Dokumentation zum Projekt „4 Wins“

Modul: Programmieren 2 Praktikum
Datum: 07.06.2022

Name:	Matrikelnummer:
Furkan Akbas	2210083
Noel Thorwesten	2190180
Mohammad Azizi	2190043

Einleitung zu 4 Gewinnt

Im Rahmen des vorliegenden Projekts wurde das bekannte Brettspiel „4 Gewinnt“ auf Grundlage der vorgegebenen Anforderungen umgesetzt. Es handelt sich um ein Zweipersonenspiel, in dem jeder Spieler abwechselnd Spielsteine in das Brett fallen lassen, um 4 gleiche Steine entweder vertikal, horizontal oder diagonal als Erster zu platzieren. Der Spieler, der dies zuerst schafft, gewinnt. Sind alle Felder mit Steinen befüllt und kein Spieler hat es geschafft eine Reihe von vier gleichen Steinen zu platzieren, geht das Spiel als unentschieden aus.

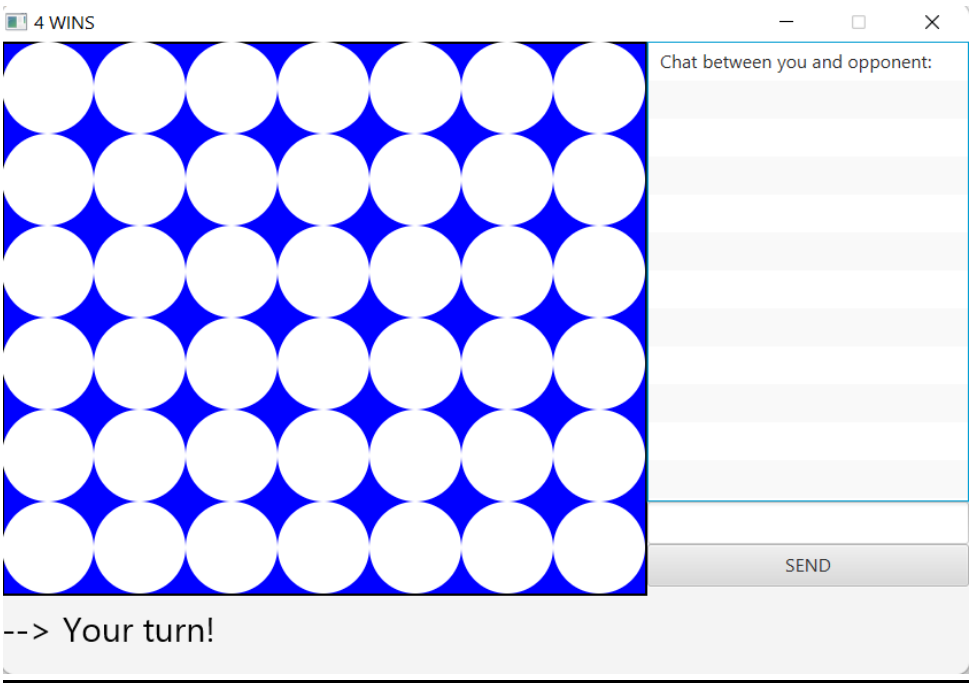
Das Spielbrett besteht aus sechs Reihen und sieben Spalten, wie in der unteren Abbildung zu sehen ist. Jeder Spieler bekommt zum Beginn entweder rote oder gelbe Spielsteine, die er in das Brett fallen lässt.

Vorstellung des Spiels:

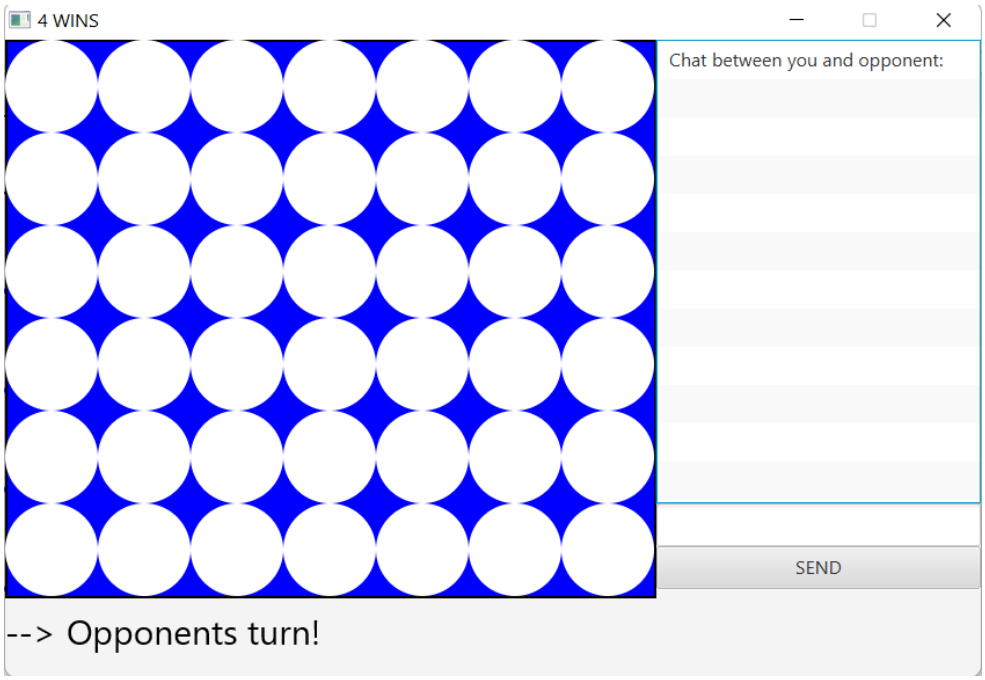
Bevor zwei Gegner gegeneinander antreten können, muss der „Server“, der zur Kommunikation des Spiels dient, gestartet werden. Daraufhin muss jeder Spieler die Klasse Game starten, um mit dem Spielen zu beginnen. Nach Starten der zuvor genannten Klasse ist folgender Willkommensscreen zu sehen:

Sobald der Spieler auf den Button „Start Game“ klickt, wird im Hintergrund eine Verbindung zum Server aufgebaut und mitgeteilt, dass ein Spieler sich zum Spielen registriert hat und auf einen Gegner wartet. Verbindet sich daraufhin ein weiterer Spieler zum Server, werden diese zwei Spieler zu einem Duell hinzugefügt und sehen die folgenden Screens:

Spieler 1 (hat sich zuerst zum Server verbunden, weshalb dieser als Erstes einen Spielzug machen darf):



Spieler 2 (hat sich nach Spieler 1 zum Server verbunden, wartet deshalb auf den Spielzug von Spieler 1):



Es ist außerdem möglich dem Gegner Nachrichten zu schreiben, wie auf der rechten Seite des Screens zu sehen ist. Versucht ein Spieler einen Spielzug zu betätigen, obwohl dieser nicht an der Reihe ist, wird eine entsprechende Fehlermeldung angezeigt und den Spieler darüber informiert, dass der Gegner dran ist.
Ebenso werden passende Meldungen angezeigt, sobald das Spiel durch Sieg, Niederlage, Unentschieden oder Aufgeben beendet ist. Im Anschluss wird wieder der Startscreen angezeigt und es kann ein neues Spiel begonnen werden.

Herangehensweise:

Zur Implementierung der Aufgabe wurde diese in folgende, logische, technische Schritte unterteilt:

1. Implementierung der Grund-Oberfläche mittels JavaFX
2. Funktionalitäten zur Oberfläche implementieren
3. Logik zum Spielverfahren implementieren
4. Verteilung des Spiels implementieren

Mit dieser Herangehensweise konnten wir Schritt für Schritt mit der notwendigen Materie vertraut machen.
In den nachfolgenden Punkten werden diese einzelnen Schritte technisch näher erläutert.

1. GUI mittels JavaFX

   Das Spiel ist grundlegend in 2 Screens unterteilt: StartScreen und GameScreen. Aus diesem Grund besteht für jeden Screen eine separate Klasse – StartView und GameView.

Diese werden jeweils in der Klasse Game instanziiert, welche die Klasse Application aus der JavaFX-Bibliothek „extended“ bzw. von der Klasse Application erbt und die start-Methode überschreibt.

Zur Positionierung der verschiedenen UI-Elemente werden BorderPanes in beiden Views genutzt. Das Spielbrett ist ein Rectangle-Objekt, welches fixiert auf eine Width von 560 und eine Height von 480 ist, da das Spielbrett aus 7 Spalten und 6 Reihen besteht und wir für ein Spielstein einen Durchmesser von 80 verwendet haben.

Um den Zustand des Spielbrettes speichern zu können, wurde ein zweidimensionales Integer-Array verwendet:

- int[][] board = new int[7][6];

Jedes Feld in diesem Array kann 3 Werte annehmen:

- 0: kein Spieler hat hier einen Spielstein platziert (Feld ist weiß)
- 1: Spieler 1 hat hier einen Spielstein platziert (Feld ist rot)
- 2: Spieler 2 hat hier einen Spielstein platziert (Feld ist gelb)

Dementsprechend befindet sich in der Klasse GameView eine drawBoard()-Methode, welche das Spielbrett neu „zeichnet“.

```
for (int i = 0; i < 7; i++) {
    for (int j = 0; j < 6; j++) {
        // create new circles holes for empty spaces in board
        // game board has 7 columns and six rows each has height and width of 80
        Circle circle = new Circle((i * 80) + 40, (j * 80) + 40, 40, Color.WHITE);

        // if current index of array has a value, fill with corresponding color
        if (board[i][j] == 1)
            circle.setFill(Color.RED);
        if (board[i][j] == 2)
            circle.setFill(Color.YELLOW);
        pane.getChildren().add(circle);
    }
}
```

In dieser Methode wird über das zweidimensionale Array iteriert mithilfe einer verschachtelten for-Schleife. Die erste bzw. äußere for-Schleife iteriert durch die Spalten und die innere iteriert durch die Reihen pro Spalte. Pro Iteration wird ein Circle-Objekt erstellt und platziert.

Mithilfe der Zählvariablen i und j werden die Positionen der Circle-Objekte bestimmt. In der ersten äußeren Iteration werden alle Kreise in der ersten Spalte gerendert. An den Positionen:
(40, 40)
(40, 120)
(40, 200)
(40, 280)
(40, 360)
(40, 440)

Sodass folgende Positionen entstehen:

(40, 40)	(120, 40)	(200, 40)	(280, 40)	(360, 40)	(440, 40)	(520, 40)

(40, 120)	(120, 120)	(200, 120)	(280, 120)	(360, 120)	(440, 120)	(520, 120)
(40, 200)	(120, 200)	(200, 200)	(280, 200)	(360, 200)	(440, 200)	(520, 200)
(40, 280)	(120, 280)	(200, 280)	(280, 280)	(360, 280)	(440, 280)	(520, 280)
(40, 360)	(120, 360)	(200, 360)	(280, 360)	(360, 360)	(440, 360)	(520, 360)
(40, 440)	(120, 440)	(200, 440)	(280, 440)	(360, 440)	(440, 440)	(520, 440)

Nach Erstellen der einzelnen Kreis-Objekte werden diese zu den „Children-UI-Elementen“ des Borderpanes hinzugefügt.

2. Funktionalitäten zur GUI hinzufügen

Um den Zustand des Spielbrettes zu ändern, muss ein Spieler auf das Feld/die Spalte klicken, in die der neue Spielstein platziert werden soll. Um dies zu ermöglichen, müssen zur Oberfläche „ClickEvents“ registriert werden. Dazu bietet JavaFX eine Methode, die „setOnMouseClicked“, welche einen Event-Parameter innehält, der unter anderem angibt, welcher Mausbutton geklickt wurde. Dazu ist es ebenfalls möglich auf die X- und Y-Positionen zurückgreifen, an der in der View geklickt wurde.

Diese Methode wurde zu Borderpane-Objekt hinzugefügt und nach jedem Klick auf das Borderpane-Objekt wird mithilfe der X-Position des Events berechnet, in welche Spalte geklickt wurde, indem die X-Position durch 80 dividiert wird und das Ergebnis in ein Integer-Wert umgewandelt wurde.
Anschließend wird in dem Integer-Array an der jeweiligen, geklickten Spalte das nächstmögliche freie Feld in der Reihe berechnet und in dem Integer-Array aktualisiert. Nach jeder Veränderung des Arrays wird das Spielbrett neu gerendert durch den Aufruf der Methode drawBoard().

3. Logik zum Spielverfahren

Die Logik des Spielverfahrens besteht aus drei zentralen Bestandteilen:

- Setzen eines Spielsteins
- Reihenfolge der Spieler beachten und behandeln, was passieren soll, wenn der Spieler auf das Brett klickt, wenn dieser nicht an der Reihe ist
- Kontrollieren, welcher Spieler gewonnen hat bzw. ob das Spiel unentschieden ausgegangen ist

Das Setzen der Spielsteine funktioniert wie im oberen Punkt erläutert.

Die Variable *int player* speichert die Information über den Spieler, der als Nächstes an der Reihe ist. Diese Variable wird nach dem Setzen eines Spielsteins ebenfalls angepasst. Klickt ein Spieler, der nicht an der Reihe ist, wird ihm eine Fehlermeldung über ein Alert angezeigt. Dazu besteht folgende Funktion:

```
public static Alert showMessageDialog(String msg) {
   Alert dialog = new Alert(AlertType.INFORMATION);
   dialog.setContentText(msg);
   dialog.showAndWait();
```

```
    return dialog;
}
```

Nach jedem Setzen eines Spielsteins wird ebenfalls kontrolliert, ob ein Spieler gewonnen hat oder das Spiel unentschieden ausgegangen ist. Dazu existieren folgende Methoden in der Klasse *GameView*:

hasFourEqualCirclesInColumn – kontrolliert pro Spalte, ob 4 gleichfarbige, aufeinanderfolgende Spielsteine existieren
hasFourEqualCirclesInRow – kontrolliert pro Reihe
hasFourEqualCirclesDiagonal – kontrolliert diagonal
checkIfTiedGame – kontrolliert auf Gleichstand, wenn alle Felder belegt sind und die vorherigen 3 Methoden false liefern

4. Verteilung des Spiels implementieren

Damit das Spiel auf mehreren Rechnern verteilt funktionieren kann, muss eine Möglichkeit zur Kommunikation dieser Rechner bestehen. Hierzu bietet sich eine Socketverbindung auf der TCP-Ebene hervorragend an.
Dabei wird in der Server-Klasse ein ServerSocket auf dem Port 7777 gestartet, welcher auf Verbindungen eines Client-Sockets wartet und diese dann durch die Methode *accept()* akzeptiert.
Diese Methode liefert ein Socket-Objekt zurück, welches gespeichert werden kann und In- und Outputstreams bereitstellt mit der zum ServerSocket Informationen versendet werden können.
Sobald zwei Sockets sich zum Server verbinden konnten, werden diese zu einem Spiel hinuzgefügt und können dann gegeneinander spielen. Hierzu besteht eine weitere Klasse namens Protocol, welche die Befehle, die zwischen Spielern ausgetauscht werden kann vorschreibt.

```
public class Protocol {
    public static final String WAITING_FOR_OPPONENT = "waiting";
    public static final String OPPONENT_FOUND = "opponent found";
    public static final String GAME_OVER = "game over";
    public static final String MOVE = "move";
    public static final String QUIT = "quit";
}
```

Diese Befehle können von den Spielern aus zum Server gesendet werden über die vorhandene Socketverbindung. Die Klasse namens *ClientHandler* wird zum Empfangen von Client-Nachrichten auf dem Server verwendet und in einem separaten Thread gestartet, weshalb diese Klasse das Runnable Interface implementiert. Dies ermöglicht, dass der Server zustandslos ist. Der ClientHandler wartet auf Nachrichten durch den Inputstream des einen Spielers und leitet diese sofort an den anderen Spieler weiter.

Das Verhalten eines Clients wird in der Klasse Client definiert. Der Client ist zuständig den Server über die Züge des Spielers zu informieren und sendet dazu auf dem Outputstream Nachrichten an den Server.

Die Nachrichten, die der Client vom Server erhalten soll, wird in der ServerListener-Klasse vom Inputstream ausgelesen. Um dies ebenfalls zustandslos zu ermöglichen, wird auch hierbei ein separater Thread gestartet.
Das bedeutet, wenn ein Spieler auf den Button „Start game“ klickt, wird im Hintergrund für diesen Spieler ein Objekt der Client-Klasse erstellt, welcher sich zum ServerSocket verbindet. Diesen bestehenden Socket übergibt die Client-Klasse an den ServerListener und startet diesen in einem neuen Thread.
Die Zuständigkeit des Serverlisteners ist es auf Nachrichten vom Server zu warten. Die Zuständigkeit des Clients ist es Nachrichten zum Server zu schicken, wie z.B. die Information darüber, wo der Spieler seinen Stein gesetzt hat. Man hätte den Server und den Client auch ohne Helferklassen, die auf einem separaten Thread laufen, implementieren können, wobei dann der Server dann einen festdefinierten Zustand/Ablauf hätte, wie z.B.

- Der Server wartet auf den Zug des ersten Spielers, leitet diesen dann weiter
- Danach empfängt der Server keine Nachrichten mehr vom ersten Spieler bis der Gegner seine Informationen geschickt hat
- Server wartet auf den Zug des ersten Spielers
- Server wartet auf den Zug vom zweiten Spieler usw.

Eine Problematik, die dabei entstanden ist, war z.B. was passiert, wenn während der Spieler 1 seinen Zug macht, der Spieler 2 das Spiel abbricht?

Mithilfe der ServerListener und ClientHandler werden solche Probleme umgegangen, weil diese Klassen jederzeit unabhängig vom Spielablauf Nachrichten empfangen können. Deshalb haben wir ebenfalls einen Chat hizugefügt, der diesen Vorteil demonstrieren soll. Beide Klassen lesen in einer while-Schleife alle Informationen als String aus und kontrollieren dann in einem switch-Zweig, ob der Befehl ein gültiger Befehl der Klasse Protocol ist. Wenn dies der Fall ist, werden ja nach Befehl unterschiedliche Aktionen ausgelöst. Ist der Befehl ungültig, wird dieser als Chat-Nachricht interpretiert, wie hier zu sehen:

```java
public void run() {
    System.out.println("Server listener running...");
    command = "";

    try {
        while (!command.equals(Protocol.QUIT)) {
            command = this.dis.readUTF();
            System.out.println("Remote: " + command);

            switch (command) {
            case Protocol.WAITING_FOR_OPPONENT:
                int turn = this.dis.readInt();
                System.out.println("You are player " + turn);
                gameController.setPlayer(turn);
                startController.showAlert();
                break;
            case Protocol.OPPONENT_FOUND:
                startController.startGame();
                break;
            case Protocol.MOVE:
                int row = this.dis.readInt();
                int column = this.dis.readInt();
                gameController.setOpponentCircle(row, column);
                break;
            case Protocol.GAME_OVER:
                break;
            case Protocol.QUIT:
                System.out.println("Opponent quit");
                gameController.send(Protocol.QUIT);
                this.command = Protocol.QUIT;
                Platform.runLater(() -> gameController.showDialog("Opponent left the game! Start new game."));
                break;
            default:
                ChatHandler.addMessage(command, 2);
                Platform.runLater(() -> gameController.showChatMessages());
                break;
            }
        }
    } catch (IOException e) {
        System.out.println("Error occured.");
    }
    System.out.println("Listener stopped.");
```