# 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](#_Toc99554005)

[Purpose 1](#_Toc99554006)

[Document Conventions 1](#_Toc99554007)

[Intended Audience and Reading Suggestions 1](#_Toc99554008)

[Product Perspective 1](#_Toc99554009)

[Static Modeling 2](#_Toc99554010)

[Package General 2](#_Toc99554011)

[Class GameState 2](#_Toc99554012)

[Class Player 2](#_Toc99554013)

[Class Shoe 3](#_Toc99554014)

[Class Card 3](#_Toc99554015)

[Class Diagram of Package General 3](#_Toc99554016)

[Package Client 3](#_Toc99554017)

[Class BlackJack 3](#_Toc99554018)

[Class GameControl 3](#_Toc99554019)

[Class GUI\_Window 4](#_Toc99554020)

[Class ConnectionPanel 4](#_Toc99554021)

[Class BetPanel 4](#_Toc99554022)

[Class MainGamePanel 4](#_Toc99554023)

[Class ClientNetworkManager 5](#_Toc99554024)

[Class ResponseListenerThread 5](#_Toc99554025)

[Class client\_requests 5](#_Toc99554026)

[Class server\_response 6](#_Toc99554027)

[Package Server 6](#_Toc99554028)

[Class server\_network\_manager 6](#_Toc99554029)

[Class player\_manager 6](#_Toc99554030)

[Class game\_instance 6](#_Toc99554031)

[Class request\_handler 7](#_Toc99554032)

[Class Diagram (next Page) 7](#_Toc99554033)

[Composite Structure Diagram 9](#_Toc99554034)

[Sequence Diagrams 10](#_Toc99554035)

[Sequence <Name> 10](#_Toc99554036)

[<Another Sequence Name> 10](#_Toc99554037)

[Interface Modeling 11](#_Toc99554038)

[Interface <Name> 11](#_Toc99554039)

[<Message Name> 11](#_Toc99554040)

[*<Another Message Name>* 11](#_Toc99554041)

[*<Another Interface Name>* 11](#_Toc99554042)

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Date | Release Description | Version |
| **Felix Friedrich** | 3/31/22 | Template for Software Engineering Course in ETHZ. | 0.2 |
| Jack Blacks | 3/31/22 | 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.*

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,holds number of players, needed to compute players turns,
* players: vector<Player>, vector of players which participate in the game,
* turn: int, holds which players turn it is. Players are numerated from 0 to number\_players – 1,
* round: int, holds the number of the current round. First round is round 0,
* min\_bet: int, holds the smallest bet a player can make,
* dealer\_hand: vector<Card>, represents the hand of the dealer,
* shoe: Shoe, holds the card shoe which will be used for the game.

The class operations are:

* next\_turn: void, increase turn counter, make player perform his available actions,
* start\_round: void, make the dealer compute his score, make players take their turn, increase round counter when last player has taken turn,
* check\_winner: bool, compare the scores of the player to determine 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 his number of points,
* show\_first\_card: void, show the dealers first card.

Class Player

This class holds the information about the state of every player.

The class attributes are:

* money: int, the capital which every player possesses,
* bet\_size: int, how much money a player has bet in a round,
* cards: vector<Card>, the cards which a player has,
* has\_insurance: bool, true if a player has taken insurance for a round,
* has\_doubled\_down: bool, true if a player has decided to double down.
* player\_name: string, name that a player has chosen

The class operations are:

* hit: void, a player takes one more card,
* stand: void, a player doesn’t take another card, his turn ends,
* take\_insurance: void, set has\_insurance to true,
* double\_down: void, double the size of the bet, set has\_doubled\_down to true,
* get\_points: int, compute the amount of points based on the cards a player has,
* is\_broke: bool, check if player has more capital than the minimal bet,
* check\_if\_over\_21: bool, check if the points are over 21,
* check\_if\_less\_than\_dealer: int, returns -1 if player has less points than the dealer, 0 if he has the same and 1 if the player has more points than the dealer,
* win\_round: void, increase capital by bet size, or half the bet size if he has a blackjack,
* lose\_round: void, deduct bet size from capital.

Class Shoe

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

The class attributes are:

* cards: vector<Card>, holds all the cards which are in the shoe.

The class operations are:

* pop\_card: Card, removes the last card of the vector and returns it,
* shuffle\_cards: void, replenishes the shoe and shuffles the cards.

Class Card

This class holds the description of a card.

The class attributes are:

* suit: char, suit of the card, one of “c”, “p”, “h”, “d”
* value: char what is on the card, so “7”, “Q”, “A”, …
* point\_value: int, point value of card in Blackjack

Class Diagram of Package General

Diagram

Description automatically generated

**Package Client**

*This package contains all classes and functions required to run the game client and play Blackjack. It is responsible for GUI, interaction of user with the game interface, communication between client and server and running the game locally.*

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:

* bool OnInit: returns bool which indicates whether processing should continue, derived from wxApp, initializes our 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 connection to server and controls 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:

* void init: initializes all panels and displays the connection panel,
* void connectToServer: reads the user inputs on the connection panel and sends a join\_game request,
* void updateGameState: saves the latest game state in currentGameState,
* void startGame: sends a start\_game request to the server and then starts a game if at least 2 players are in lobby,
* void hit: sends a hit action\_request to the server,
* void stand: sends a stand action\_request to the server,
* void split: sends a split action\_request to the server,
* void double\_down: sends a double\_down action\_request to the server,
* void insure: sends the insure action\_request to the server,
* void showNewRoundMessage: displays a box showing current round number, the winnings/losses from last round and that next round is about to start,
* void showGameOverMessage: 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 outputting 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:

* void showPanel: 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, holds the address of the server,
* serverPort: string, holds the port of the server,
* playerName: string, holds the username picked by the player.

The class operations are:

* string getServerAddress: returns the server address,
* string getServerPort: returns the server port,
* string getPlayerName: returns the player name.

Class BetPanel

The BetPanel class represents the panel that user interacts with to place his/her bet of choice. 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:

* int getBetSize: returns the size of the players bet,
* int getPlayerMoney: returns the amount of money the player has (before betting).

Class MainGamePanel

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

The class operations are:

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

Class ClientNetworkManager

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

The class attributes are:

* is\_connected: bool, true if connection to host was successful,
* connection: tcp\_connector (defined in sockpp library), used to connect to the host and to initialize the response listener thread.

The class operations are:

* void init: returns the size of the players bet,
* void sendRequest: sends a client request to the server,
* void parseResponse: parses a received server response for further processing.

Class ResponseListenerThread

The purpose of this class is to listen to 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:

* void entry: loop which deals with incoming server responses,
* void outputError: communicates error to the user.

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,*
* type: enum RequestType: *either join\_game, start\_game, make\_bet, hit, stand, split, double\_down, insure.*

Different 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 first player who joined the game,*
* make\_bet\_request: *requests to make a bet of a certain amount,*
* action\_request: *requests an action (one of possible during turn).*

Class server\_response

*Base class for server communication to the client.*

Class attributes are:

* type: enum ResponseType: *either answer\_request or change\_gamestate.*

Different possible requests, implemented as subclasses, are:

* answer\_rqst\_response: *answers directly to a request of a client,*
* change\_gamestate\_msg: *tells all clients about a change of the game state.*

Class Diagram of Package Client

Diagram

Description automatically generated

Package Server

*This package contains all classes and functions required for running a server, which allows many players to join the game and play against each other. It is responsible for updating the current game state, communication between the server and all clients connected to it, but also starting and maintaining game.*

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:

* void listener\_loop: keeps the server running and catches incoming requests
* void handle\_incoming\_message: receives a message and checks the contents
* void read\_message: parses a received message for further processing
* void send\_message: sends a message to a client
* void broadcast\_message: causes a send to all clients
* void on\_player\_left: 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:

* bool get\_player: retrieves a player from the player map, returns true if successful
* bool add\_player: adds new player to 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:

* bool is\_started: returns if the game has already started
* bool is\_finished: returns if the game has already finished
* bool start\_game: attempts to start the game, returns true if successful
* bool add\_player: if possible adds a player to the game, otherwise return false
* bool try\_remove\_player: if possible removes a player from the game, otherwise return false
* bool hit: performs “hit” action and updates the game state
* bool stand: performs “stand” action and updates the game state
* bool split: performs a split (if allowed) and updates the game state
* bool double\_down: makes player “double down” (if allowed) and updates the game state
* bool insure: gives player insurance and updates the game state

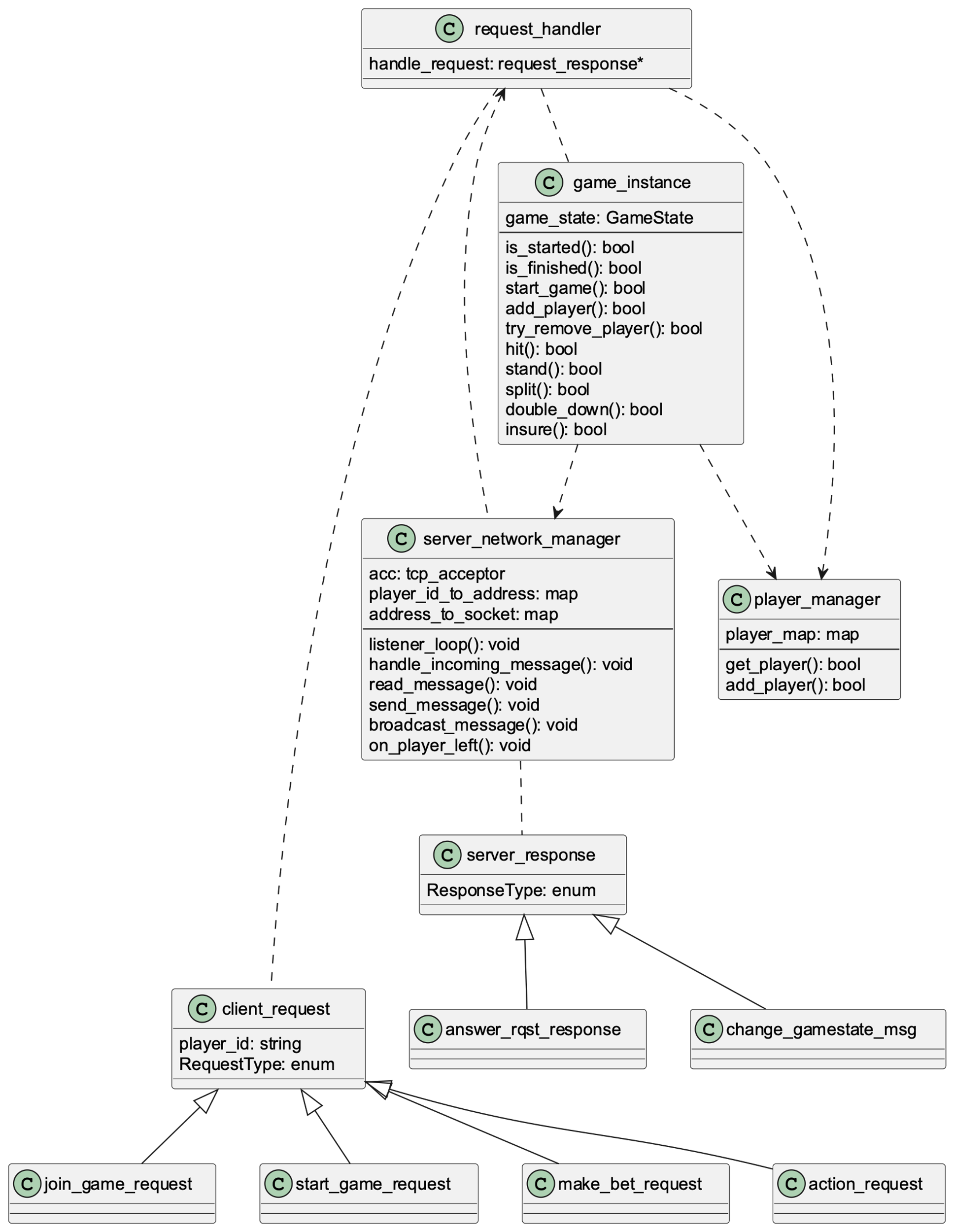
Class request\_handler

Handles different requests from clients: start\_game, join\_game, make\_bet, action.

The class operations are:

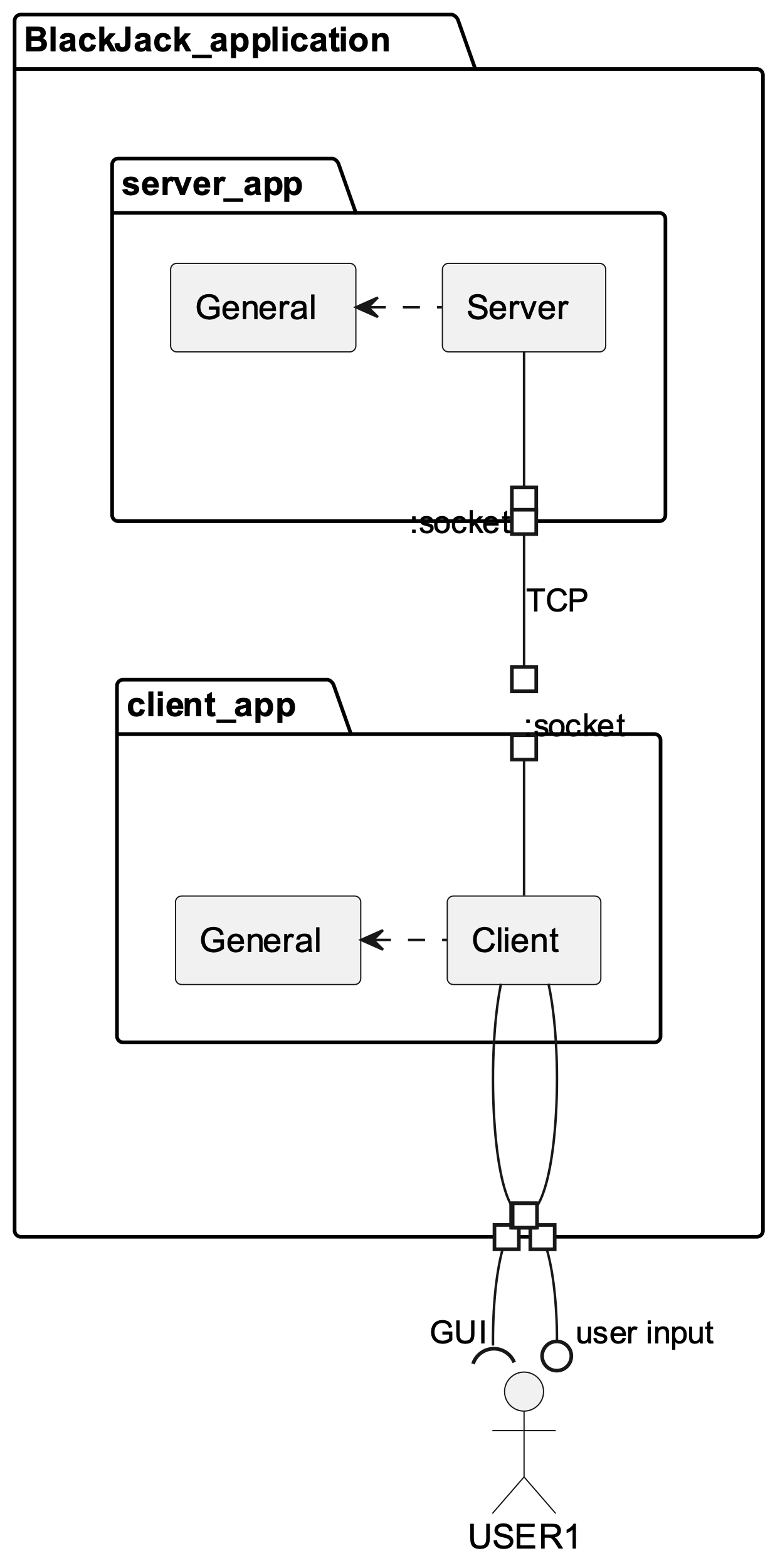
* handle\_request: request\_response\*, handles a client\_request, changing the game state and returning a corresponding response.

Class Diagram of Package Server (next page)



Composite Structure Diagram

The following figure provides an overview of the Blackjack 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.



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.

**A picture containing calendar

Description automatically generated**

**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 a ‘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 recalculates the player's score and sends the updated game state to all players, whose clients update their GUIs. The server then sends the response to the initiating player.

Diagram

Description automatically generated

**Sequence End of Round**

The current round ends.

A picture containing diagram

Description automatically generated

Interface Modeling

Interface Server\_2\_Client

*This interface is required for communication between server and game clients of the players for unified overlook over the game state and exchanging information between the two. It is essential so that players can send requests to the server, receive updates on game state and communicate any 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)

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 available actions during his/her 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.

*answer\_rqst\_response*

**Purpose:** answer to 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 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: -----**