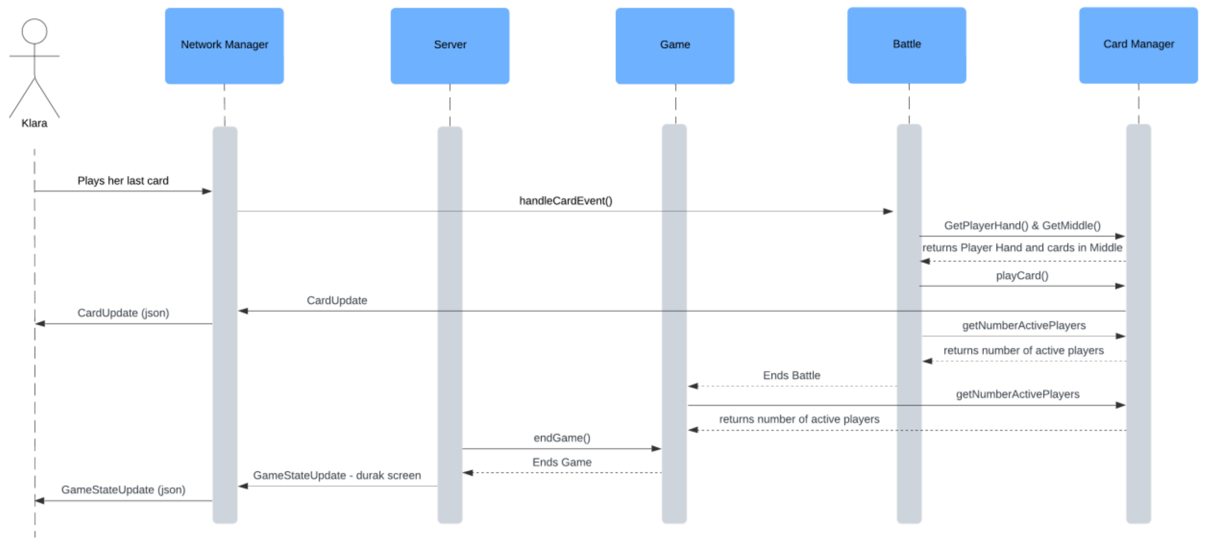


Figure 8: Sequence Diagram for endgame

4. Interface Modeling

4.1 Interface Client-Server

The purpose of this interface is the communication of the clients with the server. The interface allows transfer of Player actions from the Client to the Server. It is also used to transfer the Servers Game State to the Client to be displayed on Screen. As well as an invalid move Message. Communication happens in both directions independently.
Therefore, no responses are expected. Generally, communication happens as follows:
A player triggers an event by doing something. This Event is sent to the server.
The Server handles the event and sends updates to each Client to be Displayed.

Communication between: Client and Server, initiated by either.

Protocol: TCP

Communication modes: Client Actions, Server Updates, illegal Move Notify

4.1.1 MESSAGETYPE_ILLEGAL_MOVE_NOTIFY

Purpose: Notifies the client that a move is illegal
Direction: Server:Server to Client:MasterNode

Content:
- Enum MessageType (transmitted as uint)
- String Error Message

Format: as JSON string

Example:
```
{
      "messageType": 1,
      "content":
            {
                  "error" “invalid card”
            }
}
```

4.1.2 MESSAGETYPE_CARD_UPDATE

Purpose: Communicates the current status of the cards in play.
Direction: Server:CardManager to Client:MasterNode

Content:
- Map of ClientID to card count, map: ClientID(uint) -> uint
- Number of cards on draw pile, uint
- Trump card (can be NULL), Card(uint)
- Trump suit, Suit(uint)
- Cards in the middle, map: Slot(uint) -> uint
- Hand, list of cards(uint)

Format: as JSON string

Example:
```
{
  "opponentCards": { "1": 6, "2": 4 },
  "drawPileCards": 10,
  "trumpCard": 12,
  "trumpSuit": 2,
  "middleCards": { "1": 8, "2": 10 },
  "hand": [5, 13, 27]
}
```

4.1.3 MESSAGETYPE_PLAYER_UPDATE

Purpose: Provides an update on the players in the game.
Direction: Server:Server to Server:Game

Content:
- Name of each player, map: ClientID(uint) -> string
- Number of players, uint
- Durak, uint

Format: as JSON string

Example:
```
{
  "playerNames": { "1": "Alice", "2": "Bob", "3": "Charlie" },
  "numPlayers": 3,
  "durakID": 2
}
```

4.1.4 MESSAGETYPE_BATTLE_STATE_UPDATE

Transmits the information about the Meta Battle State.

Purpose: *Transmits update about the Meta Battle State*

Direction: Server:Server to Client:MasterNode

Content:
- Enum MessageType (transmitted as uint)
- ClientID Defender (transmitted as uint)
- List ClientIDs(uint), Attackers
- List ClientIDs(uint), Idle

Format: as JSON string

Example:
```
{
      "messageType": 4,
      "content":
            {
                  "defender": 2,
                  "attacker": [1, 3],
                  "idle": [4, 5]
            }
}
```

4.1.5 MESSAGETYPE_AVAILABLE_ACTION_UPDATE

Purpose: *Tells the client what actions are available in order to render appropriate buttons on the client*

Direction: Server:Battle to Client:MasterNode

Content:
- Enum MessageType (transmitted as uint)
- Bool passon
- Bool ok
- Bool pickup

Format: as JSON string

Example:
```
{
      "messageType": 6,
      "content":
            {
                  "pass_on": true,
                  “ok”: false,
                  “pickup”: true
            }
}
```

4.1.6 MESSAGETYPE_GAME_STATE_UPDATE

Transmits update about the Meta Game State. It is used by the client to know what screen to render (lobby, game, ...). It is sent by the Server every time this state changes.

Purpose: *Transmits update about the Meta Game State*

Direction: Server:Server to Client:MasterNode

Content:
- Enum MessageType (transmitted as uint)
- Enum current of Meta state
  (lobby, game, spectator, game over, Durak screen)
  (transmitted as uint)

Format: as JSON string

Example:
```
{
      "messageType": 6,
      "content":
            {
                  "state": 3
```

```
		}
}
```

4.1.7 MESSAGETYPE_PLAYCARD_EVENT

Contains the information about the card and the place of the card. Multiple cards can be played at once.

Purpose: Informs the Server that a player is trying to play a card.

Direction: Client to Server:Server

Content:

- Enum MessageType (transmitted as uint)
- List of Card card (transmitted as uint)
- Slot slot (passed as uint)

Format: as JSON string

Example:

```
{
	"messageType": 8,
	"content":
		{
			"card": [45],
			“slot”: 6
		}
}
```

4.1.8 MESSAGETYPE_CLIENT_ACTION_EVENT

Purpose: *Contains the information which client action is performed.*

Direction: Client to Server:Server

Content:

- Enum MessageType (transmitted as uint)
- Enum Action (cardclick, battleFieldClick, buttonclickDone, buttonclickPassOn, buttonclickPickUp)
  (transmitted as uint)

Format: as JSON string

Example:

```
{
	"messageType": 8,
	"content":
		{
			"action": 3
		}
}
```

4.1.9 MESSAGETYPE_CLIENT_CONNECT_EVENT

Purpose: *Contains information about the player.*

Direction: Client:MasterNode to Server:Server

Content:

- Enum MessageType (transmitted as uint)
  Username (transmitted as string)

Format: as JSON string

Example:
```
{
      "messageType": 8,
      "content":
            {
                  "username": "Noah"
            }
}
```