Fachhochschule Nordwestschweiz
Hochschule für Wirtschaft

Dokumentation Projekt 3: Project option 2 & 1: Implement an entirely new game with the implementation of a client

Projektmitglieder:
Severin Nyffenegger
Kevin Bernet

Dozent:
Bradley Richards

Olten, 31.12.2023

Programm starten und Erklärung:

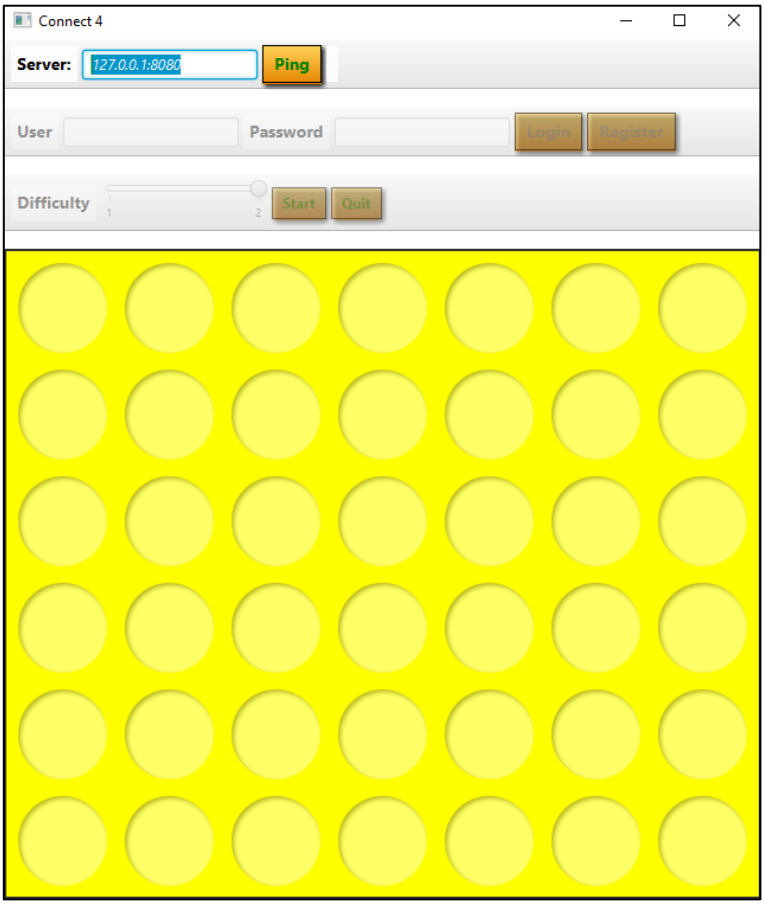
Wir haben zwei Projekte auf GitHub erstellet. Eines für das neue Spiel das wir implementiert haben. Wir haben uns für das Vier gewinnt spiel entschieden. Das zweite Projekt ist für den Client. Wir haben den Client mit JavaFX umgesetzt. Um uns die Aufgabe des Clients zu vereinfachen, haben wir das von Bradley Richards zur Verfügung gestellten JavaFX-Programm genommen und auf unser Projekt angepasst.
Nachdem die beiden Projekte von GitHub geklont worden sind, muss zuerst der Server mit der Klasse:

Project3\src\main\java\ch\fhnw\richards\aigs_spring_server\AigsServerApplication.java

gestartet werden. Der Server startet im Port 8080 und ein User «test» mit dem Passwort «test» wird auch gleich initiiert. Danach kann im Client das JavaFX-Programm mittels der Klasse:

Project3Client\src\main\java\com\example\project3client\Connect4.java

gestartet werden. Das GUI seht wie folgt aus:



Zuerst muss der Server gepingt werden. Danach kann man sich einloggen oder registrieren als neuen Benutzer. Schlussendlich kann die Schwierigkeitsstufe aussuchen und das Spiel starten. Die jeweiligen Abschnitte in der App werden jeweils erst freigeschaltet, wenn man die vorhergehende erfüllt hat. Beispielsweise kann man sich erst einloggen oder registrieren, wenn der Server korrekt gepingt worden ist.

Wie unser Programm aufgebaut ist:

Auf der Serverseite haben wir im Projekt im Ordner:

Project3\src\main\java\ch\fhnw\richards\aigs_spring_server\gameEngines

Einen neuen Ordner: Connect4 erstellt und die Klassen und Interfaces:

- C4_AI (Interface)
- Connect4 (Implements GameEngine)
- MinmaxPlayer (Implements C4_AI)
- Move
- RandomPlayer (Implements C4_AI)
- GameEngine (Interface)

vom TicTacToe-Spiel übernommen und angepasst.

Auf der Clientseite haben wir uns für die klassische MVC-Architektur entschieden. Im Package model werden alle Interaktionen mit dem Server geregelt. Im Package view ist das Layout der Applikation definiert. Im Package controller wird die Interaktion zwischen dem view und dem model geregelt.
Wie oben bereits erwähnt haben wir die JavaFX anwendung basierend auf dem TicTacToe Projekt aufgebaut. Somit haben wir ebenfalls im model, view und controller mehrere Klassen für Server (Ping), Userinteraktionen (Register/Login), Setup (Schwierigkeit des Gegners und Start des Spiels) und dem Game (das Spiel selbst). So ist das Projekt strukturiert und bleibt übersichtlich.

Game Clientseite

POST und GET Server Anfragen:

```
public void register(String serverAddress, String user, String password){
    User userQuery = new User();
    userQuery.setUserName(user);
    userQuery.setPassword(password);
    try {
        ObjectMapper objectMapper = new ObjectMapper();
        String sreq = objectMapper.writeValueAsString(userQuery);
        BodyPublisher json_req = BodyPublishers.ofString(sreq);
        HttpRequest request = HttpRequest.newBuilder()
                .uri(new URI( str: "http://" + serverAddress + "/users/register"))
                .setHeader( name: "Content-type", value: "application/json")
                .timeout(Duration.of( amount: 3, ChronoUnit.SECONDS))
                .POST(json_req)
                .build();
        HttpResponse<String> response = HttpClient.newBuilder().build().send(request, BodyHandlers.ofString());

        if (response.statusCode() == 200) {
            String responseBody = response.body();
            if (responseBody.contains("already exists")) {
                // Handle error message separately
                // For example:
                ObjectMapper errorMapper = new ObjectMapper();
                 Kevin
                Map<String, String> errorResponse = errorMapper.readValue(responseBody, new TypeReference<Map<String, String>>() {});
                String errorMessage = errorResponse.get("error");
                System.out.println("Error: " + errorMessage);
                // Handle the error message as needed (show an alert, log it, etc.)
                showErrorAlertMessage("This user already exists"); // Show alert for existing user
            } else {
                // Proceed with deserialization if no error
                ObjectMapper objectMapper2 = new ObjectMapper();
                objectMapper2.registerModule(new JavaTimeModule());
                User u = objectMapper2.readValue(responseBody, User.class);
                showAlertMessage("Registered Successfully!");
            }
        }
    } catch (URISyntaxException | IOException | InterruptedException e) {
        e.printStackTrace();
    }
}
```

Die GET und POST Methoden sind immer gleich aufgebaut. Wir erklären die am Beispiel der register Methode.
Die register Methode nimmt drei Parameter entgegen (String serverAddress, String user, String password). Zuerst wird ein neuer User erstellt und ihm dann den Username und das Passwort übergeben. Danach wird ein Objekt vom Typ ObjectMapper von Jackson erstellt, um den Benutzer (userQuery) in ein JSON-Objekt umzuwandeln. Dann wird mit *writeValueAsString* wird das Objekt in einen JSON-String umgewandelt. Danach wird ein *BodyPublisher* mit dem erstellten String erstellt. Der wird für den darauffolgenden POST-Request verwendet, um die Daten zu senden.
Danach wird der HttpRequest erstellt mit der URI ("http://" + serverAddress + "/users/register"), dem Header ("Content-type", "application/json") einer timeout duration von 3 Sekunden und einem POST-Request. Dann wird die vorbereitete Anfrage an der Server gesendet.
Wenn der Statuscode der gesendeten Anfrage 200 ist, wird die Antwort weiterverarbeitet. Danach wird geprüft, ob die die Antwort «already exists» enthält. Wenn ja, heisst das, dass bereits ein User mit dem Username besteht. Diese Antwort wird dann dem User aufbereitet als einen Alert angezeigt. Wenn der User noch nicht besteht, dann wird die Antwort des Servers in einen User umgewandelt und dem User den Alert gezeigt, dass er erfolgreich registriert ist.

In der Klasse SetupModel wird vor dem try-catch kein User-Objekt erstellt sondern ein neues Game-Objekt mit den dazugehörigen Attributen. Die URI ist selbstverständlich auch anders als bei der register-Methode und auch die Verarbeitung wenn der Serverantwortcode 200 ist.

Illegale Moves verhindern:

```
public void updateGame(Game game) {
    long[][] board = game.getBoard();

    for (int col = 0; col < 7; col++) {
        for (int row = 0; row < 6; row++) {
            Button button = buttons[row][col];
            if (board[row][col] > 0) {
                button.getStyleClass().removeAll( ...es: "blueButton", "gameButton");
                button.getStyleClass().add("redButton"); // Add redButton style class
            } else if (board[row][col] < 0) {
                button.getStyleClass().removeAll( ...es: "redButton", "gameButton");
                button.getStyleClass().add("blueButton"); // Add blueButton style class
            } else {
                button.getStyleClass().removeAll( ...es: "redButton", "blueButton");
                button.getStyleClass().add("gameButton"); // Reset to default style
            }

            button.setDisable(board[row][col] != 0 || game.getResult() == true);
        }
    }
}
```

Moves ausserhalb des Spieles sind im Client gar nicht möglich, denn es sind nur die Buttons verfügbar, wo auch ein Move gespielt werden kann. In der Methode oben «updateGame» in:

Project3Client\src\main\java\com\example\project3client\view\GameBoard.java

Erhalten die Buttons nicht nur die Farbe des jeweiligen Spielers, sondern werden sie auch deaktiviert, damit nicht im gleichen Feld zweimal ein Spielzug durchgeführt werden kann.

Game Serverseite

Connect4 Klasse:

Diese Klasse implementiert das GameEngine Interface und somit die Methoden newGame und move. Bei der Methode newGame mussten wir nicht viel anpassen, ausser das Spielbrett auf 6x7 ändern. In der move-Methode und weiteren Klassen mussten wir alles neu aufbauen, da wir die Logik einbauen mussten, dass der Coin immer in die tiefste mögliche Zeile in einer Spalte fallen muss. Um diese Logik einzubauen, hat uns Chat-GPT geholfen.

Die Methode getResult gibt uns das Ergebnis des Spiels zurück, true = gewonnen, false = verloren, null = am spielen. Um Herauszufinden, ob das Spiel im Gleichstand geendet hat, haben wir die Methode isBoardFull eingebaut. Diese überprüft uns, ob es auf dem Spielbrett eine freie Zelle gibt. Falls nein, wird das Game wie bei einem Win auf true gesetzt.

```
private boolean getResult(long[][] board) {
    Long winner = getWinner(board);
    return winner != null || isBoardFull(board);
}

1 usage   Kevin
private boolean isBoardFull(long[][] board) {
    for (int row = 0; row < board.length; row++) {
        for (int col = 0; col < board[0].length; col++) {
            if (board[row][col] == 0) {
                return false; // If any cell is empty, the board isn't full
            }
        }
    }
    return true; // No empty cell found, board is full
}
```

```
static Long getWinner(long[][] board) {
    long winner = 0;
    // Check all rows for horizontal win
    for (int row = 0; row < board.length; row++) {
        for (int col = 0; col < board[0].length - 3; col++) {
            if (board[row][col] != 0 &&
                    board[row][col] == board[row][col + 1] &&
                    board[row][col] == board[row][col + 2] &&
                    board[row][col] == board[row][col + 3]) {
                winner = board[row][col];
                return winner;
            }
        }
    }
```

Eine der wichtigsten Methoden ist getWinner. Diese Methode gibt uns zurück mittels einer Long Variablen in welcher Zelle der Gewinner steht. Nochmals zu Verständnis, in der Matix ist der Spieler die 1 und der AI ist die -1. Diese Methode hat 4 verschiedene Abschnitte, zum einen prüft sie ob 4 verschieden Zellen horizontal gleich sind (1 oder -1), zum anderen überprüft sie ob vertikal 4 darauffolgende Zellen den gleichen Wert haben. Ebenso überprüft es ob Diagonal nach links und rechts 4 darauffolgende Zellen den gleichen Wert haben. Wichtig zu erwähnen ist, dass die Spalten oder Zeilen im for-loop nicht 6 oder 7-mal durchgelaufen werden müssten, da es nicht möglich ist z.B. von der 5. Spalte aus einen Gewinner horizontal zu finden. Falls noch kein Gewinner gefunden wurde, gibt die Methode null zurück und somit wird das Spiel fortgeführt.

MinmaxPlayer Klasse:

Die Methoden basieren auf der TicTacToe-Engine, jedoch mussten wir diese Methoden umschreiben, damit die Logik von 4-Gewinnt eingebaut werden konnte. Wir implementieren ebenfalls eine AI interface und somit die Methode makeMove. In dieser Methode geben wir als Parameter unser 6x7 Board und erwarten, dass der AI einen passenden Spielzug macht. Als Antwort erwarten wir nur die Spalte, da die Logik von 4-Gewinnt vorschreibt, dass der Coin in die tiefste Mögliche Zeile fällt. Die passende Spalte finden wir mittels der Methode findMove.

```
@Override
public void makeMove(long[][] board) {
    int move = findMove(board, myPiece).col;
    int row = findFirstEmptyRow(board, move);
    if (row != -1) {
        board[row][move] = myPiece;
    }
}
```

In der Methode findMove sucht der AI einen passenden Spielzug und simuliert mittels der Methode copyBoard seinen Spielzug. Die copyBard-Methode konnten wir ohne Anpassungen übernehmen. Mittels der Methode miniMax evaluiert er, ob dieser Spielzug der bestmögliche war. Dies macht er maximal 6-mal, um in keine Endlosschlaufe zu geraten und somit den Server

ihn nach 3 Sekunden Time outen würde. Falls kein besserer Spielzug gefunden wurde, gibt die Methode die beste Spalte zurück.

Wir hatten Probleme, dass der MinmaxPlayer wenn er keinen schlauen Move ausführen konnte immer in der 3. Reihe den Coin (-1) setzte, was nicht möglich ist. Deswegen haben wir in der Move-Methode das if-Statement eingefügt, wenn der Bot die Spalte -1 aussucht, dann wird die Methode playRandomMove aufgerufen, sonst wird vom Bot kein Move ausgeführt und der User spielt zwei Züge hintereinander. In der Methode wird zuerst eine Liste erstellt mit den Spalten, die noch freie Zeilen haben und anschliessend wird daraus mittels Zufallsprinzips eine dieser Spalten ausgewählt.

```
private int playRandomMove(long[][] board) {
    List<Integer> availableColumns = new ArrayList<>();
    for (int col = 0; col < board[0].length; col++) {
        if (board[0][col] == 0) { // Check if the top of the column is empty
            availableColumns.add(col);
        }
    }

    if (!availableColumns.isEmpty()) {
        Random rand = new Random();
        return availableColumns.get(rand.nextInt(availableColumns.size()));
    }

    return 0; // Return column 0 if no other move is available (should not happen if called correctly)
}
```

RandomPlayer Klasse:
Diese Klasse wird gerufen, falls der Spieler gegen einen schwachen AI spielen will. Der RandomPlayer erstellt zuerst eine Liste mit den Spalten, die noch nicht gefüllt sind. Danach sucht er aus den verfügbaren Spalten eine aus und überprüft welche Zeile die tiefst mögliche ist, da der Coin immer in die unterste Zelle fallen muss. Mit diesen zwei Informationen setzt der schwache AI seinen Zug. Somit wird die Wahl des Spielzugs ohne Logik gewählt. Im gegensatz zum starken AI berücksichtig dieser nicht den Spielzug des menschlichen Spielers.

```
@Override
public void makeMove(long[][] board) {
    List<Integer> availableColumns = new ArrayList<>();
    for (int col = 0; col < board[0].length; col++) {
        if (board[0][col] == 0) { // Check if the top of the column is empty
            availableColumns.add(col);
        }
    }

    if (!availableColumns.isEmpty()) {
        Random rand = new Random();
        int col = availableColumns.get(rand.nextInt(availableColumns.size()));
        int row = findFirstEmptyRow(board, col);
        board[row][col] = -1; // Assuming -1 is the AI's piece
    }
}
```

Move Klasse:
Diese Klasse haben wir unverändert von der TicTacToe-Engine übernommen.

Methoden
Einzelne Methoden, die nicht selbsterklärend sind, sind direkt im Code mit Kommentaren beschrieben.