

Software Design Specifications

for

Blackjack

Version 1.0 approved

Prepared by Severin Klapproth, Flavia Taras, Luca Wolfart,
Pascal Engeler, Noé Canevascini, Stanislaw Piasecki

Jack Blacks

29.03.2022

Table of Contents

| | |
|---|----|
| Introduction | 1 |
| Purpose | 1 |
| Document Conventions | 1 |
| Intended Audience and Reading Suggestions | 1 |
| Product Perspective | 1 |
| Static Modeling | 2 |
| Package General | 2 |
| Class GameState | 2 |
| Class Player | 2 |
| Class Shoe | 3 |
| Class Card | 3 |
| Class Diagram of Package General | 3 |
| Class BlackJack | 4 |
| Class GameControl | 4 |
| Class GUI_Window | 4 |
| Class ConnectionPanel | 5 |
| Class BetPanel | 5 |
| Class MainGamePanel | 5 |
| Class ClientNetworkManager | 5 |
| Class ResponseListenerThread | 6 |
| Class Diagram of Package Client | 6 |
| Package Server | 7 |
| Class server_network_manager | 7 |
| Class player_manager | 7 |
| Class game_instance | 7 |
| Class request_handler | 8 |
| Class client_requests | 8 |
| Class server_response | 8 |
| Class Diagram of Package Server | 9 |
| Composite Structure Diagram | 10 |
| Sequence Diagrams | 11 |
| Sequence Join Game | 11 |

| | |
|----------------------------|----|
| Sequence Perform an Action | 12 |
| Sequence End of Round | 13 |
| Interface Modeling | 15 |
| Interface Server_2_Client | 15 |
| join_game_request | 15 |
| start_game_request | 15 |
| make_bet_request | 16 |
| action_request | 16 |
| answer_rqst_response | 17 |
| change_gamestate_msg | 17 |

Revision History

| Name | Date | Release Description | Version |
|-----------------|--------|---|---------|
| Felix Friedrich | 4/1/22 | Template for Software Engineering Course in ETHZ. | 0.2 |
| Jack Blacks | 4/1/22 | Design Specification for Blackjack | 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 General

This package contains the classes GameState, Player, Shoe and Card which describe the game state and game logic. An overview of this package can be found in Figure 1.

Class GameState

This class holds the state of the game and controls the actions which are performed by the players.

The class attributes are:

- max_number_rounds: int, set to 100
- max_number_players: int, set to 5
- num_players: int, the number of players, needed to compute players' turns
- players: vector<Player>, vector of players that participate in the game
- turn: int, stores whose turn it is. Players are numbered from 0 to number_players – 1
- round: int, holds the number of the current round. First round is round 0
- dealer_hand: vector<Card>, represents the hand of the dealer
- min_bet: int, the smallest bet a player can make
- shoe: Shoe, stores the card shoe which is used in the game

The class operations are:

- next_turn: void, increases turn counter, makes player perform their possible actions
- start_round: void, makes the dealer have their score computed, makes players take their turn, increases round counter when last player has taken their turn
- check_winner: bool, compares the scores of the players to determine the winner, true if player won
- compute_dealers_hand: vector<Card>, computes the hand that the dealer gets by performing the actions which are dictated by the blackjack rules, returns their number of points
- show_first_card: void, shows the dealer's first card

Class Player

This class contains information about the state of every player.

The class attributes are:

- money: int, the capital that the player possesses
- bet_size: int, how much money the player has bet in a round
- cards: vector<Card>, the cards the player has
- has_insurance: bool, true if a player has taken insurance for that round
- has_doubled_down: bool, true if a player has decided to double down
- player_name: string, name that a player has chosen
- player_id: int, identification number of a player

The class operations are:

- hit: void, a player takes one more card
- stand: void, a player does not take another card, their turn ends
- take_insurance: void, sets has_insurance to true
- double_down: void, doubles the size of the bet, sets has_doubled_down to true
- get_points: int, computes the number of points based on the cards the player has
- is_broke: bool, checks if the player has more capital than the minimum required bet size
- check_if_over_21: bool, checks if the player's points amount to over 21
- check_if_less_than_dealer: int, returns -1 if the player has less points than the dealer, 0 if they have the same number of points and 1 if the player has more points than the dealer

- win_round: void, increases the player’s capital by the bet size, or by half the bet size if they had a blackjack
- lose_round: void, deducts the bet size from the player’s capital

Class Shoe

This class holds the state of the shoe, i.e., many card decks.

The class attributes are:

- cards: vector<Card>, stores all the cards that are in the shoe

The class operations are:

- pop_card: Card, removes the last card of the cards vector and returns it
- shuffle_cards: void, replenishes the shoe and shuffles the cards

Class Card

This class contains the description of a card.

The class attributes are:

- suit: char, suit of the card, one of “c”, “p”, “h”, “d”
- value: char, the value written on the card, so “7”, “Q”, “A”, ...
- point_value: int, point value of the card in Blackjack

Class Diagram of Package General

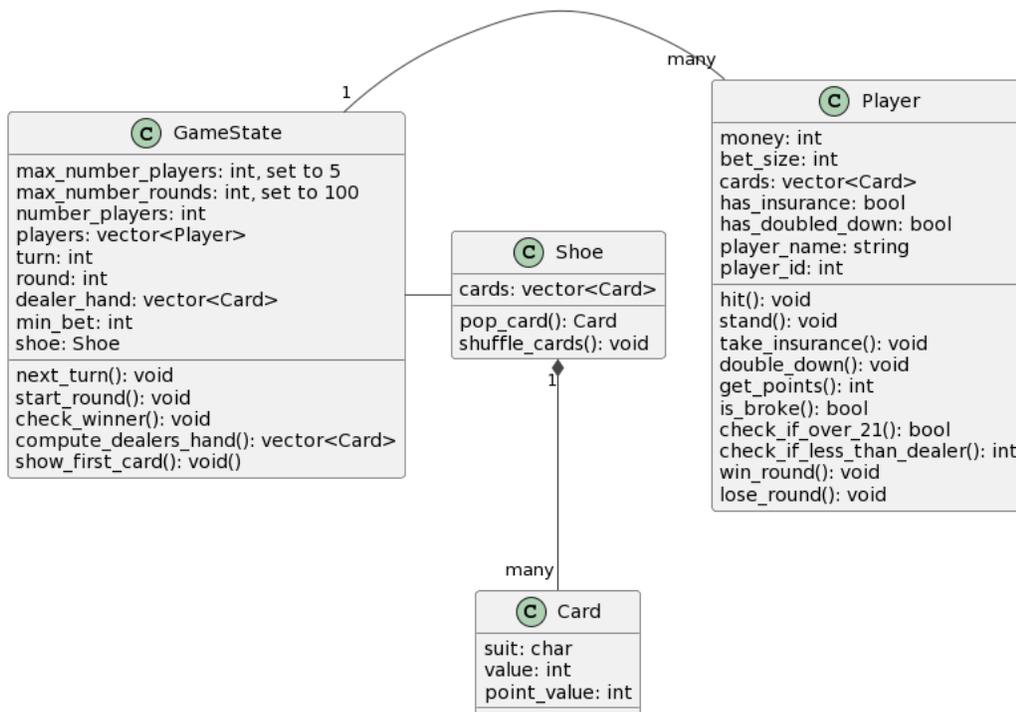


Figure 1: Class diagram of package General.

Package Client

This package contains all classes and functions required to run the game client and play Blackjack. It is responsible for GUI, the interaction of the user with the game interface, the communication between client and server and running the game locally. An overview of the package can be found in Figure 2.

Class BlackJack

This class is used to initialize our application and GUI for a player that wants to run the game. It is derived from the wxApp class defined in the wxWidgets GUI library.

The class operations are:

- OnInit: bool, returns bool which indicates whether the processing should continue, derived from wxApp, initializes the game client

Class GameControl

The GameControl class serves as the controller of ongoing actions for the client. It manages user interaction with the Graphical User Interface, but also the connection to the server, and it controls the current state of the game.

The class attributes are:

- gameWindow: GUI_Window, main game window, current panel
- connectionPanel: ConnectionPanel, connection panel used to connect to the game
- mainGamePanel: MainGamePanel, the main game panel used in the game
- betPanel: BetPanel, the bet panel used to make bets
- my_id: int, id of the player interacting with the client
- currentGameState: GameState, the latest game state

The class operations are:

- init: void, initializes all panels and displays the connection panel
- connectToServer: void, reads the user inputs on the connection panel and sends a join_game request
- updateGameState: void, saves the latest game state in currentGameState
- startGame: void, sends a start_game request to the server and can only start a game if at least two and at most five players are in lobby
- hit: void, sends a hit action_request to the server
- stand: void, sends a stand action_request to the server
- split: void, sends a split action_request to the server
- double_down: void, sends a double_down action_request to the server
- insure: void, sends an insure action_request to the server
- showNewRoundMessage: void, displays a box showing the current round number, the winnings/losses computed in the previous round and that next round is about to start
- showGameOverMessage: void, displays a box showing the winning/losing message and a button to leave the game

Class GUI_Window

This class represents the window of our application. It is responsible for displaying the correct panels to the user. It is derived from the wxFrame class in the wxWidgets GUI library.

The class attributes are:

- currentPanel: wxPanel, the panel to be displayed in the game window

The class operations are:

- showPanel: void, displays the given panel

Class ConnectionPanel

This class represents the GUI panel that the user uses to input the data needed to host or join a game hosted on the server. It is derived from the wxPanel class in the wxWidgets GUI library.

The class attributes are:

- serverAddress: string, stores the address of the server
- serverPort: string, stores the port of the server
- playerName: string, stores the username picked by the player

The class operations are:

- getServerAddress: string, returns the server address
- getServerPort: string, returns the server port
- getPlayerName: string, returns the player's name

Class BetPanel

The BetPanel class is the panel that the user interacts with to place their bet. It is derived from the wxPanel class in the wxWidgets GUI library.

The class attributes are:

- betSize: int, holds the size of the players bet
- playerMoney: int, holds the amount of money the player has (before betting)

The class operations are:

- getBetSize: int, returns the size of the player's bet
- getPlayerMoney: int, returns the amount of money the player has (before betting)

Class MainGamePanel

The MainGamePanel class is the panel that the user interacts with during the actual game. It is derived from the wxPanel class in the wxWidgets GUI library.

The class operations are:

- buildGameController: void, removes existing GUI elements and builds latest game state GUI
- buildOthers: void, builds the GUI elements of other players' hands, money and bets
- buildRoundCounter: void, builds the GUI of the current round number
- buildMyself: void, builds the GUI of the hand, money and bet of the player
- buildShoe: void, builds the GUI of the shoe
- buildDealer: void, builds the GUI of the dealer's hand

Class ClientNetworkManager

This class handles the client-server communication on the client side.

The class attributes are:

- is_connected: bool, true if connecting to the host was successful
- connection: tcp_connector (defined in sockpp library), used to connect to the host and to initialize the ResponseListenerThread

The class operations are:

- `init`: void, creates a connection to a host
- `sendRequest`: void, sends a client request to the server
- `parseResponse`: void, parses a received server response for further processing

Class ResponseListenerThread

The purpose of this class is to listen to the responses of the server to the client and catch them.

The class attributes are:

- `connector`: `tcp_connector` (defined in `sockpp` library), listens to incoming server responses

The class operations are:

- `entry`: void, loop which deals with incoming server responses
- `outputError`: void, communicates errors to the user

Class Diagram of Package Client

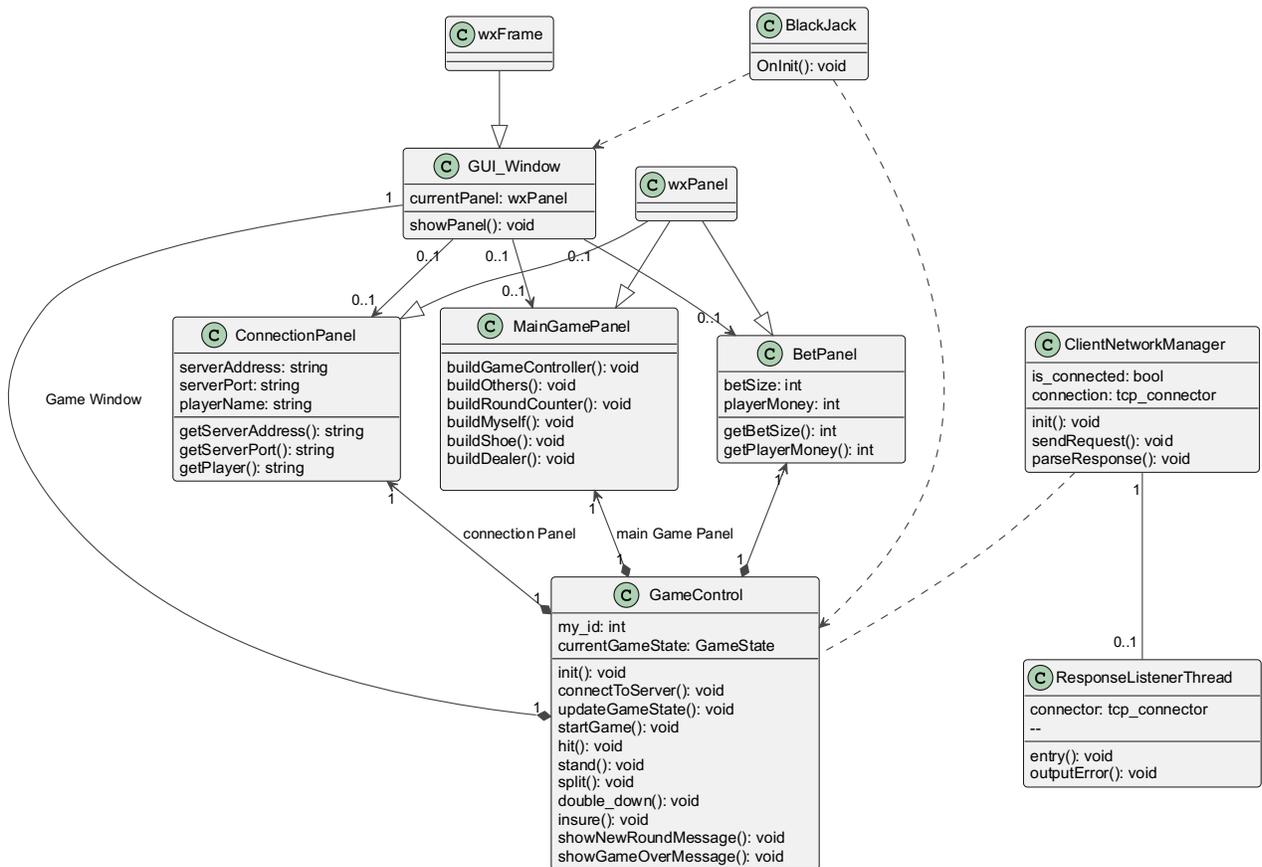


Figure 2: Class diagram of package Client.

Package Server

This package contains all classes and functions required for running a server, which allows multiple players to join the game and play against each other. It is responsible for updating the game state, communication between the server and all clients connected to it, as well as starting and maintaining a game. An overview of the package can be found 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: tcp_acceptor, for incoming connection requests
- player_id_to_address: map, maps the player ids to client addresses
- address_to_socket: map, maps the client addresses to TCP sockets

The class operations are:

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

Class player_manager

Handles player management during a game.

The class attributes are:

- player_map: map, keeps track of the players and their names

The class operations are:

- get_player: bool, retrieves a player from the player map, returns true if successful
- add_player: bool, adds new player to the player map, returns true if successful
- remove_player: bool, removes a player from the player map, returns true if successful

Class game_instance

This class maintains the game instance by tracking the game state based on all server received messages and making sure it is updated. It also passes the updated game information to the server_network_manager class.

The class attributes are:

- game_state: GameState, holds the current updated game state

The class operations are:

- is_started: bool, returns if the game has already started
- is_finished: bool, returns if the game has already finished
- start_game: bool, attempts to start the game, returns true if successful
- add_player: bool, if possible, it adds a player to the game, otherwise it returns false
- try_remove_player: bool, if possible, it removes a player from the game, otherwise it returns false
- hit: bool, performs "hit" action and updates the game state
- stand: bool, performs "stand" action and updates the game state
- split: bool, performs a split (if allowed) and updates the game state

- `double_down`: bool, makes player “double down” (if allowed) and updates the game state
- `insure`: bool, gives player insurance and updates the game state

Class request_handler

Handles different requests from clients: start_game, join_game, make_bet, action.

The class operations are:

- `handle_request`: `request_response*`, handles a `client_request`, changing the game state and returning an appropriate response

Class client_requests

Base class for client requests to the server.

The class attributes are:

- `player_id`: string, ID of the player whose client makes the request
- `RequestType`: enum, either `join_game`, `start_game`, `make_bet`, `hit`, `stand`, `split`, `double_down` or `insure`

Possible requests, implemented as subclasses, are:

- `join_game_request`: requests to join into the game currently hosted by the server
- `start_game_request`: requests to start the game (can only be done by the lobby host)
- `make_bet_request`: requests to make a bet of a certain amount
- `action_request`: requests an action (one of the possible ones during a player’s turn)

Class server_response

Base class for server communication to the client.

Class attributes are:

- `ResponseType`: enum, either `answer_request` or `change_gamestate`

Possible requests, implemented as subclasses, are:

- `answer_rqst_response`: answers directly to a request of a client
- `change_gamestate_msg`: lets all clients know about changes in the game state

Class Diagram of Package Server

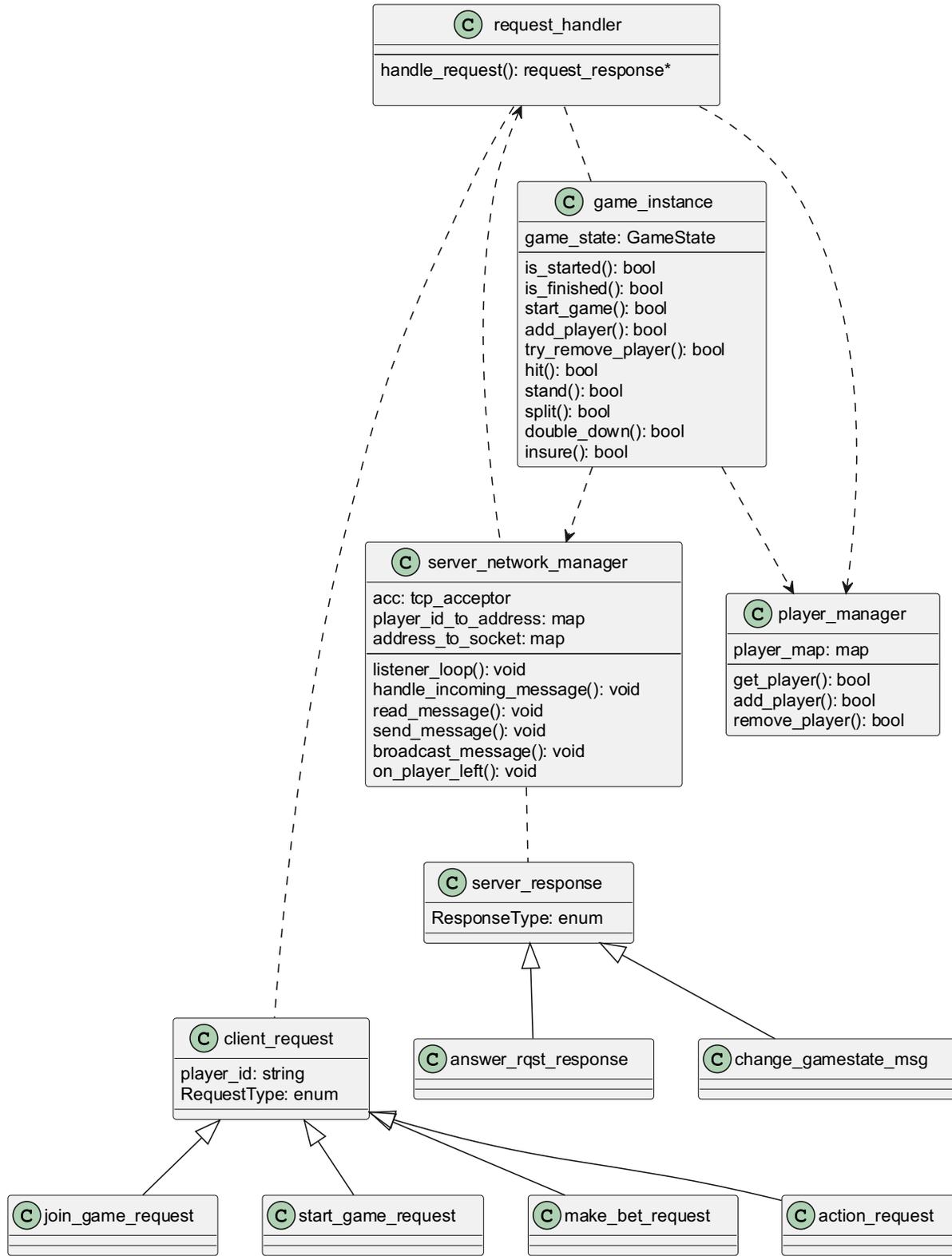


Figure 3: Class diagram of package Server.

Composite Structure Diagram

Figure 4 provides an overview of the Blackjack application structure: Both the Client and the Server package use parts of the Common package. The Client App and the 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.

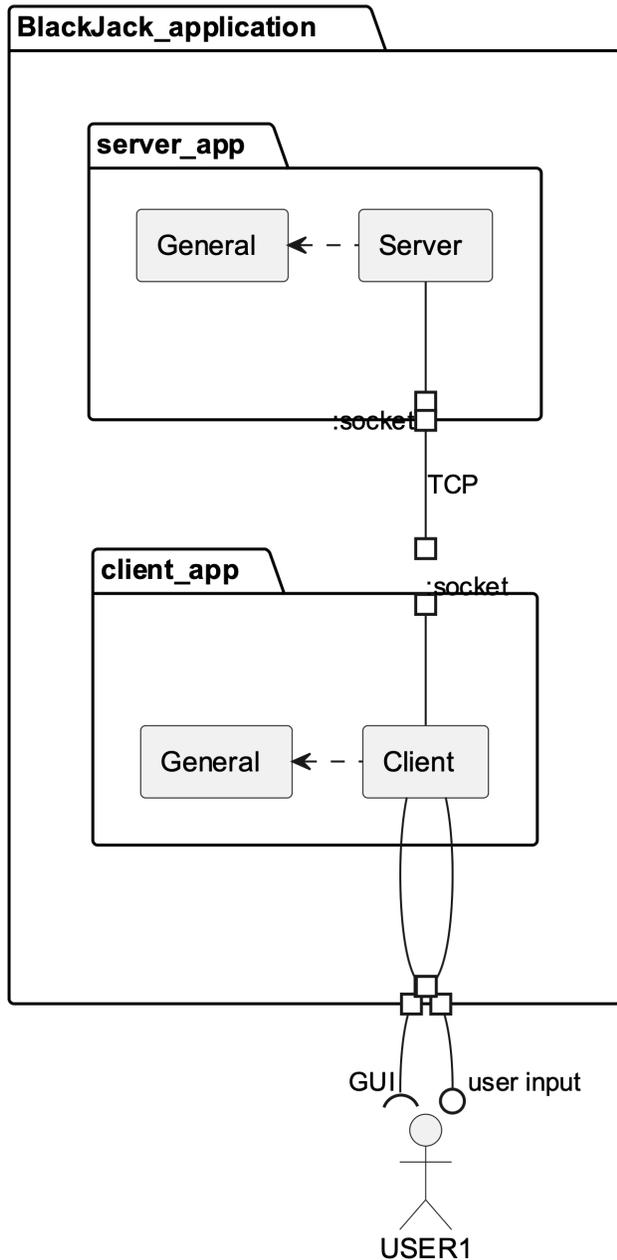


Figure 4: Composite structure diagram of Blackjack application.

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-5: User Input

The scenarios which are related to this sequence are:

- SCN-1: Starting a game

Scenario Narration:

The player fills in their username and the server address in the connection GUI and presses 'connect'. The client sends a 'join_game_request' to the server, where the server tries to add the player to the game. Upon success, the server sends the updated game state to all players. Finally, the server sends the result to the client.

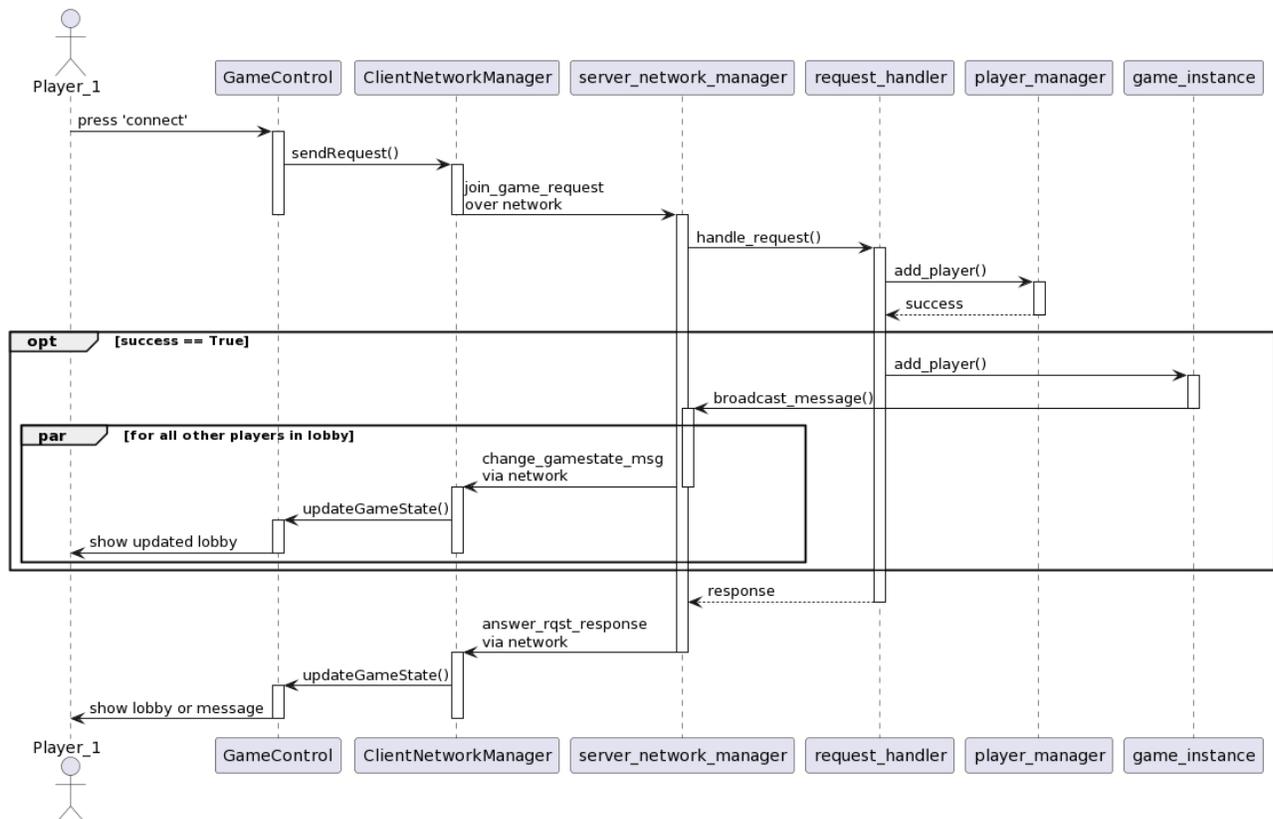


Figure 5: Sequence diagram of sequence Join Game.

Sequence Perform an Action

The player requests one more card.

The functional requirements related to this sequence are:

- **FREQ-1:** Game Server
- **FREQ-4:** GUI
- **FREQ-7:** Turns
- **FREQ-9:** Make a Move
- **FREQ-10:** User Input

The scenarios which are related to this sequence are:

- **SCN-3:** Playing a turn

Scenario Narration:

During their turn, the player decides they want to 'hit', i.e., obtain another card. They press the 'hit' button. The client sends an 'action_request' with action_type="hit" to the server. The server checks if the player is allowed to play, and, if so, pops a card from the shoe and pushes it to the player's cards. The server recomputes the player's score and sends the updated game state to all other players, whose clients update their GUIs. The server reports back to the initiating player.

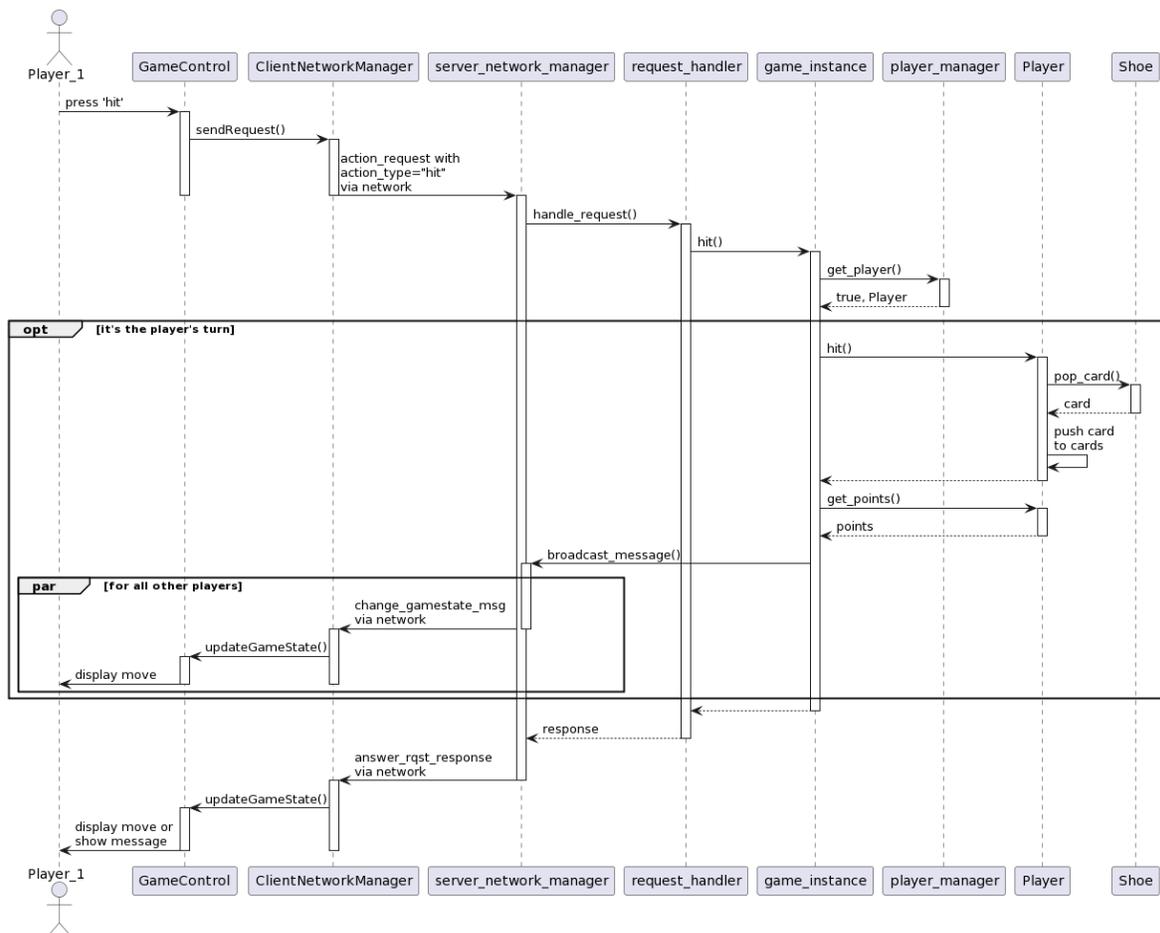


Figure 6: Sequence diagram of sequence Perform an Action.

Sequence End of Round

The last player of the round finishes their turn. The round ends.

The functional requirements related to this sequence are:

- **FREQ-1:** Game Server
- **FREQ-4:** GUI
- **FREQ-7:** Turns
- **FREQ-9:** Make a Move
- **FREQ-10:** User Input
- **FREQ-11:** Dealer's turn / Dealer AI
- **FREQ-12:** Round End & Returns

The scenarios which are related to this sequence are:

- *SCN-3: Playing a turn*
- *SCN-4: Ending a round*

Scenario Narration:

During their turn, the player decides they want to 'stand', terminating their turn. The client sends an 'action_request' with action_type="stand" to the server. The server checks if the player is allowed to play, and, if so, either the next turn or the steps to end a round are initiated. When the round is over the server computes the hand of the dealer and the players that lost or won the round. Then the game state is updated accordingly and sent to all players. The GUI of each player is updated and displays the corresponding end of round screen.

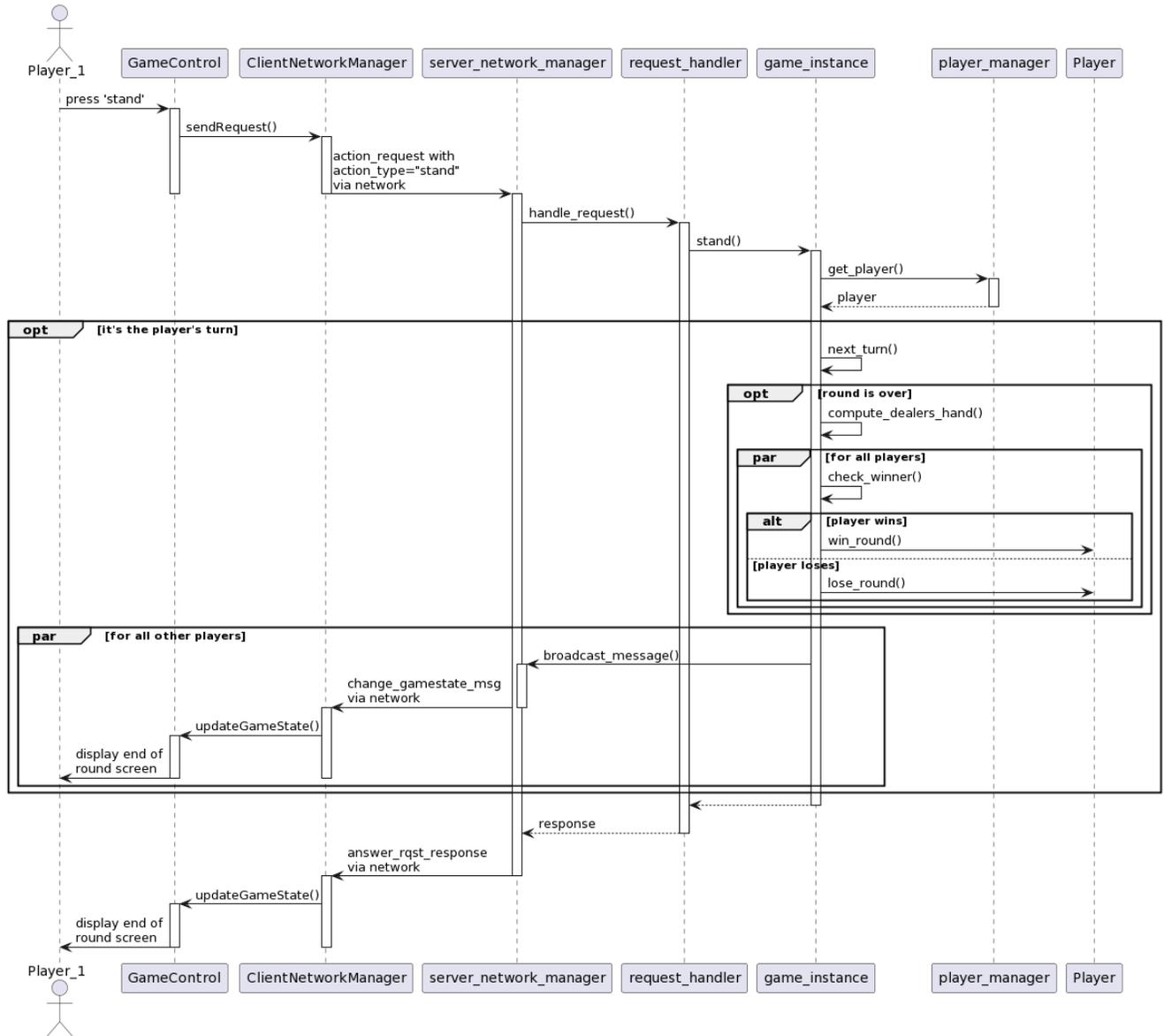


Figure 7: Sequence diagram of sequence End of Round.

Interface Modeling

Interface Server_2_Client

This interface is required for communication between server and game clients of the players for unified overview of the game state and exchanging information between the two. It is essential for allowing players to send requests to the server, receive updates on the game state and communicate errors.

Purpose: Exchanging information between Server and Client

Communication between: Server and Client, initiated by Client

Protocol: TCP

Communication modes: Client request – Server response (1 to 1) + Server broadcast (1 to All)

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

join_game_request

Purpose: request from player (client) to join a game.

Direction: Client to Server.

Content:

- type : string (required),
- playerID: string (required),
- player_name: string (required).

Format: as JSON string.

Example:

```
{
  "type": "join_game",
  "playerID": "1678",
  "player_name": "Player_791"
}
```

Expected response: answer_rqst_response.

start_game_request

Purpose: request from game host Client to Server to start the game.

Direction: Client to Server.

Content:

- type : integer (required),

- playerID: string (required).

Format: as JSON string.

Example:

```
{
  "type": "start_game",
  "playerID ": "5330"
}
```

Expected response: answer_rqst_response.

make_bet_request

Purpose: player request to place a bet.

Direction: Client to Server.

Content:

- type: string (required),
- playerID: string (required),
- bet: integer (required).

Format: as JSON string.

Example:

```
{
  "type": "make_bet",
  "playerID ": "6892",
  "bet": 10
}
```

Expected response: answer_rqst_response.

action_request

Purpose: player request to perform one of the available actions during their turn (hit, stand, etc.).

Direction: Client to Server.

Content:

- action_type: string (required),
- playerID: string (required).

Format: as JSON string.

Example:

```
{
```

```
    "action_type": "stand",
    "playerID": "3456",
  }
```

Expected response: answer_rqst_response.

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

answer_rqst_response

Purpose: answer a request from a Client.

Direction: Server to Client.

Content:

- type : string (required),
- playerID: string (required),
- error: string,
- success: bool (required),
- game_state: string (required).

Format: as JSON string.

Example:

```
{
  "type": "answer_rqst_response",
  "playerID": "5326",
  "error": ""
  "success": true,
  "game_state": <serialization of class GameState in JSON format>
}
```

Expected response: -----

change_gamestate_msg

Purpose: inform all clients about changes in game state and provide them with its newest version.

Direction: Server to Clients.

Content:

- type : integer (required),
- game_state: string (required).

Format: as JSON string.

Example:

```
{
```

```
"playerID": "4325",  
  "game_state": <serialization of class GameState in JSON format>  
}
```

Expected response: -----