university of koblenz
Computer Science

Institute for Software Technology

MazeGame 2

University of Koblenz

Faculty 4: Computer Science

Institute for Software Technology

Programming Techniques and Software Design (PTSD)

Summer Term 2025

Document Version 2025-07-03

Table of Contents

1 Introduction

The course project is a network based game. We will provide the game server that can also be run locally on your machines. Your task is design, implementation, testing, and documentation of the game client. You will have to complete this work as a team.

2 About this Document

This document contains the description of the course project you have to conduct in order to pass the exam along with information about the requirements to fulfill and the modalities how to submit your project artifacts.

You will find the latest version of this document on the course website in OLAT in the folder `exam-project` of the Downloads section.

PLEASE READ EVERYTHING IN THIS DOCUMENT CAREFULLY. If you happen to get lost, or if you don't understand what the explanations mean, please ASK QUESTIONS! Use your Mattemost channel and/or your team sessions.

If you find errors, unclear or contradicting information, please report them to the organizers of the lecture so that the problems can be solved and the project description can be improved.

Throughout this document we use the key words "MUST", "MUST NOT", "SHOULD", "SHOULD NOT" and "MAY" as described in RFC 2119[1].

1 https://tools.ietf.org/html/rfc2119

3 Terms and Definitions

Client
The **client** is the part of the game software that runs on a player's computer and enables a user to issue actions in the game. It connects to a game server.

Connect
A client **connects** to a server to participate in a game. After connecting, the client has to provide a nickname for the player to participate actively.

Disconnect
A client **disconnects** from a server in order to no longer participate in a game.

Game
A **game** is coordinated by a game server. Multiple clients can participate in a game. The number of simultaneous connections can be limited by the server. Each game has a maze with static elements (accessible fields, non-accessible fields, walls) and dynamic elements (players, baits).

Login
See “Connect”.

Logout
See “Disconnect”.

Player
A **player** is a participant in the game and controlled by the user or a robot strategy via the client.

Position
A **position** is a two dimensional coordinate in the maze given as tuple (X,Y).

Protocol
The **protocol** describes the format and sequence of messages exchanged between server and client and between client and server.

Score
A **score** is an integer value associated with a player. Better players have higher scores.

Server
A game **server** coordinates a game. Clients connect to a server to join a game.

Source Code
Unless otherwise specified, the term “**source code**” refers to both, source code of the application and source code of automated tests.

User
A **user** is a natural person (human) that interacts with the computer on which the client is running.

4 Task Description

You will have to provide a fully-functional client implementation. Of course, you won’t only submit a compiled binary, instead you will provide all relevant artifacts of your development process.

We expect you to apply and demonstrate what you have learned during the lecture and the assignments.

In particular, you will provide the following:

- **UML models** created during the design of your client, e.g. class diagrams, state machines, and sequence/activity diagrams
- **Java source code** of your implementation which will at least consist of the following components (we will provide an architecture outline):
  - **Network** component that handles all communication between the client and the server
  - **Game status** component that holds all game specific logic and data
  - **Command handler** component that updates the data in the game status component according to received server messages and other events from the client's graphical user interface
  - **Graphical user interface** that enables users to play the the game (via keyboard/mouse), to display the maze, players, scores, and baits
  - **Robot Strategy** component that enables your client to play automatically (we also plan to conduct a "MazeGame contest" where the robots of the various teams can compete against the others)
- **JavaDoc** for all packages, classes, fields, methods, interfaces, enums, etc. you have created
- **JUnit test code** for the public interfaces of the *GameStatusModel* and the *CommandHandler* components Your automated tests should cover the complete implementation. You can use the coverage metrics tool in Eclipse to determine which parts of your implementation are covered by the test cases. If you submit untested parts, please give a rationale why you could not test those parts.
- A **Project report** that must document not only the software but should also provide answers to the following questions:
  - How did your software development process look like?
  - Which technical, organizational or team related issues and problems occurred during the development?
  - Which part(s) of the task have not been accomplished (if there are any) and why have they not been accomplished?
  - How can we configure / run your software?

- What else should we know about your submission?
- How was the workload distributed in your team?
  (a record of topics and working hours per team member)

5 Organizational Details

5.1 GitLab, Language, Naming

You MUST use your GitLab project `ptsd-<year>-<teamname>-exam-project`. Add all team members and all supervisors to that project. The supervisors need *Owner* access.

You MUST maintain a meaningful `README.md` in the top-level directory of your repository that allows us to quickly understand what we will find inside the repository and where we will find it.

Your UML model labels, Java source code, JavaDoc and JUnit test code MUST be written in English. Your `README.md(s)` and your Project Report MUST be either English or German.

You MUST use *meaningful* git commit messages.

Each team member MUST commit her or his contributions with her or his own user account - i.e., you MUST NOT delegate the "git stuff" to a single expert in the team.

You MUST NOT use “git force push” (e.g. by `git -f push`) which would modify the git commit history on the GitLab server.

You SHOULD use the GitLab project’s issue tracker to organize and document and discuss and track issues and features to be developed (other ways of communication are also allowed, of course).

You MUST record the approximate time spent by each team member for each task so that the workload distribution is transparent.

You MAY use the GitLab wiki in your project for the project report and any other documentation. Wiki usage is optional, but a strong recommendation.

5.2 Submission of your solution

To submit your project for review, you MUST create a GitLab branch named `submission`. Please put all artifacts that you intend to submit into that branch.

6 Game Description

Several players move in a simple rectangular maze. The maze contains certain objects ("baits") that have to be picked up by the players in order to score points. Some of the objects are visible, some are invisible.

Movement in the maze is done by turning on a field to change the direction of view and by moving forward. A player can only move forward one field at a time.

The players are lone fighters. If two players meet on the same field of the maze, they are both displaced by magic, almost instantaneous, teleportation to another place in the maze. From there, they have to continue searching separately for the scattered treasures.

Teleportation also happens if a player plunges into the water or tries to step outside the maze.

The following baits (treasures and traps) are spread throughout the maze:

- `gem` a valuable diamond (+314 points)
- `coffee` a cup of coffee (+42 points)
- food a voucher for a free dish in the university restaurant (+13 points)
- trap a very bad trap; the player is teleported and has to write the bachelor thesis with EDLIN[2] under MS-DOS 3.1 and print it out on a typewriter printer (-128 points)

The MazeGame is a network game. A server provides the playing field and the necessary information on the course of the game. Each of the participating players connects to the server via a client program. All logic, player movements, baits, and status updates and are handled by the server. The server informs each client about updates by sending messages that contain all necessary information so that a client can create and maintain a representation of the complete game status.

2 https://en.wikipedia.org/wiki/Edlin

7 MazeGame Client Usecases

A *MazeGame Client* (MGC) can be active for one or more players. Each player has an identification code (unique number assigned by the server), and a nickname chosen by the user. Each player is Player at a specific position `(x,y)` in the maze, looks in one of the four directions `(n,e,s,w)` and has an individual score. Initially, after joining the game, the score is 0.

Every client knows the data mentioned above from all other players and can represent this accordingly. A client also maintains a list of known visible items and their location in the maze.

The players can move by turning to the left or to the right (keeping the position) and by stepping one step forward in the current view direction

The client has to provide an option to let the user control the player manually, or let a "robot strategy" automatically decide what the next move should be. When the robot strategy controls the player, user control has to be disabled.

Since it's impossible to die in this game, only the fastest players can get the highest scores. Very sophisticated teams could also work on collaborating robots that try to hinder other players in order to support a specific player.

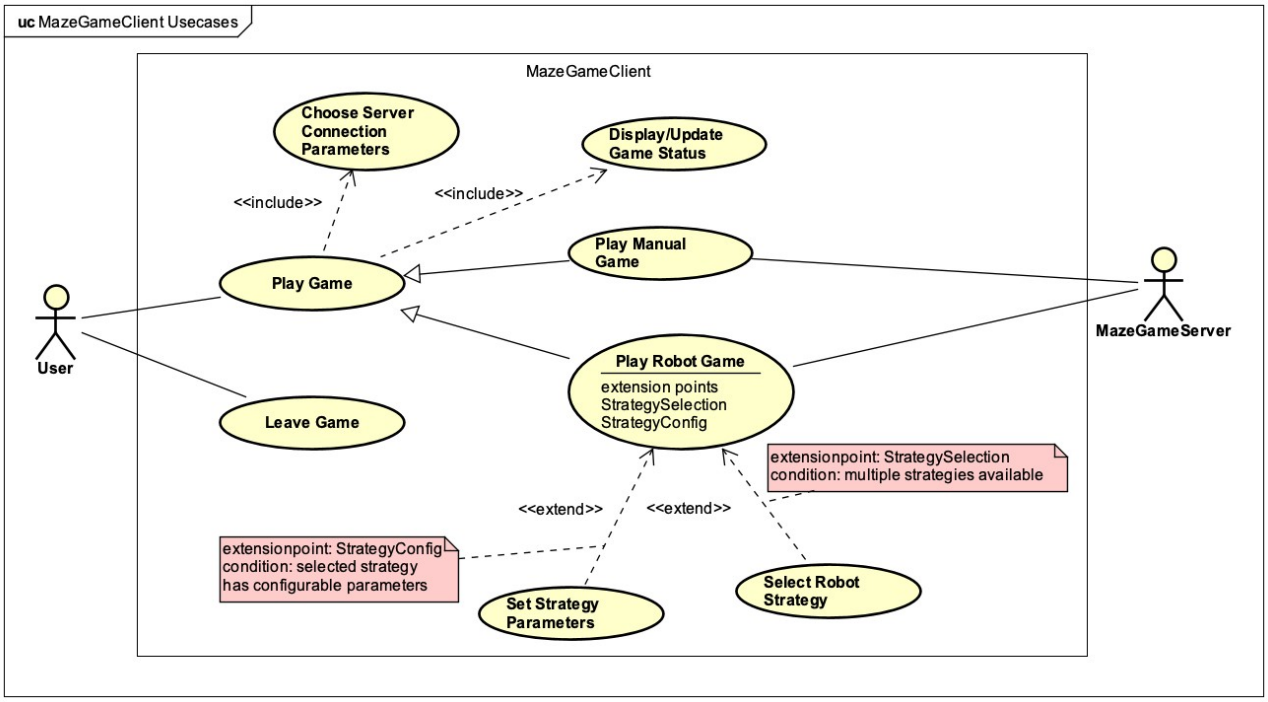


Figure 1: Use Case Diagram of the MazeGame Client

Figure 1 shows a Use Case Diagram of the MazeGame Client. You will find a more comprehensive description and additional details of the relevant use cases in a separate document. Of course, you are free to identify and implement additional use cases.

The Astah model `MazeGameClient.asta` contains use case descriptions, and we provide an exported document (`MazeGameClient.pdf`) with basic documentation of the model elements.

8 Getting Started

This section gives some hints on how to get started. The approach is bound to the proposed architecture.

The model file `MazeGameClient.asta` with all diagrams is available in the download area. You may use this file as a baseline for your model.

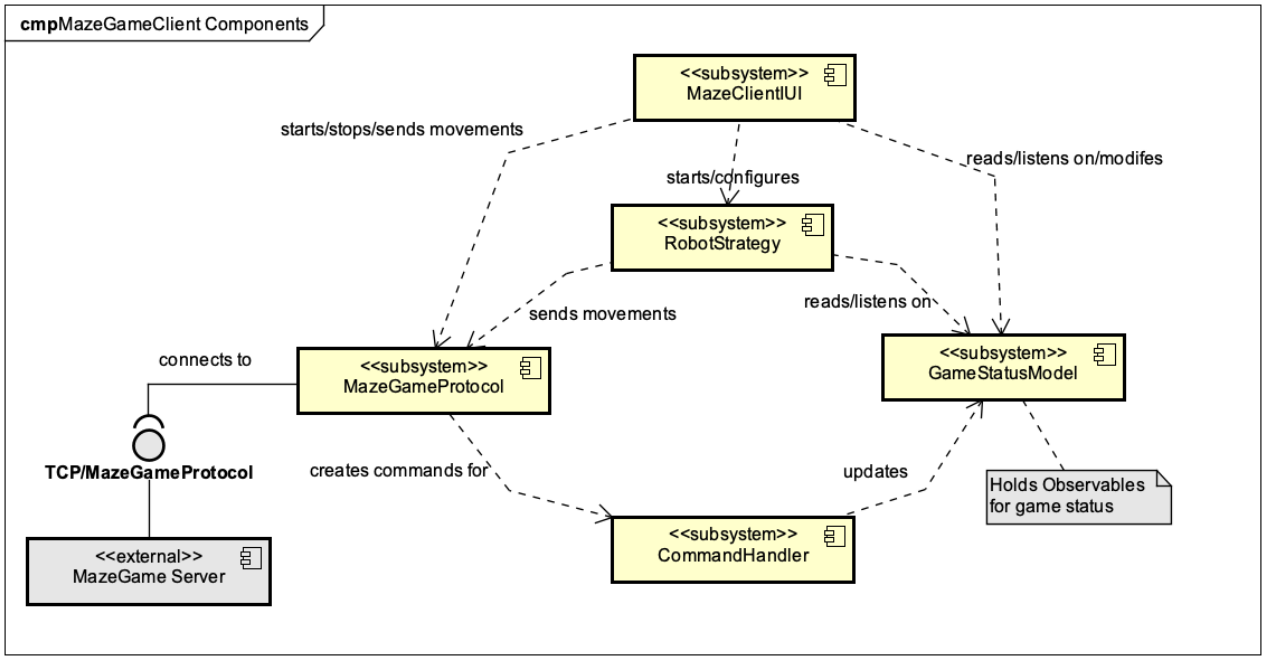


Figure 2: Top-Level Components the MazeGame Client

Your solution MUST use the Java Module System. This means that you MUST create five (5) Java modules. Each module requires a separate Eclipse project.

You may also begin with a single module for all subsystems that can be cut into 5 parts later, but you can expect more effort/trouble to separate the monolithic solution than to start with modules right from the beginning.

To make it very clear and explicit: You SHOULD NOT[3] implement the MazeGame Server. It is already provided as executable JAR file with the task. You have to start this server (manually, in a terminal/console/command line window), and your client has to connect to the server via TCP.

At first, create five Java projects in Eclipse, one for each of the subsystems (yellow boxes in the figure above). Please choose meaningful project and module names. Each project realizes one of the subsystems. Make sure that you pick meaningful package names as well, e.g. `de.uni_koblenz.ptsd.<teamname>.mazeclient.ui` as base package for the user interface subsystem.

3 SHOULD NOT means that you MAY implement such a server if you have a lot of time left and you are curious on how to develop such a software. There are many different aspects to deal with when you want to realize a reliable, fast, and robust server. However, your MazeGame Client will be tested against the MazeGame Server provided with the task.

When implementing the subsystems, we suggest to start with the GameStatusModel. You should have designed an initial class diagram with assignment 07. Please use **observable values/lists/maps from the JavaFX library** to store the game status data since those will be used by the UI components. Refer to the JavaFX lectures/demos to learn about observables.

Ideally, each subsystem provides it's services via one (or more) exported interfaces or Facade classes. You may use the ServiceLoader of the module system to minimize the dependencies between the components. You may also start with closely coupled modules (i.e. all classes/interfaces in a module are visible to the using modules). This can be easier at the beginning but creates a lot of dependencies that are hard to remove later.

After the GameStatusModel is complete, we suggest to go ahead with the MazeGameProtocol subsystem. It's responsible to connect to the server and to send and receive messages (the CSV strings specified in the protocol documentation).

To update the GameStatusModel, we recommend to use the Command pattern (see software design patterns). The command classes can go into the CommandHandler subsystem. Every incoming server message shall be converted into a specific command instance. To *create* the command objects is the task of the MazeGameProtocol, to *execute* the commands is the responsibility of the CommandHandler. Executing the commands means to update the GameStatusModel values, which, in turn, notify their change listeners.

Example: The server sends a PSCO;17;42 message telling your client that the score of player #17 is now 42. Your protocol component constructs a PlayerScoreCommand from the message string. This command object contains all relevant data. When the command is executed by the CommandHandler, it has to update the GameStatusModel such that the score property of the respective player object receives the new value. Similarly, all the other messages can be dealt with.

In your implementation, you will have to implement/deal with multiple Threads. One of the Threads that is created automatically is the *user interface thread* of JavaFX, another one will be the reader for the input messages from the MazeGame server in your MazeGameProtocol component. Most probably, you'll need more than those two threads, e.g., to execute the commands in the CommandHandler subsystem. Please remember that it's mandatory to properly synchronize the concurrently accessed data structures. In case of changes to observable properties, you have to take care that those changes are only allowed from the UI thread. This means that you have to execute such changes (i.e. the command objects) via a Runnable submitted to `Platform.runLater(…)`. Please look into the JFXTrafficLight demo for an example.

We strongly suggest to write JUnit tests to verify that the various commands modify the status model correctly. Doing so means also that you can work in parallel in your team. People responsible for the command handler can work (relatively) independent from the ones who deal with the protocol. Of course, you may also develop everything as a team without splitting the responsibilities.

The user interface component MUST be implemented using JavaFX. Following the use cases in the task description, there are (at least) three parts: connecting/disconnecting to/from a game, displaying the maze, and displaying the score table.

We suggest to begin with the maze display and to use a fixed server address, port, and nickname for the connection. You have to tell the MazeGameProtocol subsystem to connect, and after that, all other modifications of the user interface shall be triggered by changes of the observable values in the GameStatusModel (which, in turn, are updated by executing commands).

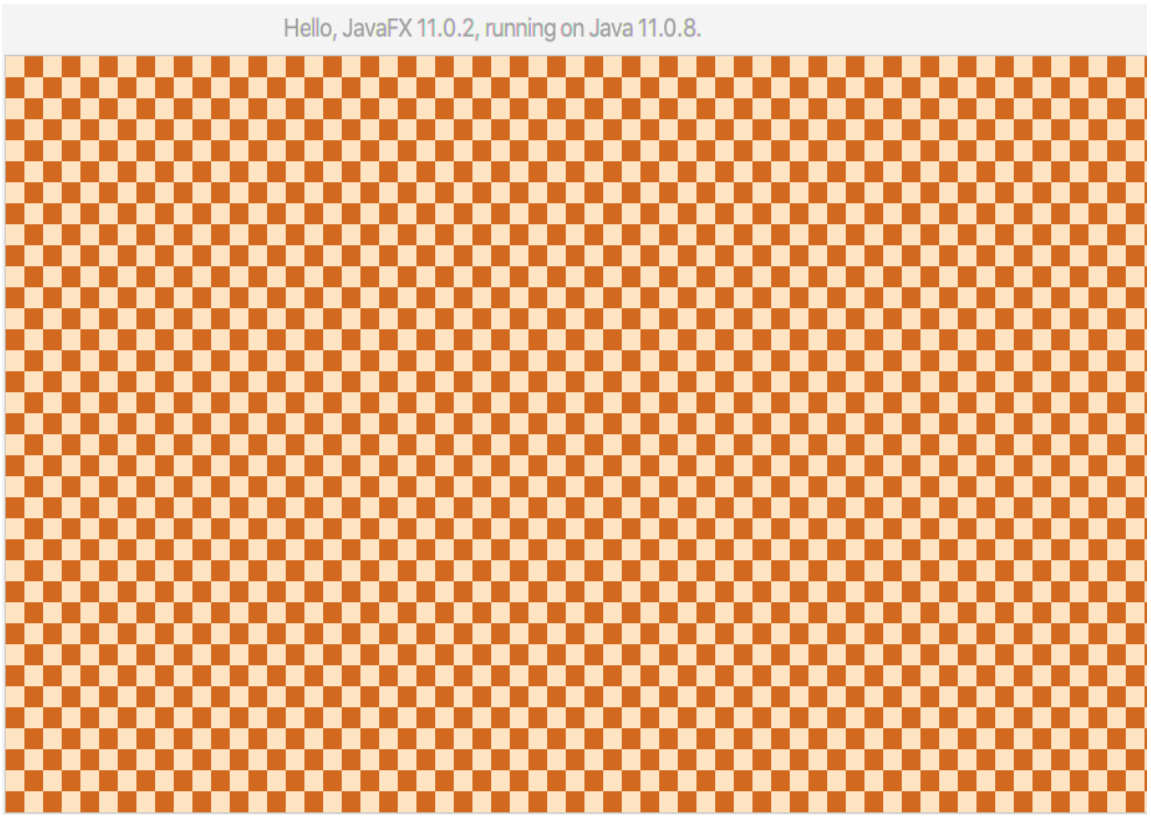
To display the maze (without players and baits), we suggest to use a JavaFX Group node. You have to add Rectangle nodes for the maze fields to that Group. For the players and baits, you can use another Group node. The two groups can be combined by a StackPane so that the maze group is in the background and the player/bait group is in the foreground. Please refer to the scene graph descriptions in the lecture and the JavaFX documentation to gain detailed knowledge.

To get an idea on how the score table can be realized, we suggest to review the JFXPersonList example application. There you can learn about how to realize a table view together with the observable values that are used to store the data.

You may also start UI implementation without protocol and command components. That means, you can refer to the GameStatusModel and execute the changes “manually”, e.g., by adding some buttons that modify the model with simulated values.

When you start your project, you probably need to experiment a lot to figure out how things work, especially regarding the user interface. Please use SEPARATE projects for the experiments. Otherwise, your code gets bloated with things that you later have to remove.

To get going, we suggest to write a simple JavaFX program that displays a checkerboard of arbitrary size inside a scroll pane. Arbitrary size means, width and height can be constant values in your program. This could look like the following figure:



When your user interface can display all information of a real game correctly, you can add the few things needed to control the own player, e.g. by receiving keyboard events and telling the protocol subsystem to send the respective turn or step command to the server.

The last component that you should deal with is the RobotStrategy. Whenever the client receives a `RDY.` message from the server, the strategy shall compute the next move (turn or step) and ask the protocol system to send the command. To try if the basic commanding works, you might want to send random turns or steps after each ready message (this is exactly the behavior of the dummy players that move around on the server).

If you don’t know at all how to compute the next move, we suggest to look at descriptions of the *Dijkstra algorithm* and/or the *A* algorithm*. Both algorithms compute the shortest path to nodes in a graph based on a distance measure. The distance measure to use in your RobotStrategy can be the number of turns/steps to reach the desired position. When a target node can be reached, the remaining problem is to decide which command has to be sent next. Your strategy should take into account that valuable baits may be taken by other players while your robot is on it’s way. It doesn’t make sense to pick a target field and keep going there when the bait has already been collected by another player in the meantime…

9 Requirement List

We expect you to fulfill the following requirements. Additionally, the requirement list is helpful during the design of your client implementation, because you get an impression of what has to be implemented, and it might remind you of certain aspects. The list is not complete, there are more requirements, of course.

The initial requirement list contains mostly high-level requirements, far from the level of detail of a full specification. This is to enable the teams to develop their own ideas. You may extend the requirements, but you must not drop one of the following entries.

The teams have to document their final requirements (and if they have been fulfilled) in the project manual.

If teams have clarifying questions, it can happen, that we extend or modify the common requirement list during the course project.

Client Software

CL-01 The MGC ***MUST*** provide users with the ability to participate in a game using a graphical user interface (MGC UI).

CL-02 If connected to a server, the MGC ***SHOULD NOT*** terminate itself.

CL-03 The MGC ***MUST*** use a non empty nick name to connect to a server.

CL-04 If the player can't move one step forward, the MGC ***SHOULD NOT*** issue a STEP message.

CL-05 The MGC ***MUST*** run at least two threads (UI and network).

CL-06 The MGC ***MUST*** fully comply to the protocol specification in this document.

CL-07 The MGC ***MUST NOT*** crash on faulty protocol messages or broken network links, i.e. the client must be tolerant towards erroneous or unexpected behavior of the connected server.

CL-08 The public interfaces of the MGC components ***MUST*** be fully documented in JavaDoc.

CL-09 The MGC ***MUST NOT*** send invalid protocol messages.

CL-10 The MGC ***SHOULD*** provide a way to set the server connection via command line parameters.

CL-11 If the MGC uses command line parameters, the MGC ***MUST*** provide a way to explain command line parameters (e.g. by `--help`)

User Interface

UI-01 The MGC UI ***MUST*** provide a way to configure server connection parameters.

UI-02 The MGC UI ***MUST*** enable the user to connect to the server.

UI-03 The MGC UI ***MUST*** display the current connection state and the parameters of the server to connect with or connected with.

UI-04 The MGC UI ***MUST*** enable the user to disconnect from the server.

UI-05 The MGC UI ***MUST*** enable the user to issue TURN command to turn the player while in a game.

UI-06 The MGC UI ***MUST*** enable the user to issue STEP commands to move the player while in a game.

UI-07 The MGC UI ***SHOULD*** provide meaningful error messages (e.g., reason for not being able to connect to the server).

UI-08 The MGC UI ***MUST*** display the maze with all accessible and non-accessible fields.

UI-09 The MGC UI ***MUST*** update the displayed maze according to position changes of baits and players.

UI-10 The MGC UI ***MUST*** display all players of the game along with their nick names.

UI-11 The MGC UI ***MUST*** provide a way to display the current scores of all players.

UI-12 The MGC UI ***SHOULD*** inform the user about exceptions in the program flow.

UI-13 The MGC UI ***MUST*** be implemented using JavaFX.

UI-14 The MGC UI ***MUST*** be able to display mazes of sizes up to 160x90.

UI-15 The MGC UI ***SHOULD*** be able to scale or zoom the maze display.

UI-16 The MGC UI ***SHOULD*** be able to display different mazes of different sizes without restarting the application.

UI-17 The MGC UI ***MUST*** be implemented as a desktop application using the JavaFX framework.

Robot Strategy

RS-01 The MGC ***MUST*** provide the user with the ability to play a game with an automatic robot strategy.

RS-02 The robot strategy ***MUST*** take decisions on the next move of the player based on the current game status.

Source Code

SC-01 The MGC source code ***MUST*** be comprehensively documented by meaningful JavaDoc comments.

SC-02 The MGC source code ***MUST*** comply to the communicated formatting rules.

SC-03 The set of applied formatting rules ***SHOULD*** be defined in the project report.

SC-04 The source code of the MGC ***MUST*** follow the Java naming conventions explained in the lecture.

SC-05 The MGC ***MUST*** be structured into at least 5 subsystems implemented as Java modules according to the top-level component diagram in Figure 2. The responsibilities of the components are explained in the UML model provided with the task.

SC-06 The MGC source code ***MUST*** be analyzed by a quality metrics tool, e.g., the *SonarLint* tool presented in the tutorials.

SC-07 The source code ***MUST*** be refactored to remove the detected quality flaws. If some findings can't be fixed, a justification has to be provided.

UML Models

MO-01 The project team ***MUST*** provide a single UML model that contains all elements used in the diagrams.

MO-02 The UML model ***MUST*** contain all classes of your source code (without test code).

MO-03 The UML model ***MUST*** contain a UML state machine diagram with the different states and state transitions of the MGC (connected, etc.).

10 Protocol Specification

The communication between server and client happens bidirectionally via a TCP connection. The messages are sent in CSV format, one message per line. The message structure is described by the grammar given below. In the following, the protocol is introduced by an exemplary procedure. The different message types are described in detail.

The server recognizes which client has sent a message by the client's network address.

After successful connection, the server sends its own ID and the version number of the protocol in use (currently `MSRV;2`) to the client.

Then, the client has to login into the game on the server. To log in, the client sends a `HELO` message with its nickname. This message has to be sent within 30 seconds after the connection was established. Otherwise an `INFO` message notifies about timeout and the server terminates the connection.

The server checks, whether the maximum number of clients has already been reached, and if the nickname is correct and not yet in use. If everything is fine, a `WELC` message is sent back to the client as confirmation. After that, all other connected clients are informed about the arrival of the new player (`JOIN`), its position (`PPOS`) and score (`PSCO`). The player itself gets these information only after sending the `MAZ?` command and after receiving the maze (see below).

If the client is not allowed to join the game, it will be rejected by an according `INFO` message containing an error code.

With the `WELC` message, the client receives its numerical player ID from the server. With this ID, the client can be identified by itself and all other players.

After successful login, the client has to request the maze data from the server. To do so, it sends a `MAZ?` to the server, which responds with a `MAZE`. This message contains the entire game field description in a compact format. After that, the player is considered active in the game and receives status updates. Hence, the server sends information about the visible objects spread across the maze (`BPOS`), the other players (`JOIN`, `PPOS`) and their positions and their scores (`PSCO`).

If a client wants to leave the game, it sends a `BYE!`, and the server acknowledges with a `QUIT` message. The server sends a corresponding `LEAV` message to all other clients, to inform them about the withdrawing player.

If a client wants to move its player, it sends `STEP` or `TURN` to the server. The server responds either with `PPOS` (new position or viewing direction, respectively) or with `INFO` and a corresponding error message if the requested action was not possible.

The computation of the destination field of a `STEP` is done on the server, there's no need to compute it on your own. Each and every game status change results in a message sent from the server to the client, hence, you don't have do move the players.

The client is allowed to send another `STEP` or `TURN` only after receiving a `RDY.` from the server. After sending a command, after small delay, the client receives a `RDY.` again. Usually, the client

will receive other other messages before the `RDY.` since other players can move, join, or leave the game, baits can be collected or appear, and player scores may change.

Teleportation of players that collide on a game field is handled by `PPOS` messages. Teleportation also takes place when a player collects a Trap or hits a hidden Trap.

If a player enters a field with a collectible object, the object disappears from the game (message `BPOS` to all clients) and the player receives a score update (message `PSCO` to all clients).

When the server is shut down, all clients are notified via a `TERM` message.

10.1 Protocol Syntax

This section defines the grammar for the protocol in Extended Backus-Naur Form (EBNF). The start symbol is `Message`. **The individual messages are sent line by line as CSV strings.** Each line corresponds to a message. The only exception is the `MAZE` message that consists of more than one line. The first value of a message is a unique message identifier. The respective arguments follow, **separated by semicolons**. The line terminator and the separators are not part of the EBNF.

Messages

```
Message ::= ClientMessage | ServerMessage

ClientMessage ::= Bye | Hello | MazeQuery | Step | Turn
ServerMessage ::= ServerVersion | BaitPos | Maze | PlayerPos | PlayerScore |
                  Welcome | Join | Leave | Info | Terminate | Ready | Quit
```

Coordinates

```
Coord ::= Integer Integer
```

- The first value is x direction (horizontal), the second value is y direction (vertical). The following holds:
- `0 <= x < Width`
- `0 <= y < Height`
- The line with `y = 0` is in the north
- The column with `x = 0` is in the west.

ID of a player

```
PlayerID ::= Integer
```

Nickname of a Player

```
Nickname ::= Letter { Letter | Digit }
```

Viewing direction

```
ViewDir ::= "n" | "e" | "s" | "w"
```

- The viewing direction is always parallel to one of the two axes.
- The letters denote the individual cardinal points (North, East, South, West)

Direction of turns

```
TurnDir ::= "r" | "l"
```

Types of objects

```
BaitType ::= "gem" | "food" | "coffee" | "trap"
```

10.2 Message Descriptions

Message `BPOS` (Bait Position)

```
BaitPos ::= "BPOS" Coord BaitType Event
Event   ::= "app" | "van"
```

Direction: Server > Client

Description: The server transmits the position of an object. The coordinates are in `Coord`. The type of the object is encoded in `BaitType` (see list of object types in "Idea of the Game" above). `Event` describes what happened to the object:

- `app` (appear): The object of type `BaitType` spawned at position `Coord`
- `van` (vanish): The object of type `BaitType` disappeared at position `Coord`

Message `HELO` (Hello)

```
Hello ::= "HELO" NickName
```

Direction: Client > Server

Description: The client wants to join a game and sends the desired nickname. The server responds either with `WELC` or with `INFO`

Message `JOIN` (Player Joins Game)

```
Join ::= "JOIN" PlayerID NickName
```

Direction: Server > Client

Description: The server announces a new player to already known clients and already known players to new clients.

Message `LEAV` (Player Leaves Game)

```
Leave ::= "LEAV" PlayerID
```

Direction: Server > Client

Description: The server announces the withdrawal of a player.

Message `INFO` (Information)

```
Info     ::= "INFO" InfoCode
InfoCode ::= Integer
```

Direction: Server > Client

Description: The server informs about an error or another exception. The error code `InfoCode` has one of the following values:

Temporary errors (can be fixed by modified parameters or by sending the message again later):

- 450 : Wrong parameter value
- 451 : Login not possible (too many clients)
- 452 : Login not possible (wrong or already used nickname)
- 453 : Wall ahead (`STEP` not possible)
- 454 : An attempt was made, to do a `STEP` or a `TURN` without prior approval by a `RDY`.
- 455 : An attempt was made, to send another `HELO` after a successful login (Already logged in).
- 456 : An attempt was made, to send a `STEP` or a `TURN` without a prior successful login (Not logged in).
- 457 : Client has wasted too much time during login (Login Timeout).

Permanent errors (can't be fixed):

- 500 : Unknown command
- 501 : Wrong number of parameters

Message `MAZ?` (Maze Query)

```
MazeQuery ::= "MAZ?"
```

Direction: Client > Server

Description: The client asks the server for maze data. The server responds with `MAZE` and subsequent `JOIN`, `PSCO`, `PPOS` and `BPOS` messages, to inform about the current game state.

Message `MAZE` (Maze)

```
Maze   ::= "MAZE" Width Height // followed by HEIGHT lines with game field data
Width  ::= Integer
Height ::= Integer
```

Direction: Server > Client

Description: The server informs about the structure of the maze. First, it sends the `Width` and `Height` of the rectangular game field. The following lines with game field data contain one character for each field of the maze. The fields are encoded as follows:

- `?` (Question Mark): Unknown, non-accessible field (may also occur inside the maze)
- `#` (Hash): Stone field; an impenetrable wall
- `~` (Tilde): Water field
- `.` (Period): Free accessible path field

Example (real maze will look differently!):

```
MAZE;20;10
#.......~~~??????
###...############
#...............#
#.###.#.~~~.##.#.#
#...#??.~~~.#..###
#.#.........#..#.#
#.#########.#.##.#
#.....~~~~~......#
#######~~~########
```

Message `PP0S` (Player Position)

```
PlayerPos ::= "PPOS" PlayerId Coord ViewDir Reason
   Reason ::= "tel" | "app" | "van" | "mov" | "trn"
```

Direction: Server > Client

Description: The server transmits the position of a player. `PlayerId` describes the ID of the player, whose position is transmitted. `Coord` is the coordinate, `ViewDir` the current viewing direction of the player. `Reason` describes one of the following actions, that happened to the player identified by `PlayerId`:

- `tel` (teleport): The player has been teleported to the given position
- `app` (appear): The player spawned at the given position (after JOIN)
- `van` (vanish): The player disappeared from the given position (before LEAVE)
- `mov` (move): The player has moved to the given position
- `trn` (turn): The player has changed the viewing direction to the given direction

Message `PSCO` (Player Score)

```
PlayerScore ::= "PSCO" PlayerID Score
      Score ::= Integer
```

Direction: Server > Client

Description: The server informs about the current scores of the player with ID `PlayerID`.

Message `STEP` (Step)

```
Step ::= "STEP"
```

Direction: Client > Server

Description: The client wants to move one step into the viewing direction.

Message `TURN` (Turn)

```
Turn ::= "TURN" TurnDir
```

Direction: Client > Server

Description: The client wants to change its viewing direction. It can either turn 90 degrees right or 90 degrees left.

Message `RDY.` (Ready)

```
Ready ::= "RDY."
```

Direction: Server > Client

Description: The client is informed that the server is ready to accept a `STEP` or `TURN` message.

Message `BYE!` (Goodbye!)

```
Bye ::= "BYE!"
```

Direction: Client > Server

Description: The client wants to leave the game, the server responds with `QUIT`. Before the `QUIT` message arrives, other messages can be sent by the server.

Message `QUIT` (Quit Game)

```
Quit ::= "QUIT"
```

Direction: Server > Client

Description: The server informs the client about the successful logout. The client may quit. The Server may also send a `QUIT` message without a preceding `BYE!` when a client is removed for other reasons. `QUIT` messages can occur at any time.

Message `MSRV` (Maze Server Version)

```
ServerVersion ::= "MSRV" Integer
```

Direction: Server > Client

Description: After a client connects, the server identifies itself with `MSRV` along with the version number of the protocol in use.

Message `TERM` (Terminate Server)

```
Terminate ::= "TERM"
```

Direction: Server > Client

Description: The server will terminate itself (and therefore finish the game) soon. `TERM` messages can occur at any time.

Message `WELC` (Welcome)

```
Welcome ::= "WELC" PlayerID
```

Direction: Server > Client

Description: The server informs about a successful login and informs the client about the assigned player ID.

11 Final Words

Last but not least, we’d like to stress our permanent offer to communicate with us. Usually, you’ll get answers within minutes, even at weekends. Asking questions is not a shame, but will speed up your project!

Finally, we wish you good progress and success completing the project! The more you implement, the more you learn, and (hopefully) in the end, it becomes fun.

Good luck, your PTSD supervisor team!