ETH zürich

*FS 2023*

# Software Design Specifications

**for**

# Battleship

**Version 1.0 approved**

**Prepared by Tejas Gupta, Marina Mancoridis, Nico Müller, Armin Riess, Mike Schmid and Robin Sieber**

**Ship Happens**

**31.03.2023**

# Table of Contents

## Revision History

| Name | Date | Release Description | Version |
|---|---|---|---|
| **Felix Friedrich** | 3/22/23 | Template for Software Engineering Course in ETHZ. | 0.2 |
| **Ship Happens** | 3/31/23 | Adapted to Battleship | 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 code for both client and server. This includes mainly the game state representation and the request and response messages needed for client-server-communication.*

### Class GameState

*Represents an ongoing game of battleship.*

The class attributes are:
- grids: std::vector<PlayerGrid>, contains the two ocean grids that players shoot at
- num_players: int, set to 2
- players: std::vector<Player>, contains the two players
- currentPlayerID: int, specifies the id of the players whose turn it is
- turnNumber: int, specifies the current numbered turn
- isFinished: bool, specifies whether or not the game is finished

The class operations are:
- getCurrentPlayer: player*, returns the current player
- removePlayer: bool, removes player from the game
- addPlayer: bool, adds player to the game
- updateBoards: bool, updates the board after a shot has been fired
- wrapUpRound: bool, 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 Player

- name: string, playername
- id: int, unique player id

### Class PlayerGrid

*Represents an ocean grid, i.e. a playing field*

The class attributes are:
- size: const int, set to 10
- owner: player*, player associated with this grid
- shotsReceived: int[size][size], stores the shots the enemy called
- shotsFired: int[size][size], stores the shots the owner called himself
- shipsPlaced: std::vector<ships>(5), stores the ship structs

## Class Ship

*Data container for all the information about a single ship*

The class attributes are:
- length: int, length of the ship
- position: int, position of the top left tile of the ship
- orientation: enum, either vertical or horizontal
- id: int, unique ship id
- hits: bool[length], stores the parts of ship hit
- sunk: bool, indicates if the ship is sunk or still afloat

The class operations are:
- hasSunken: bool, checks if the ship was hit on every of its tiles

## Class ClientRequest

*Base class for all client requests to the server. The different types of requests are specified in the interface modeling section. They will have different additional fields and will be realized in the form of subclasses.*

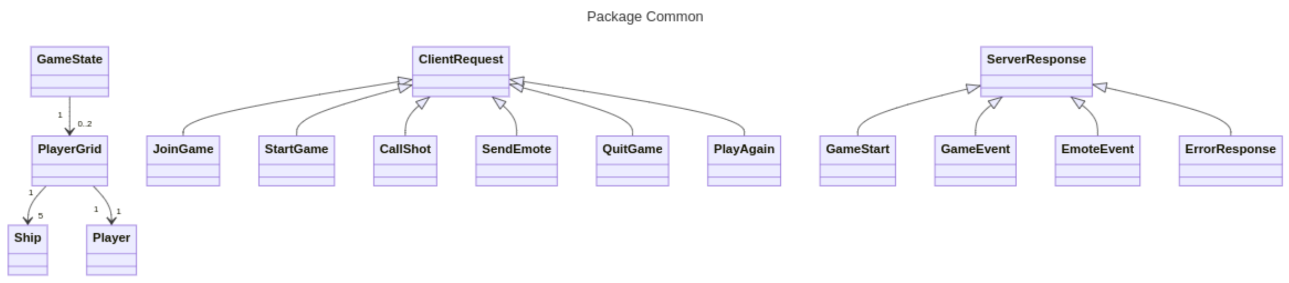
The class attributes are:
- player_id: string, id of the player who sends the request
- type: string, RequestType, either joinGame, startGame, callShot, sendEmote, playAgain or quitGame

## Class ServerResponse

*Base class for server messages to the clients. The different types of responses are specified in the interface modeling section. They will have different additional fields and will be realized in the form of subclasses.*

The class attributes are:
- type: string. ResponseType, either startGame, gameEvent, emoteEvent or errorResponse.

## Class Diagram of Package Common



## 2.2 Package Client

This package contains the classes and methods necessary to run and play a game of Battleship. The code covers the GUI, user interaction with the client as well as sending and receiving messages to/from the server.

### Class Battleship

Provides an entry point to the client application. This class is derived from the wxApp class defined in the wxWidgets GUI library.

Class Operations:

- onInit: application entry point

### Class GameController

The GameController is the central controller of the client, it handles events, triggers the server connection and deals with the interaction of the user with the GUI.

Class Attributes

- gameWindow: main game window holding current panel
- connectionPanel: panel used for user to establish a connection to the server
- setupPanel: panel used for user to place ships on the grid and notify server that they are ready to start the game
- me: information about player interacting with the client
- setupManager: handles ship placement during setup phase and sends information to server when player is ready
- currentGameState: game state containing relevant information for the current round of the game
- emoteHandler: handles emotes that one player can send to the opponent. Creates message from user input and sends it to the server

Class Operations

- init: initializes all panels and displays the connection panel s.t. user can establish a connection
- connectToServer: reads user input on connection panel and connects to server
- startGame: sends a start game request after setup phase. Transmits the coordinates of the placed ships (stored in setupManager) to the server and starts the game.
- callShot: sends a shot request to the server
- sendEmote: sends an emote to the server
- showError: prints an error message box in case of an invalid move
- showGameOverMessage: displays dialog box when game is finished and gives the players the option to start a new game or leave the game.

### Class GameWindow

Main window of the application, responsible for the display of panels. This class is derived from the wxFrame class defined in the wxWidgets GUI library.

Class Attributes:
- currentPanel: the panel to be displayed in the game window
- statusBar: displays the connection status to the user

Class Operations:
- showPanel: switches view panel displayed
- setStatus: sets status to display in status bar

### Class ConnectionPanel

Panel containing the connection GUI, displays input fields for the user to join a game with the client, derived from the wxPanel class defined in the wxWidgets GUI library.

Class Attributes:
- serverAddressField: holds server address
- serverPortField: holds port
- playerNameField: holds username of the player

Class Operations:
- getServerAddress: returns address used for connection
- getServerPort: returns the port used for connection
- getPlayerName: returns username

### Class SetupPanel

Panel containing a grid and ships s.t. player can place the ships on the grid. Also contains a ready button which initiates the server

Class Attributes:
- grid: 10x10 grid where player can place their ships
- ship view: overview of the ships that the player hasn't placed yet.
- instruction text box: text that instructs player how to place and rotate ships
- button: ready button that can be pressed as soon as all ships are placed

Class Operations:
- placeShip: places the ship on the indicated grid position. Also checks that the placement is valid (no overlap and no out of boundary)
- rotateShip: rotates a ship by 90 degrees when 'R' is pressed on the keyboard

- startGame: activated upon pressing the ready button. Transmits ship placement to server and initiates the game screen server returns start message.

### Class MainGamePanel

Panel containing the in-game GUI, displayed when starting and playing a game in the client, derived from the wxPanel class defined in the wxWidgets GUI library.

Class Operations:
- buildGameState: removes existing GUI elements and builds latest game state GUI (grid with shots fired and grid with shots received)
- buildEmoteList: Builds GUI elements where user can choose an emote if they choose to react on the opponent's shot.
- buildTurnIndicator: builds the turn indicator

### Class ClientNetworkManager

Handles server-client communication on the client side.

Class Attributes:
- connectionSuccess: bool, indicates if connection to host was successful
- failedToConnect: bool, indicates if failed to connect to host
- connection: TCP connector used to connect to the host and to initialize the response listener thread

Class Operations:
- init: creates a connections to a host
- sendRequest: sends a client request to the connected host
- parseResponse: parses a received server response for further processing

### Class ResponseListenerThread

Listener thread to catch incoming server responses

Class Attributes:
- connection: TCP connector on which listener listens for incoming responses

Class Operations:
- Entry: threaded loop which deals with incoming server responses
- outputError: communicates error to the user

### Class SetupManager

Handles the ship placement process and validates and stores the positions until setup is completed and positions are transmitted to the server.

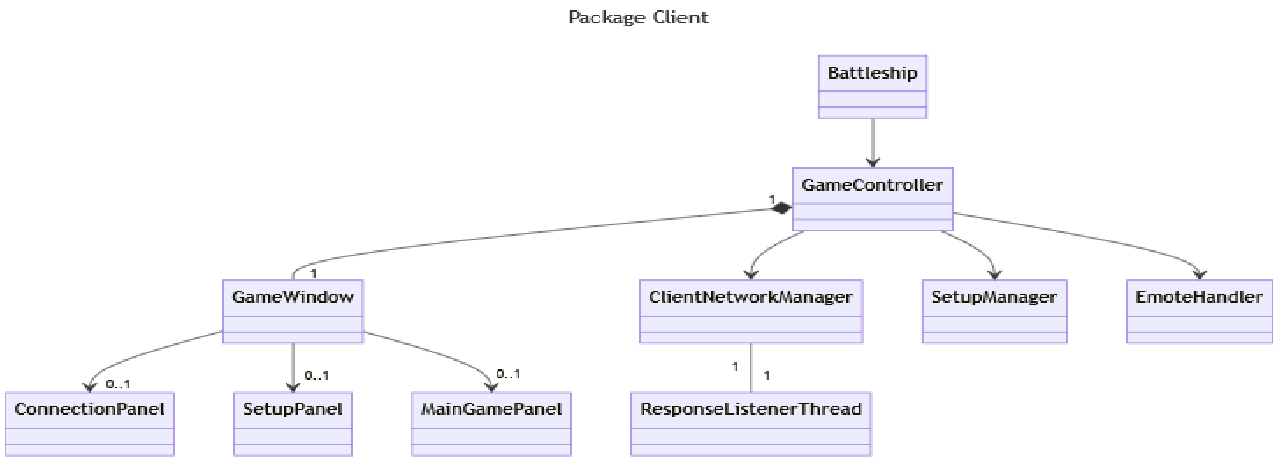
Class Attributes:

- vector<Ship> ships_placed

Class Operations:

- placeShip: handles user interaction with the SetupPanel during setup phase. Validates ship position.
- checkPlacement: bool, returns true if ship position is valid on the grid.
- wrapUpSetup: checks if all 5 ships are placed.

### Class Diagram of Package Client



## 2.3 Package Server

*The package server contains all classes and methods necessary to host a game of Battleship. This includes managing the network connections, handling requests (actions), enforcing the game rules, sending responses (events) and updating the game state. These methods are meant to be run server-side.*

### Class ServerNetworkManager

*Handles server startup, client requests and sending responses to clients. On startup the server will execute a listener loop and handle incoming requests from the clients.*

The class attributes are:

- acc: a TCP acceptor socket for incoming connection requests
- idToAddress: maps player ids to client addresses
- addressToSocket: maps client addresses to TCP sockets

The class operations are:

- listenerLoop: keeps the server running and catches incoming requests

- handleIncomingRequest: receives a request and checks its contents
- sendResponse: sends a response to a specific client
- broadcastResponse: sends a response to all clients
- onPlayerQuit: handles the event that a player quit the game

### Class PlayerManager

Handles player management during the game

The class attributes are:
- players: std::vector<Player>, contains the two players
- readiness: std::vector<bool>, contains info if the players are ready to play

The class operations are:
- getPlayer: retrieves a player
- addPlayer: adds new player
- removePlayer: removes player
- markReady: mark a player that the ships has placed placed
- checkPlayerReadiness: checks if the two player are ready

### Class GameInstance

*Keeps track of the game state and makes sure that events are communicated correctly to all clients through the serverNetworkManager.*

The class attributes are:
- gameState: GameState, the current state holding all relevant information about the game

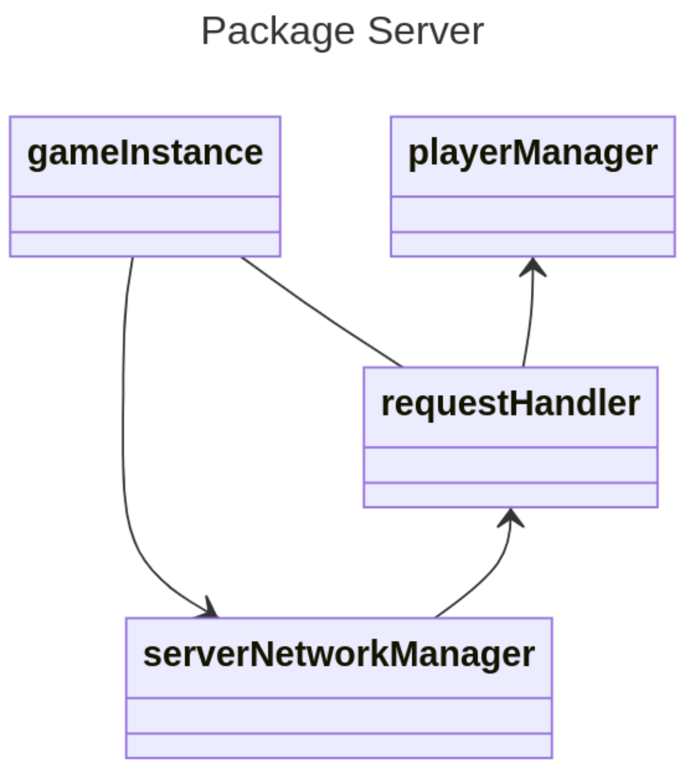
The class operations are:
- executeShot: registers a shot and checks if it is a miss or hit and if it sunk a ship, notifies clients accordingly by emitting game events.
- endGame: ends the game on the server side, gives clients the option to play again
- shutdown: quits the server-side application
- gameStart: used to go from the setup phase to the game phase if the two players are ready

### Class RequestHandler

*Handles received client requests of the types: joinGame, startGame, callShot, sendEmote, playAgain, quitGame*

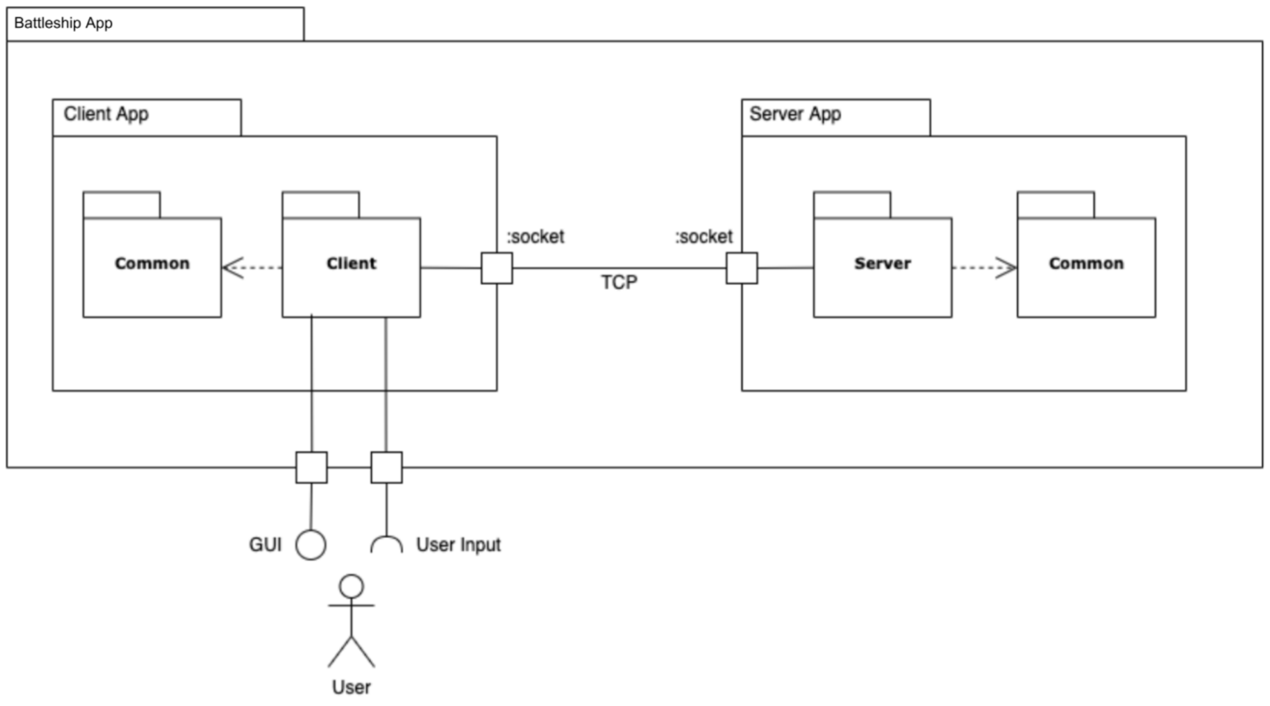
The class operations are:
- handleRequest: ServerResponse*, handles a ClientRequest, changes the gamestate and returns a corresponding ServerResponse

### Class Diagram of Package Server


Package Server
gameInstance
playerManager
requestHandler
serverNetworkManager

## 2.4 Composite Structure Diagram

*The figure below provides an overview of the Battleship 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.*



# 3. Sequence Diagrams

## 3.1 Sequence Join Game

*This sequence describes what happens when a player tries to join the game.*

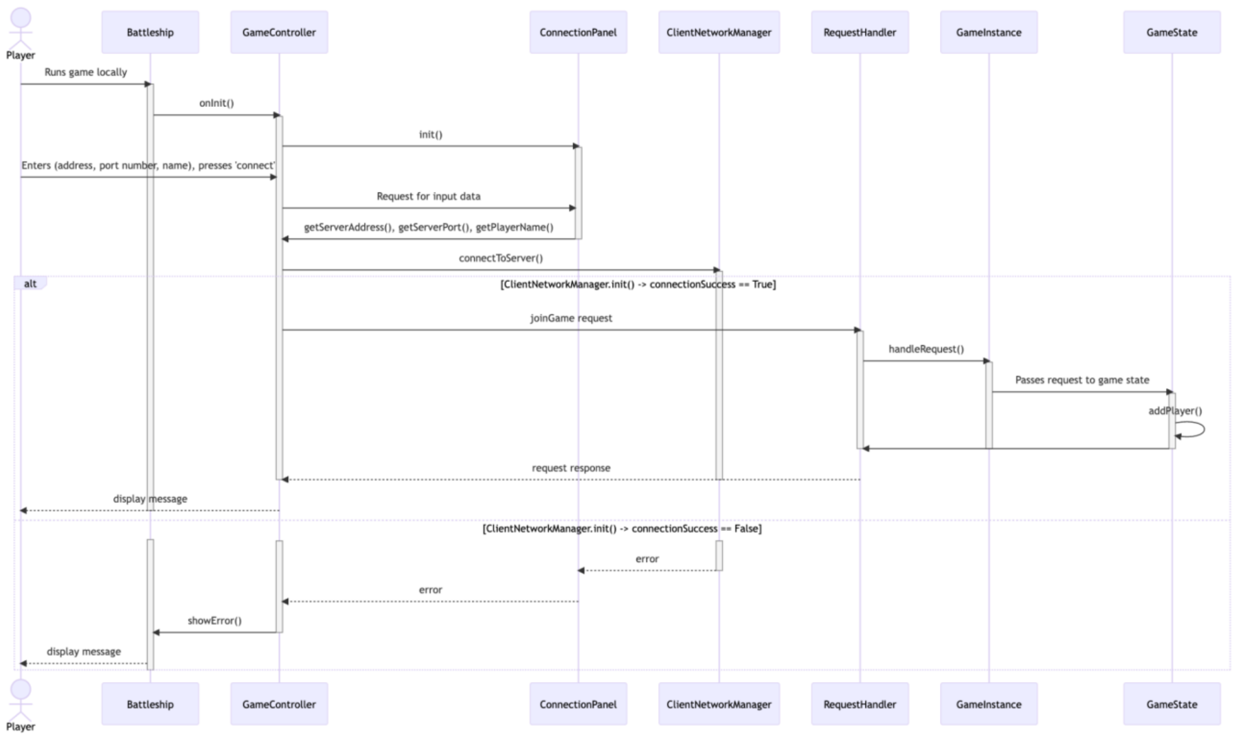
**The functional requirements related to this sequence are:**

FREQ-1: Connecting clients
FREQ-2: Starting the game
FREQ-3: Initializing states and objects for each player
FREQ-4: Initializing states for the collective game
FREQ-19: Performing validity checks on every game state
FREQ-20: Game GUI
FREQ-22: Adhering to appropriate external interfaces

**The scenarios which are related to this sequence are:**

SCN-1: Setting up a game

**Scenario Narration**:
*The player starts the game locally. This creates a Battleship class that then creates a game controller. This propagates the creation of a connection panel. The player uses the connection panel to input their desired address, port number, and name. When the client presses connect, the inputted data is given to the game controller through the connection panel. Then, the game controller attempts to connect to the server. If it fails, error messages are propagated back to the player. If it succeeds, the connection request is sent to the server where it is checked. The server assigns the client to the game and reports back to the client.*



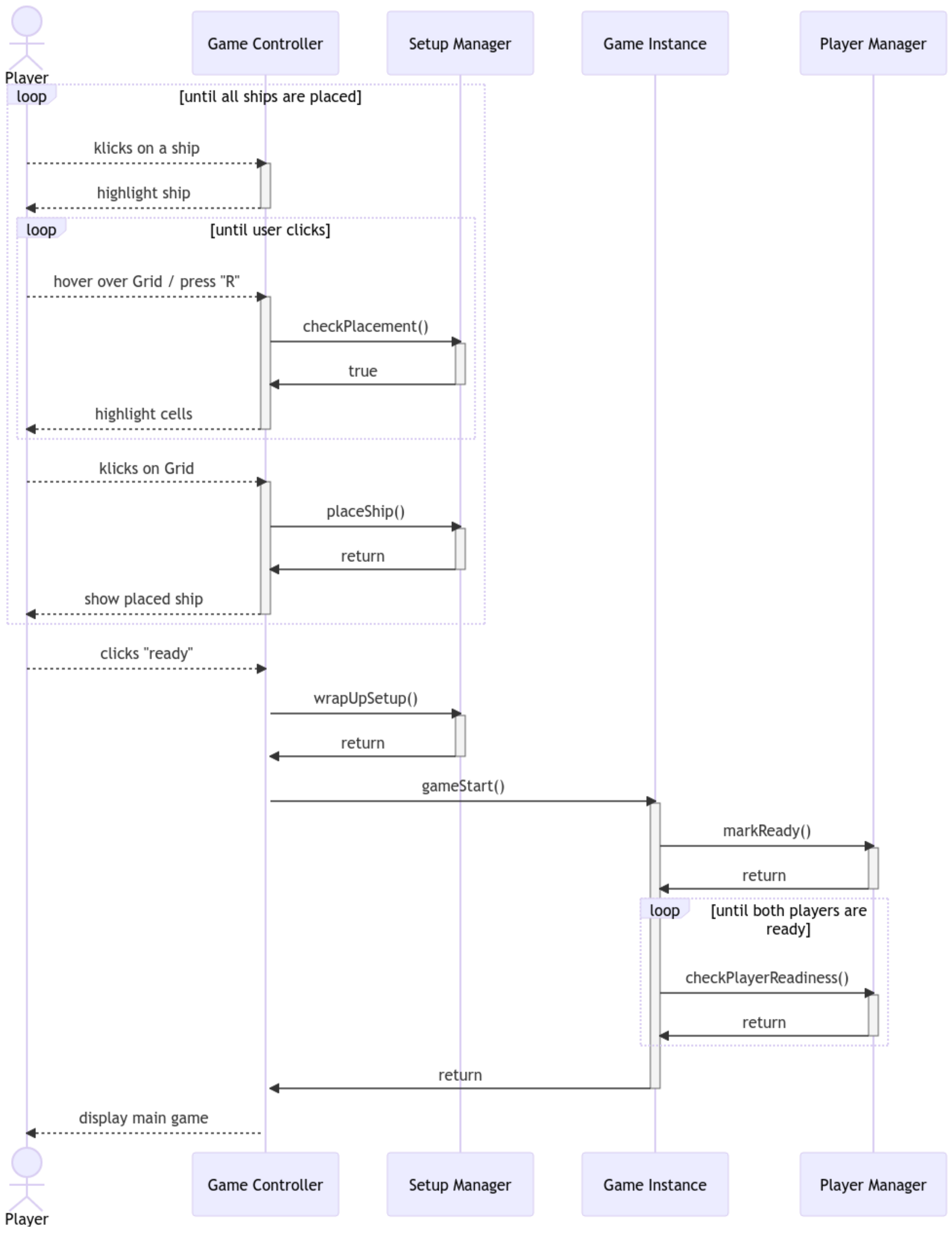
## 3.2 Sequence Ship Placement Phase

*This sequence describes what happens when a player places their ships.*

**The functional requirements related to this sequence are:**
FREQ-6: Placing Ships during Battle Preparation
FREQ-7: Ending Battle Preparation Stage
FREQ-19: Performing validity checks on every game state

**The scenarios which are related to this sequence are:**
SCN-2: Placing the ships

**Scenario Narration**:
*The player is in the ship placement phase. The available ships and the player's grid are shown. The player clicks on a ship to select it. When the player hovers the mouse over the grid, the cells where the ship would be placed are determined by the game state. The game state also checks if the ship can be placed there. If that's the case, the grid cells are highlighted. By pressing "R" on the keyboard, the ship can be rotated. When the player klicks again, the ship is placed on the grid. The game state updates the player's grid. This repeats until every ship is placed. Then, the player can press the "ready" button. The player is marked as "ready" in the game state by the game instance manager. The game state manager keeps checking if the other player is also ready. When that's the case, the game begins.*


Player
Game Controller
Setup Manager
Game Instance
Player Manager
loop
[until all ships are placed]
klicks on a ship
highlight ship
loop
[until user clicks]
hover over Grid / press "R"
checkPlacement()
true
highlight cells
klicks on Grid
placeShip()
return
show placed ship
clicks "ready"
wrapUpSetup()
return
gameStart()
markReady()
return
loop
[until both players are ready]
checkPlayerReadiness()
return
return
display main game
Player
Game Controller
Setup Manager
Game Instance
Player Manager

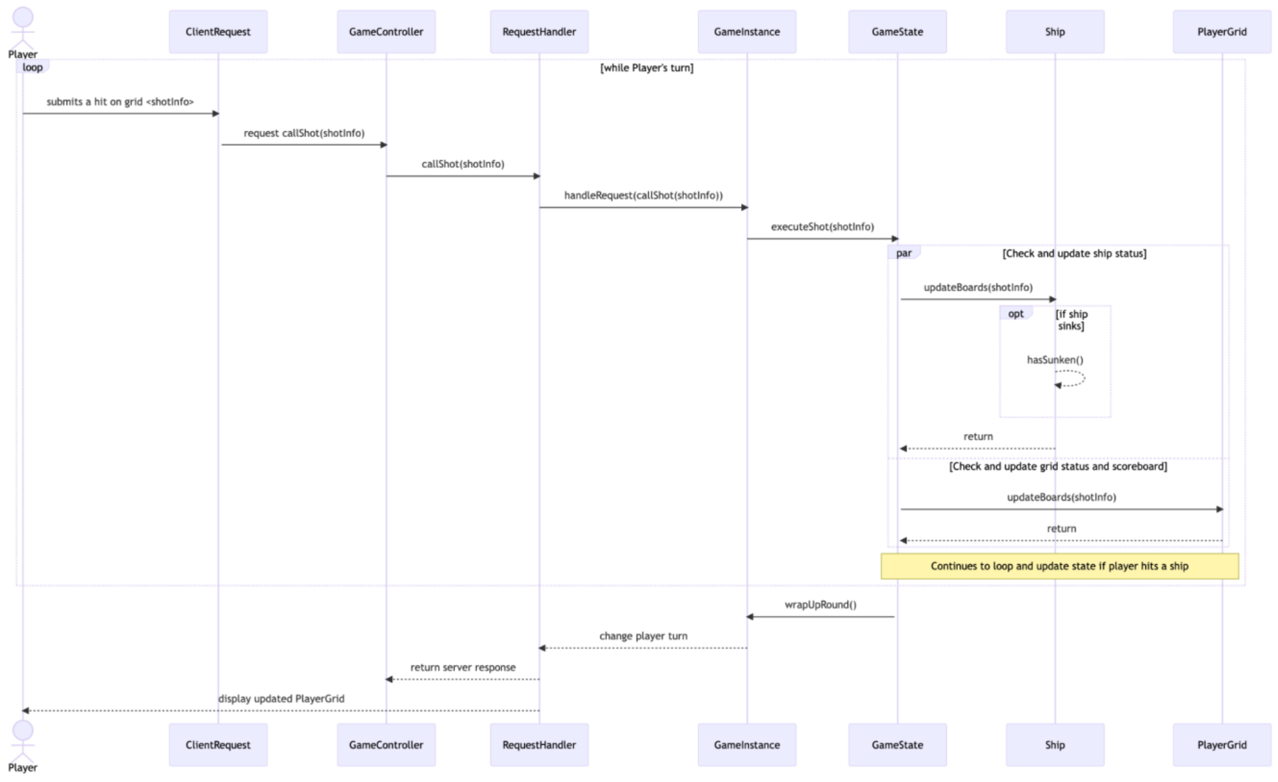
## 3.3 Sequences Calling an unsuccessful shot (miss) & Calling a successful shot (hit)

*This sequence describes what happens when a player calls a shot during his or her turn.*

**The functional requirements related to this sequence are:**
- FREQ-9: Calling a shot
- FREQ-10: Keeping the scoreboard updated
- FREQ-11: Encountering a hit
- FREQ-12: Encountering a miss
- FREQ-13: Performing validity checks on every game state
- FREQ-20: Game GUI
- FREQ-22: Adhering to appropriate external interfaces

**The scenarios which are related to this sequence are:**
- SCN-4: Calling an unsuccessful shot (miss)
- SCN-5: Calling a successful shot (hit)

**Scenario Narration**:
*It is the player's turn. The server receives a ClientRequest from the player to call a shot on the GameController. The shot is executed by the game instance as the game state is updated. When the game state is updated, each ship is checked whether it is hit. If it is hit, the player continues to call a shot until the player misses a ship. The game state updates after every shot (missed or hit). When the game state is updated, each ship is checked for whether or not it has sunk. When the game state wraps up the round, it is informed that each ship has sunk. The game instance changes the player's turn unless the game is over. The server tells the game controller to tell the player his or her turn is over.*



## 3.4 Sequence Ending a Game

*This sequence describes what happens when a player sinks the final remaining ship and ends the game.*

**The functional requirements related to this sequence are:**

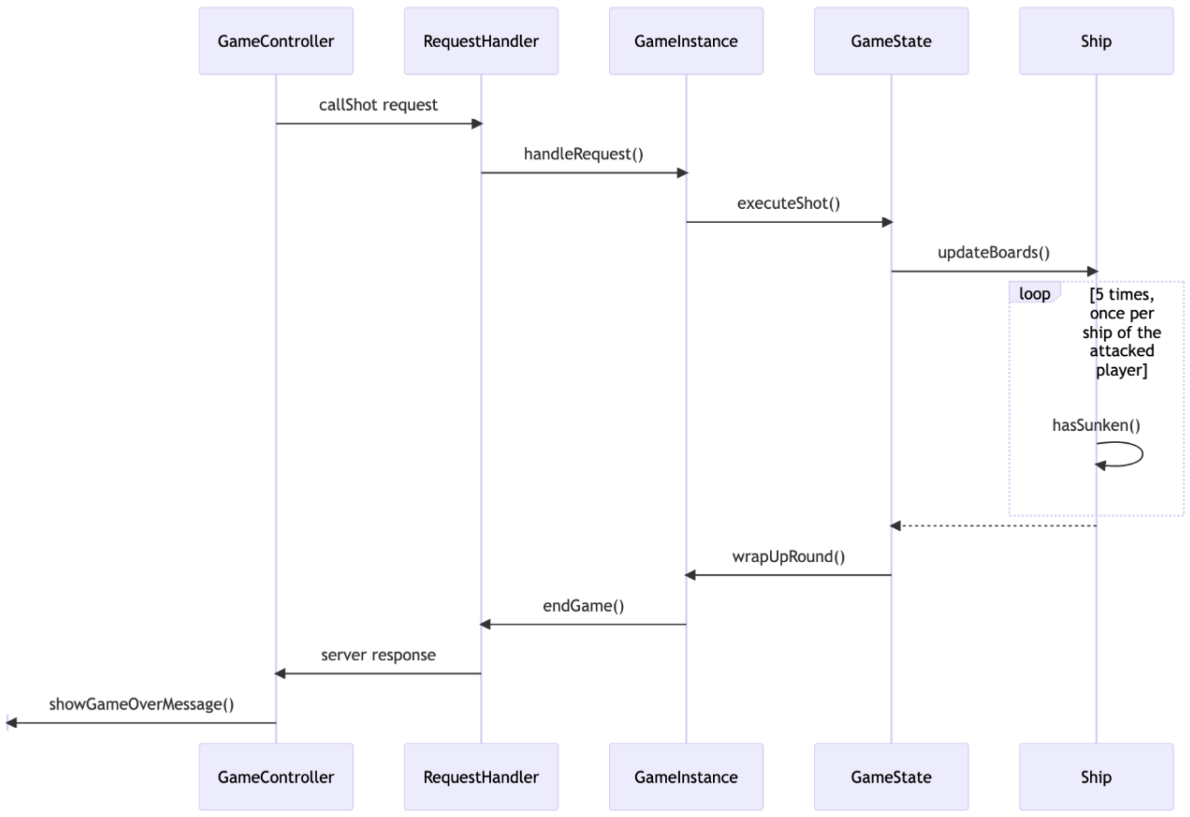
FREQ-9: Calling a shot
FREQ-10: Keeping the scoreboard updated
FREQ-11: Encountering a hit
FREQ-13: Sinking a ship
FREQ-14: Winning the game
FREQ-19: Performing validity checks on every game state
FREQ-20: Game GUI
FREQ-22: Adhering to appropriate external interfaces

**The scenarios which are related to this sequence are:**

SCN-6: Ending a game

**Scenario Narration**:

*The Server receives a request from a player to call a shot. The shot is executed by the game instance as the game state is updated. When the game state is updated, each ship is checked for whether or not it has sunk. When the game state wraps up the round, it is informed that each ship has sunk. When it tells the game instance to wrap up the round, it also indicates that the game is over. The game instance ends the game. The server tells the game controller to show the game over message to the client.*


GameController
RequestHandler
GameInstance
GameState
Ship
callShot request
handleRequest()
executeShot()
updateBoards()
loop
[5 times, once per ship of the attacked player]
hasSunken()
wrapUpRound()
endGame()
server response
showGameOverMessage()
GameController
RequestHandler
GameInstance
GameState
Ship

# 4. Interface Modeling

## 4.1 Interface Client-Server

**Purpose:** The purpose of this interface is the communication of the two clients with the server. The interface allows client requests and server answers, game event updates and error communication.

**Communication between:** Client and Server, initiated by Client

**Protocol:** TCP

**Communication modes:** Request-Response and Broadcast of game event

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

### 4.1.1 JoinGameRequest

**Purpose:** Request to join a game as a player

**Direction:** Client to Server

**Content:**

- type: “join_game” (required)
- player_id: integer (required)
- player_name: string (required)

**Format:** as JSON string

**Example:**

```
{
      "type": "join_game",
      "player_id": 13634,
      "player_name": "Bob"
}
```

**Expected response:** GameEventResponse or ErrorResponse

### 4.1.2 StartGameRequest

**Purpose:** Request to say the player is ready to play and the game can be started

**Direction:** Client to Server

**Content:**

- type: “start_game” (required)
- player_id: integer (required)
- ships_placment: array of ships position and orientation (required)

**Format:** as JSON string

**Example:**

```
{
        "type": "start_game",
        "player_id": 13634,
        "ships_placment":
        [
                {
                        "ship_id": 624321,
                        "length": 3,
                        "position": {"x": 1, "y": 4},
                        "orientation": "v"
                },
                …
        ]
}
```

**Expected response:** GameEventResponse or ErrorResponse

### 4.1.3 CallShotRequest

**Purpose:** Request to make a shot

**Direction:** Client to Server

**Content:**

- type: “call_shot” (required)
- player_id: integer (required)
- position: x, y coordinate of the shot (required)

**Format:** as JSON string

**Example:**

```
{
      "type": "call_shot",
      "player_id": 13634,
      "position": {"x": 4, "y": 8}
}
```

**Expected response:** GameEventResponse or ErrorResponse

### 4.1.4 SendEmoteRequest

**Purpose:** Request to send a emote

**Direction:** Client to Server

**Content:**

- type: “send_emote” (required)
- player_id: integer (required)
- emote: string (required)

**Format:** as JSON string

**Example:**

```
{
      "type": "send_emote",
      "player_id": 13634,
      "emote": "💣"}
}
```

**Expected response:** EmoteEventResponse or ErrorResponse

### 4.1.5 PlayAgainRequest

**Purpose:** Request to play a new round after the game ends

**Direction:** Client to Server

**Content:**

- type: “play_again” (required)
- player_id: integer (required)

**Format:** as JSON string

**Example:**
```
{
        "type": "play_again",
        "player_id": 13634
}
```

**Expected response:** GameEventResponse or ErrorResponse

### 4.1.6 QuitGameRequest

**Purpose:** Request to make when the play quits the game

**Direction:** Client to Server

**Content:**

- type: “quit_game” (required)
- player_id: integer (required)

**Format:** as JSON string

**Example:**
```
{
        "type": "quit_game",
        "player_id": 13634
}
```

**Expected response:** GameEventResponse

### 4.1.7 StartGameResponse

**Purpose:** Response to start the game

**Direction:** Server to Client

**Content:**

- type: “start_game” (required)
- starting_player_id: integer (required)

**Format:** as JSON string

**Example:**
```
{
        "type": "start_game"
        "strating_player_id": 13634
}
```

**Expected response:** none

### 4.1.8 GameEventResponse

**Purpose:** Response to notify that the game state has changed

**Direction:** Server to Client

**Content:**

- type: “game_event” (required)
- sequences_number: integer (required)
- event_info: info object (required)

**Format:** as JSON string

**Example:**
```
{
      "type": "game_event",
      "sequences_number": 45,
      "event_info":
      {
            "event_type": "new_shot",
            "player_id": 13634,
            "position": {"x": 4, "y": 0},
            "ship_hit": False,
            "ship_sunk": False
      }
}
```

**Expected response:** none

### 4.1.9 EmoteEventResponse

**Purpose:** Response to notify that a player has send a emote

**Direction:** Server to Client

**Content:**

- type: “emote_event” (required)
- sequences_number: integer (required)
- player_id: integer (required)
- emote: string (required)

**Format:** as JSON string

**Example:**

```
{
	"type": "emote_event",
	"sequences_number": 12,
	"player_id": 13634,
	"emote": "💣"
}
```

**Expected response:** none

### 4.1.10 ErrorResponse

**Purpose:** Response to communication an error

**Direction:** Server to Client

**Content:**

- type: “error” (required)
- error_id: integer (required)
- error_message: string (required)

**Format:** as JSON string

**Example:**

```
{
	"type": "error",
	"error_id": 3,
	"error_message": "invalid move"
}
```

**Expected response:** none