

Projekt - Lösungsdokument v2

Snake-Core, SnakeFX, SnakeServer

Ostfalia Fachhochschule für angewandte Wissenschaften

Benjamin Wulfert
Leonard Reidel

Semester: Wintersemester 2020

29. November 2020

Inhaltsverzeichnis

Projektdokumentation	4
Modul-Architektur	4
Core	4
SnakeFX - Frontend	4
SnakeServer - Backend	4
SnakeFX - Front End	5
Login-Screen	5
Registrierungs-Prozess	5
Home-Screen	5
Aktive Spieler	5
Aktive Spiele	5
Spielhistorie-Screen	5
New-Game-Screen	6
Snake-Implementierung	6
SnakeServer	8
Persistenz-Layer	8
H2 Database Engine	8
API-Layer / Schnittstelle	9
Tasks / Aufgabenstellungen	9
Realisiert	10
UML-Diagramme	10
Front-End Architektur	10
User-Interfaces	10
Initiale Asynchronität	10
Initialer Konsum der Schnittstelle	10
Planung und Realisierung der Bewegungsberechnung der Snake-Implementierung	10
Snake-Implementierung – Mechanik - Wegfindung	11
Integration der Design Patterns	11
Überarbeitung der UML-Diagramme	11
Ausstehende Arbeitspakete	11
Absicherung der Schnittstelle (API) mittels Spring Security und JSON-Web-Tokens	11
Vollständige Implementierung der Klassenrelationen	11
Vollständiger Konsum der Schnittstellen	12
Austausch der Spielereingaben (Implementierung von Web-Sockets)	12
Komplexität des GameControllers reduzieren	12

Regelwerk ausbauen	12
Feedback aus der Präsenz	12
RespawnSnake() → Raus.....	12
Draw-Funktionalität in Klassen auslagern.....	12
Szenen-Wechsel.....	13
Design-Patterns	14
Composite-Pattern.....	14
Factory-Pattern.....	14
Observer-Pattern	15
Anhang	17
UML – Anwendungsfalldiagramm	17
UML – Aktivitätsdiagramm.....	17
UML – Klassendiagramm.....	19
User-Interface – Frontend - SnakeFX	20

Projektdokumentation

Dieses Dokument stellt die Dokumentation für das Projekt dar. Im Folgenden werden die verschiedenen Aspekte des Projekts beschrieben. Des Weiteren wird in dieser Lösungsdokumentation dargelegt welche Aspekte des Systems umgesetzt wurden sowie noch ausstehende Punkte erläutert, welche in den folgenden Wochen umgesetzt werden.

Der aktuelle Projektstand (Dokumente, UML-Diagramme, gesamter Projektcode, etc.) kann dem folgenden GitHub-Repository entnommen werden:

https://github.com/Bummelnderboris/Patterns_and_Frameworks

Modul-Architektur

Das gesamte Anwendungssystem wird mit dem Build-Management-System Apache Maven realisiert. Dabei wurde ein Front-End Modul (SnakeFX), ein Back-End Modul (SnakeServer), und ein Core Modul geplant und implementiert welche im Folgenden beschrieben werden.

Das folgende Schaubild stellt die Modul-Architektur des Systems dar.

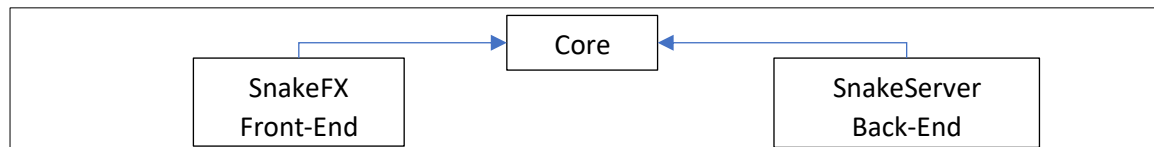


Abbildung 1 - Modul-Architektur

Core

Das Core-Modul enthält die Kern-Aspekte der Anwendung – dazu zählen beispielsweise Ausschnitte der Modelle welche im Klassendiagramm enthalten sind, die gemeinsam genutzten Endpoints der Schnittstelle sowie Konstanten welche sowohl im Backend als auch im Frontend verwendet werden. Das Core-Modul enthält also Programmteile welche sowohl im Front- als auch im Backend verwendet werden.

SnakeFX - Frontend

Das Modul SnakeFX ist das Front-End der Anwendung. Im Front-End sind die User Interfaces (UI) definiert und implementiert. Des Weiteren konsumiert das Front-End mittels REST-Schnittstelle Daten aus dem Backend. Die Implementierung und die gesamten Mechaniken des Snake-Spiels sind ebenfalls Teil des Front-Ends.

SnakeServer - Backend

Das Modul SnakeServer enthält alle Aspekte des Backend – dazu zählen die Persistenz-Schicht der Anwendung welche mit Spring Data JPA / Hibernate realisiert werden, so wie eine in Java implementierte Datenbank welche direkt mit dem Backend initialisiert wird (H2) – die Vorteile dieses Vorgehens werden weiteren Verlauf des Dokuments dargestellt. Ein weiterer Aspekt des Backend ist die Bereitstellung der REST-Schnittstelle sowie die Auslieferung der Persistenz-Daten darüber. Der letzte Aspekt des Backend ist die direkte bidirektionale Kommunikation mit den angemeldeten Clients über Web-Sockets um Eingaben der Spieler entgegenzunehmen und an die Teilnehmer einer Runde zu replizieren.

SnakeFX - Front End

Die folgenden Abschnitte beschreiben den Aufbau und die Prozesse des Front-Ends:

Login-Screen

Der Login-Screen enthält vier Schaltflächen für die Interaktion des Benutzers. Zwei Texteingaben zur Angabe eines Benutzernamen und eines Passworts – und zwei Schaltflächen / Buttons um die Benutzerangaben (Name, Passwort) mittels Login oder Registrierung an das Backend zu übertragen. Beide Prozesse werden im Folgenden näher beschrieben.

Registrierungs-Prozess

Das Front-End sendet einen POST-Request an die HTTP-Schnittstelle des Backends. Diese URL lautet <http://localhost:8080/api/login>. Als Header-Daten des Post-Requests werden der Benutzername sowie dessen Passwort (Hash) versendet. Diese Daten werden auf der Seite des Backends empfangen. Die empfangenen Daten werden mittels Unmarshalling¹ vom Backend in eine Instanz der Benutzer-Klasse umgewandelt und in die Datenbank persistiert. Die Nutzerdaten des registrierten Benutzers können anschließend für die Anmeldung am System verwendet werden.

1 - <https://docs.oracle.com/javase/7/docs/api/javax/xml/bind/Marshaller.html>

Home-Screen

Der Home-Screen stellt die zentrale Benutzeroberfläche der Anwendung dar. Der Benutzer erhält darauf Zugriff nach einem erfolgreichen Anmeldeversuch. Der Home-Screen bietet die Möglichkeit, alle aktiven Spieler und alle aktiven Spiele des Systems zu betrachten. Des Weiteren gelangt der Benutzer über den Home-Screen zur Spielhistorie-Oberfläche. Durch Betätigung der Schaltfläche „Neues Spiel“ ist der Benutzer in der Lage neue Spielrunden zu definieren und in der Lobby zu veröffentlichen. Aktive Spieler können dann, sofern noch genügend Kapazitäten vorhanden sind, dem Spiel beitreten.

Aktive Spieler

Zeigt alle aktiven Spieler (am System angemeldete Benutzer) tabellarisch an.

Aktive Spiele

Zeigt alle aktiven Spiele (Spielrunden des Systems) tabellarisch an. Ein Spieler kann an einem Spiel teilnehmen (sofern das Spiel noch nicht begonnen wurde oder ein Spieler-Slot verfügbar ist).

Spielhistorie-Screen

Die Benutzeroberfläche für die Spielhistorie zeigt dem Benutzer alle in der Vergangenheit gespielten Spiele an welche im Backend persistiert worden.

New-Game-Screen

Der New-Game-Screen erlaubt es einem Anwender ein neues Spiel zu definieren und diese Definition in der Lobby des Backend zu posten. Interessierte Spieler können sich in einem angemeldeten Spiel eintragen. Der Anwender welcher das Spiel definiert hat gilt als Admin und kann dadurch bestimmen wann das Spiel gestartet wird.

Snake-Implementierung

Für das Spiel wird ein Spielgrund mithilfe einer Config.java Klasse generiert, wobei über *rows* und *columns* ein x,y-Koordinaten System aufgespannt wird; mit diesem sowie einem Timer (timeline), lassen sich alle benötigten Operationen realisieren. (Die Map wird dabei auf einer *Stage* aufgebaut, dass diese losgekoppelt von dem laufenden Menü berechnet werden kann).

Aufbauend auf diesem Konzept betrachten wir nun drei (oder eigentlich zwei) Kernelemente des Spiels Snake:

1. Schlangenexistenz und -bewegung

1.1. Schlangeninitialisierung:

Es wird auf einer vorgegebenen Koordinate ein Punkt erzeugt, welcher mit dem nächsten Tick die x und y Koordinate um 1 dekrementiert und an selber erneut einen Punkt erzeugt, bis die Anzahl an Punkten (Startschlangenlänge) erreicht ist.

```
public Snake(Vector2 spawn, Color color){
    for (int i = 0; i < initialLength; i++) {
        Vector2 bodyPart = new Vector2( x: -1, y: -1);
        body.add(bodyPart);
    }
    head = body.get(0);

    head.x = spawn.x;
    head.y = spawn.y;

    this.color = color;
}
```

1.2. Schlangenbewegung

Die Bewegung der Schlange funktioniert, indem jedes Listenelement der Schlangenliste sich das Vorelement der Liste als neue Position holt, das letzte Listenelement entfernt wird und der Kopf der Schlange (snake.head, definiert als snake.body(0)) auf das Feld, errechnet aus snake.head, snake.currentDirection und der Eingabe (Eingabe realisiert über KeyFrames), gesetzt wird.

```
for (Snake snake : snakeList)
{
    // calculate the current position for each snake
    for (int i = snake.body.size() - 1; i >= 1; i--) {
        snake.body.get(i).x = snake.body.get(i - 1).x;
        snake.body.get(i).y = snake.body.get(i - 1).y;
    }
    snake.head = snake.head.add(snake.currentDirection);
}
```

2. Power-Ups/Food Generierung und Essen

Power-Ups/Food wird über eine Liste aus einer Koordinate und einem Bild realisiert. Dabei wird auf einer zufälligen Stelle des Koordinatensystems ein Essensteil generiert. Ist ein Schlangenkopf auf

derselben Koordinatenposition hat dies die Auswirkung (hier), dass der Schlange ein weiteres Listenelement angehängt wird und generateFood() aufgerufen wird.

```
private void checkEatFood() {  
    for (Snake snake : snakeList) {  
        for (Food food : foodList) {  
            if (snake.head.getX() == food.getPosition().x && snake.head.getY() == food.getPosition().y) {  
                foodList.remove(food);  
                snake.body.add(new Vector2( x: -1, y: -1));  
                generateFood();  
                score += 5;  
            }  
        }  
    }  
}
```

Die Realisierung des Essens als Liste ist, um mehrere Essenselemente gleichzeitig generieren zu lassen. (Regeln hierfür/sowie Power-Ups ausstehend)

3. Aktionen (Schlange, trifft Wand, andere Schlange, Essen, sich selbst...)

Eine jede Aktion lässt sich dann gleich der Essensaufnahme über einen einfachen Check, ob ein Schlangenkopf sich auf bestimmten Koordinaten befindet implementieren. Je nach Element, können verschiedene Regeln in Kraft treten.

```
public void checkGameOver() {  
    // check for wall collision  
    for (Snake snake : snakeList) {  
        if (snake.head.x < 0 || snake.head.y < 0 ||  
            snake.head.x > config.rows -1 ||  
            snake.head.y > config.columns -1) {  
            gameOver = true;  
        }  
    }  
    // destroy itself  
    for (int i = 1; i < snake.body.size(); i++) {  
        if (snake.head.x == snake.body.get(i).getX() && snake.head.getY() == snake.body.get(i).getY()) {  
            gameOver = true;  
            break;  
        }  
    }  
    // check for snake-collision  
    for (Snake a : snakeList) {  
        for (Snake b : snakeList) {  
            if (a.head != b.head && a.head.x == b.head.x && a.head.y == b.head.y) {  
                gameOver = true;  
            }  
        }  
    }  
}
```

4. Sonderaktion: Schlange beißt anderer Schlange etwas ab [Ausstehend]

Es ist angedacht, die Situation, mit einer Schlange den Körper einer anderen abbeißen zu können als eine temporäre Fähigkeit eines Power-Ups einführen. (Um zufällig zwischen verschiedenen Power-Ups auszuwählen, kann man eine Zufallsvariable über der listenlänge der verschiedenen Power-Up typen anbringen.) Pseudo-Code für die Struktur:

Snake.head = Snake.body(0);

```
If (Snake_playerA.head.x_coord == Snake_playerB.body.x_coord &&  
    Snake_playerA.head.y_coord == Snake_playerB.body.y_coord) {  
    Check list position of coordinate in Snake_playerB.body;
```

```
Adding = Length(Snake_playerB.body) - position of coordinate in Snake_playerB.body;  
// Here might be a -1 important for the list calculation.  
Drop list elements of Snake_playerB.body after Snake_playerB.body(Adding);  
  
While adding > 0{  
    Add coordinate of Snake_playerA.body(length) to Snake_playerA.body;  
    Adding - 1;  
}  
// With this logic the snake of playerA will grow the length of snake player but the case that the length  
might brake out of the field wont happen because it adds the same way snakes are spawning, with  
multiple point from one dot.  
}
```

SnakeServer

Das Snake-Server Modul stellt das Backend der Anwendung dar. Teil des Moduls ist eine Datenbank sowie deren Anbindung an das Backend. Des Weiteren stellt SnakeServer die verschiedenen Schnittstellen zur Verfügung welche zur Kommunikation mit dem Front-End benötigt werden.

Das Backend ist in verschiedene Ebenen / Layer unterteilt.

Persistenz-Layer

Der Persistenz-Layer verwaltet die Speicherung, Aktualisierung und den Bezug von Daten aus dem relationalen Datenbank Management-System (RDBMS).

Als RDBMS verwenden wir H2 – welche sowohl als In-Memory als auch als File-Storage variante verwendet werden kann. Dieses Vorgehen erweist sich insbesondere für das Aufsetzen der Anwendung als Hilfreich, da die Installation und Konfiguration des RDBMS entfällt. Des Weiteren vereinfacht das Vorgehen die Realisierung und Nutzung von Unit-Tests.

Für die Realisierung mittels der Programmiersprache Java wird das in Spring enthaltene Spring-Data JPA verwendet, welches eine Spezifikation für JPA (Java Persistence Layer) darstellt. Hibernate wird als Implementierung für das ORM-Framework verwendet und direkt von Spring-Data genutzt.

H2 Database Engine

Ein weiterer Vorteil der Nutzung von H2 ist das in der Abhängigkeit enthaltenen Weboberfläche zur Verwaltung der Datenbank, sodass auch die Installation und Konfiguration eines solchen Tools entfällt:

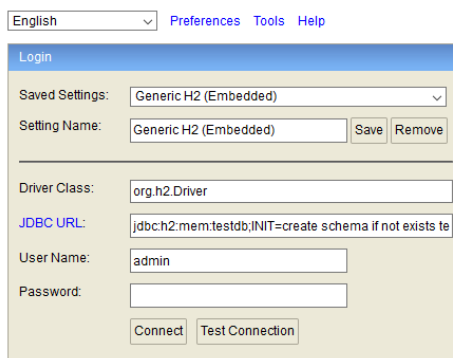


Abbildung 2 – Login des User-Interfaces zur Verwaltung der In-Memory / File-Storage Datenbank.
Die URL zur Datenbank lautet: jdbc:H2:mem:testdb.

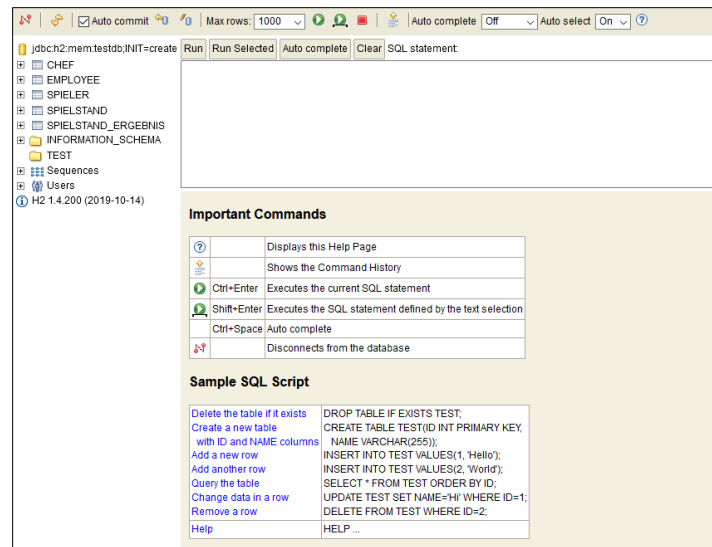


Abbildung 3 - User-Interface zur Verwaltung der Tabellen

API-Layer / Schnittstelle

Der API-Layer definiert die vom Backend bereitgestellten Schnittstellen welche zur Kommunikation vom Frontend mit dem Backend benötigt werden. Die Schnittstelle des Backend basiert auf dem HTTP (Hypertext Transfer Protocol) und stellt eine REST-Schnittstelle dar (Representational State Transfer). Dies bedeutet, dass jeder HTTP-fähige Client die Schnittstelle des Backend konsumieren (z.B. auch Internetbrowser, cURL, etc.) kann.

Das Front-End *SnakeFX* verwendet die Java-Bibliothek *Unirest* für die Kommunikation zwischen Front- und Backend. Die Funktionsweise einer REST-Schnittstelle basiert auf dem Gedanken die grundlegenden Operationen des HTTP – wie z.B. GET, PUT, POST, DELETE, ... - auf Endpunkte / URLs eines Systems abzubilden. Dabei soll eine HTTP-GET Anfrage (Request) nur für den Bezug von Daten zuständig sein – ein HTTP-POST oder -PUT Request hingegen für die Entgegennahme neuer Daten.

Die Schnittstelle bietet folgende Endpoints für die Kommunikation mit Clients an:

URL	HTTP-Methode	Beschreibung
http://localhost:8080/api/login	POST	Liefert einen JSON-Web-Token zurück welcher für die Kommunikation mit anderen Endpoints genutzt werden kann
http://localhost:8080/api/register	POST	Registriert einen Spieler mit einem Benutzernamen und Passwort
http://localhost:8080/spieler/	GET	Gibt eine Liste alle Spieler als JSON zurück
http://localhost:8080/spiele	GET	Gibt eine Liste aller Spiele als JSON zurück
...

Tasks / Aufgabenstellungen

In den folgenden Auflistungen werden die im Projekt geplanten Aufgabenpakete vorgestellt. Damit sind sowohl bereits implementierte / realisierte Aufgaben als auch noch ausstehende Aufgaben (-blöcke) gemeint.

Realisiert

Die folgenden Arbeitspakete wurden im Laufe des Moduls bereits realisiert:

UML-Diagramme

Zu Beginn des Moduls wurden sowohl Anwendungsfall-, Aktivitäts- als auch Klassendiagramme erdacht und angefertigt. Anhand dieser Diagramme wurden die ersten Aspekte für das Projekt realisiert und implementiert.

Front-End Architektur

Das Front-End besitzt eine gemeinsame Architektur welche genutzt werden kann um verschiedene Benutzeroberflächen / UIs / „Fenster“ aufrufen und in der Anwendung registrieren zu können. Darunter fällt das Setzen eines Titels, eines Icons, Vor- und Zurück-Navigation innerhalb eines Fensters, gemeinsamer Zugriff auf die verschiedenen Daten, etc. .

User-Interfaces

Alle Screens / Benutzeroberflächen wurden – teilweise prototypisch – realisiert und mit den dazugehörigen Operationen verknüpft (z.B. öffnet ein Klick auf den Button „Spielhistorie“ die Benutzeroberfläche für die Spielhistorie, ein Klick auf „Neues Spiel“ öffnet die Benutzeroberfläche zur Erstellung eines neuen Spiels. Des Weiteren wurde eine Benutzeroberfläche für Entwickler entwickelt, der Debug-SceneViewer, mithilfe dessen jede weitere Benutzeroberfläche aufgerufen und getestet werden kann. Mithilfe des Debug-SceneViewers kann die Anwendung auch in verschiedene Test-Szenarien geschaltet werden um die Entwicklungsgeschwindigkeit zu erhöhen.

Initiale Asynchronität

Die Benutzeroberfläche für den Login und zur Registrierung neuer Spieler verwendet *Tasks* um die Darstellung des User-Interfaces nicht (durch warten oder andere Tätigkeiten des Systems) zu blockieren.

Initialer Konsum der Schnittstelle

Das Front-End nutzt bereits erste Endpunkte der Schnittstellen um beispielsweise mittels POST neue Spieler am System zu registrieren oder bestehende Spieler am System anzumelden. Des Weiteren ist es möglich alle laufenden Spielinstanzen einer Server-Instanz zu beziehen. Die Kommunikation erfolgt dabei auf Basis von HTTP.

Planung und Realisierung der Bewegungsberechnung der Snake-Implementierung

Die Berechnung von Positionen sowie die korrekte Darstellung einer Schlange wurde erdacht, geplant und umgesetzt. Diese Mechanik kann bereits im Einzelspieler-Modus der Anwendung getestet werden.

Snake-Implementierung – Mechanik - Wegfindung

Für die weitere Entwicklung der Spielmechaniken und das Testen verschiedener Aspekte wurde eine Wegfindung im Spiel realisiert welche auf zufälligen berechneten Eingaben basiert. Dies ermöglicht das Spielen ohne menschliche Gegner (NPCs). Die Wegfindung funktioniert nach dem Prinzip, dass eine Schlange möglichst lange am Spiel teilnimmt ohne mit einer anderen Schlange oder Wänden zu kollidieren.

Integration der Design Patterns

Nach Überarbeitung der UML-Diagramme sollen gemäß der Anforderungen Entwurfsmuster implementiert werden. Die Implementierung des Factory-Pattern könnte bspw. Für die Erzeugung der Entitäten (Food, Power-Ups, etc.) auf dem Spielfeld verwendet werden. Das Composite-Pattern könnte ebenfalls für diese Entitäten (Food, Power-Ups, etc.) verwendet werden – da verschiedene Entitäten im Endeffekt dasselbe Verhalten realisieren (ein Spieler sammelt ein solches ein, eine bestimmte Aktion passiert). Das Observer-Pattern wird bereits im Front-End verwendet – es wäre allerdings auch denkbar weitere Punkte in der Architektur mit diesem Pattern zu versehen wie z.B. bei Highscores und auf deren Veränderung zu reagieren.

Überarbeitung der UML-Diagramme

Die Entwicklung verschiedener Systemaspekte zeigt auf, dass die UML-Diagramme (Klassendiagramme) noch verfeinert werden müssen, da die aktuelle Implementierung des Systems und die Inhalte der Diagramme auseinander gelaufen ist.

Ausstehende Arbeitspakete

Die folgenden Arbeitspakete werden in den folgenden Wochen realisiert.



Abbildung 4 - Gantt-Diagramm zur Visualisierung der verbleibenden Arbeitspakete

Absicherung der Schnittstelle (API) mittels Spring Security und JSON-Web-Tokens

Im aktuellen Stand der Entwicklung ist die Schnittstelle nicht gegen unautorisierte Zugriffe geschützt d.h. jeder http-fähige Client ist in der Lage Informationen aus dem Backend des Systems zu erhalten. Dies soll verhindert werden, in dem Spring Security für die allgemeine Autorisierung, Authentifizierung und Authentisierung implementiert verwendet werden soll. Des Weiteren soll jede Schnittstellen-Anfrage nur mit einem gültigen JSON-Web-Token möglich sein.

Vollständige Implementierung der Klassenrelationen

Im aktuellen Stand der Entwicklung sind noch nicht alle Klassen – und deren Relationen zu anderen Klassen – vollständig implementiert, so wie im Klassendiagramm eigentlich ersichtlich.

Vollständiger Konsum der Schnittstellen

Im aktuellen Stand der Entwicklung ist noch nicht jede Funktionalität in Betracht auf die Nutzung der Schnittstelle vollständig realisiert. Bereits möglich ist die Registrierung neuer Benutzer sowie der Bezug der aktuell laufenden Spiel-Instanzen. Der Bezug der Spielhistorie bspw. ist noch nicht realisiert.

Austausch der Spielereingaben (Implementierung von Web-Sockets)

Spiel-Instanzen können aktuell in verschiedensten Szenarien lokal auf einem PC ausgeführt werden. Die Anforderung, dass mehrere Spieler an einem Spiel teilnehmen können und somit Spielereingaben an das Backend gesendet und von dort aus an alle Clients repliziert werden ist noch nicht realisiert worden.

Komplexität des GameControllers reduzieren

Durch Abstraktion und Refactoring könnte die Komplexität des Snake-Spiels reduziert werden – dieser Aspekt wird im Laufe des Projekts weiter vorangetrieben.

Regelwerk ausbauen

Die Spieler sollen die Möglichkeit bekommen das Snake Spiel in verschiedenen Ausprägungen spielen zu können. Dafür soll ein Regelwerk definiert werden welches bei der Definition eines Spiels genutzt werden kann um die grundlegenden Mechaniken eines Spiels festzulegen:

- Kollision mit Wand führt zu GameOver
- Es gibt keine Wände
- Spieler können sich nicht gegenseitig abbeißen / Spieler können sich gegenseitig abbeißen
- ...

Feedback aus der Präsenz

Vorschlag – Game-Controller Auslagerung

Da wir bis zur Präsentation noch keine Realisierung im Bereich der Datenübertragung (WebSockets) durchgeführt haben kann man zu diesem Zeitpunkt noch nicht von Vorteilen eines Refactorings sprechen. Die Aussage „Spiel läuft voraussichtlich flüssiger“ ergibt zu dem jetzigen Zeitpunkt keinen Sinn.

RespawnSnake() → Raus

Diese Methode wird für Test-Zwecke der KI genutzt – für eine ergänzende Nutzung der Funktion könnte man überlegen ob ein Food oder ein Power-Up nicht dafür sorgt, dass ein Respawn eines Spielers durchgeführt wird.

Draw-Funktionalität in Klassen auslagern.

In welche Klassen auslagern? – falls gemeint ist, dass Draw-Methoden in Model-Klassen ausgelagert werden soll halte ich das für eine wenig sinnvolle Vorgehensweise, da das Projekt die MVC-Architektur nutzt – es wäre denkbar die Funktionalität in weitere Controller-Klassen aufzuteilen oder das Spiel gar mithilfe eines ECS (Entity-Component-System) zu realisieren, dies würde jedoch den zeitlichen Rahmen des Projekts übersteigen.

Szenen-Wechsel

Der vorherige Stand des Front-Ends enthielt keinerlei Logik zur Verwendung desselben Fensters in der Anwendung. Das Front-End enthielt jedoch eine gemeinsame Controller-Klasse welche bereits verschiedene gemeinsame Aufgaben bzgl. der UI übernimmt (setzen des Fenster-Titels, Wechsel des Fenster-Icons, etc.) – im Paket `src/main/java/de/ostfalia/teamx/controller/BaseController.java` – diese BaseController-Klasse wurde, so wie von der Gruppe 1 vorgeschlagen dahingehend erweitert, dass anstelle eine neuen Stage die aktuelle Stage wiederverwendet und lediglich die darin enthaltene Szene (Scene) ausgetauscht wird.

```
public void showLayout(String layoutName, String newTitle) {  
  
    FXMLLoader fxmlLoader = new FXMLLoader();  
    fxmlLoader.setLocation(getClass().getClassLoader().getResource(layoutName));  
    Parent root = null;  
  
    try {  
  
        // load the fxml by the given layoutName  
        root = fxmlLoader.load();  
  
        // get a reference to its corresponding controller  
        BaseController baseController = fxmlLoader.getController();  
  
        // create a new stage, set the newly loaded layout as the root element of it  
        Stage window = new Stage();  
        Scene scene = new Scene(root);  
        window.setScene(scene);  
  
        // setup the controller - reference it to the stage  
        baseController.currentStage = window;  
  
        // setup the title and the icon of the application  
        baseController.currentStage.getIcons().add(new Image(URI.create("icon.png")));  
        baseController.setTitle(newTitle);  
  
        // initialize everything related to the user-interface  
        baseController.postInitialize();  
  
        // show the setup and referenced window  
        window.show();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
}
```

```
public void showLayout(String layoutName, String newTitle) {  
  
    FXMLLoader fxmlLoader = new FXMLLoader();  
    fxmlLoader.setLocation(getClass().getClassLoader().getResource(layoutName));  
    Parent root = null;  
  
    try {  
  
        // load the fxml by the given layoutName  
        root = fxmlLoader.load();  
  
        // get a reference to its corresponding controller  
        BaseController baseController = fxmlLoader.getController();  
  
        // pass the reference of the current window stage to the newly instantiated controller  
        baseController.currentStage = currentStage;  
        currentStage.setScene(new Scene(root));  
  
        // setup the title and the icon of the application  
        baseController.currentStage.getIcons().add(new Image(URI.create("icon.png")));  
        baseController.setTitle(newTitle);  
  
        // initialize everything related to the user-interface  
        baseController.postInitialize();  
  
        // show the setup and referenced window  
        // window.show();  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}  
}
```

Abbildung 5 – Vergleich des Quelltests: Links bei Vorstellung (28.11.2020) – Rechts aktueller Stand

Anstelle der Instanziierung einer neuen Stage, einer neuen Scene und eines neuen Root-Elements wird also einfach die aktuelle Stage verwendet und das entsprechende Scene-Element ausgetauscht was zur Folge hat, dass die Anwendung unabhängig vom Layout nun stets im selben Fenster ausgeführt und dargestellt wird.

Design-Patterns

Im Zuge der Anforderungen sollen die folgenden Patterns evaluiert und realisiert werden.

Composite-Pattern

Die Anwendung des Composite-Pattern bietet sich an wenn Objekte durch Baum-Strukturen als Teil-Ganzes-Hierarchien zusammengefügt werden: „*Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly*“
- Design Patterns – Elements of reusable Object-Oriented Software, Seite 163, Composite-Pattern

Elemente welche von einem Client gleichartig behandelt werden könnten in der Realisierung des Snake-Spiels die Power-Ups und Food-Elemente sein. Diese Objekte haben einen Vektor mit zwei Komponenten (x,y), ein darzustellendes Bild / Bitmap und einen abstrakten nicht konkreten Effekt gemeinsam.

Die folgende Abbildung zeigt einen Entwurf für den Einsatz des Pattern:

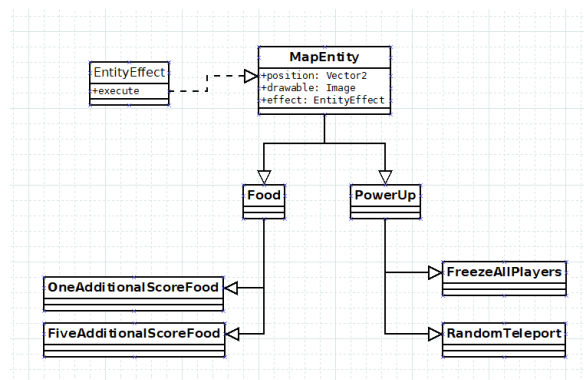


Abbildung 6 - Darstellung des Composite-Pattern als UML-Diagramm.

Wie in dem UML-Diagramm dargestellt könnte eine Klasse (z.B. namens MapEntity) realisiert werden welche alle Gemeinsamkeiten besitzt. Im Anschluss daran könnten die abstrakten Oberklassen Food und Power-Up realisiert werden um im Anschluss daran konkrete Implementierungen der Foods oder Power-Ups zu realisieren.

Auf diese Weise müsste nicht zwischen verschiedenen Implementierungen unterschieden werden sondern man könnte von einem abstrakten Typ sprechen. Das Food würde als konkreten Effekt beim Einsammeln eine Erhöhung des Spieler-Highscores auslösen. Die Power-Ups hingegen würden Zustandsveränderungen beim jeweiligen Spieler oder allen anderen Spielern zur Folge haben – bspw. der Teleport an eine andere Stelle auf dem Spielfeld oder, dass alle Spieler „gefroren“, also „eingefroren“ werden und für eine bestimmte Zeit keine Eingaben mehr tätigen können.

Factory-Pattern

Das Factory-Pattern könnte im selben Kontext wie das Composite-Pattern verwendet werden. Anstelle der Unterscheidung würde die Verwendung im Rahmen der Objekt-Erzeugung stattfinden. Eine Factory-Instanz würde dann genutzt werden um verschiedene MapEntities zu erzeugen.

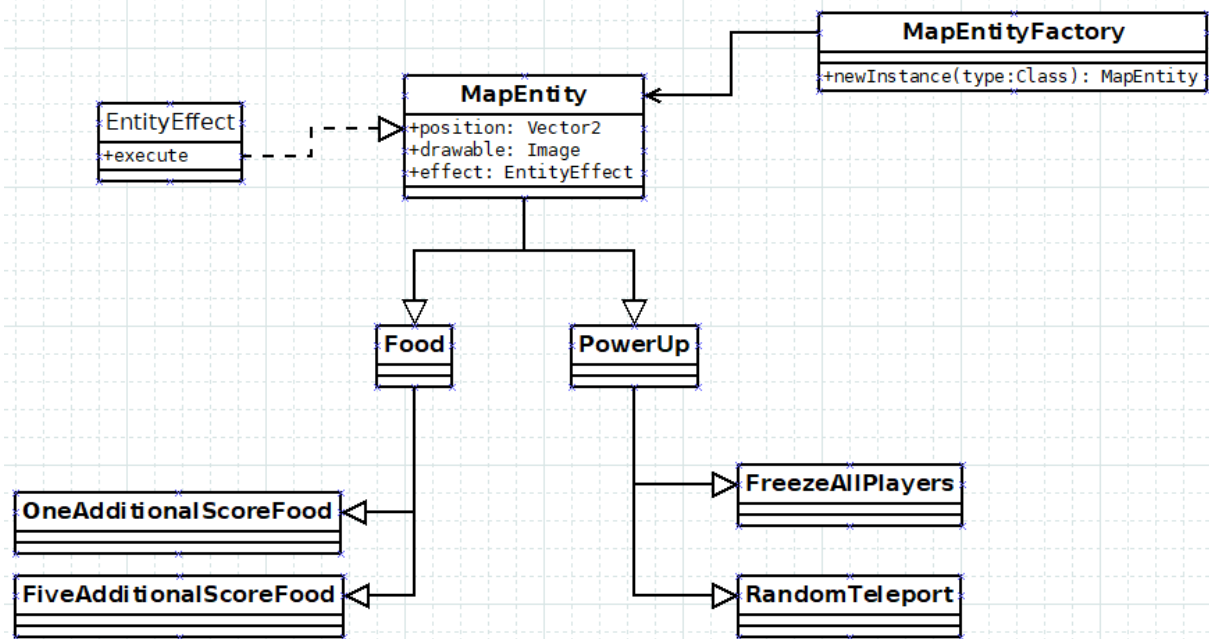


Abbildung 7 - Integration des Factory-Pattern.

Das Factory-Pattern wird realisiert in dem eine / verschiedene Factory-Ausprägungen implementiert werden. Jede dieser Factory-Implementierungen besitzt mindestens eine Methode „newInstance“ – welche als einziges Übergabe-Parameter den konkreten Klassentyp der zu erzeugenden Objekt-Instanz besitzt.

Anhand dieses Parameters wird der Factory mitgeteilt welches Objekt instanziiert werden soll. Die konkrete Instanziierung wird dann in der jeweiligen Factory definiert – zu den zu definierenden Parametern gehört eine Position (Vector2), der übergebene Typ (Type) sowie eine Grafik (Drawable) um die PowerUps, das Food, oder noch zu definierende Spiel-Entitäten wie zum Beispiel Wände oder Teleporter, zu instanziiieren.

Des Weiteren könnten auch die Spieler (die Schlangen) via Factory-Methode erzeugt werden – dies würde allerdings keinen konkreten Mehrwert liefern, da in der aktuellen Variante des Projektstandes nur eine Ausprägung einer Schlange existiert.

Observer-Pattern

Das Observer-Pattern gehört zu der Kategorie der Verhaltensmuster und dient, wie der Name vermuten lässt, der Observierung / Beobachtung eines Objekts oder Eigenschaften eines oder mehrerer Objekte.

Das Observer-Pattern wird bereits explizit in verschiedenen Szenarien des Projekts verwendet – beispielsweise zur Kontrolle der Fenster-Dimensionen – sobald sich diese verändern werden diese Änderungen an die Szenen weitergegeben, womit diese sich an die neuen Dimension des Fensters anpassen können.

Ein geeigneter Einsatzzweck des Observer-Pattern für das Snake-Spiel wäre eine kontinuierliche Beobachtung des Spielzustandes – hier wird bei jedem Tick (Ausführung der Update-Methode) der Zustand des aktuellen Spiels überprüft – solange keine Spielregel erfüllt ist (zum Beispiel die Erreichung eines Highscores, alle Teilnehmer sind ausgeschieden, o.ä.) läuft das Spiel ohne Zustandsveränderung weiter – sobald eine oder mehrere Regeln eintreten gilt das Spiel als beendet – an dieser Stelle wird

ein entsprechendes UI-Element eingeblendet (als Spieler x ist der Gewinner) sowie ein entsprechender Eintrag in der Spielhistorie hinterlegt.

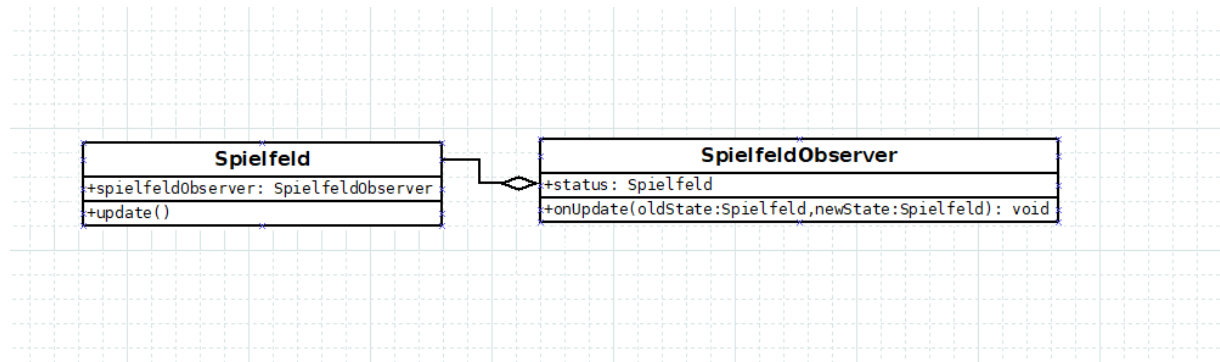


Abbildung 8 - Implementierung des Observer-Patterns

Bei jedem Aufruf der `update()`-Methode wird der aktuelle Zustand des Spielfelds übergeben – da der **SpielfeldObserver** bereits den letzten Zustands des Spielfelds kennt kann des Weiteren ein Unterschied zwischen dem aktuellen Zustand und dem letzten Zustand erkennen – dies ist Hilfreich um zu Erkennen ob eine Zustandsveränderung bereits eingetreten ist (zum Beispiel der erste Zustand wann eine Spielrunde als beendet gilt, etc.).

Anhang

UML – Anwendungsfalldiagramm

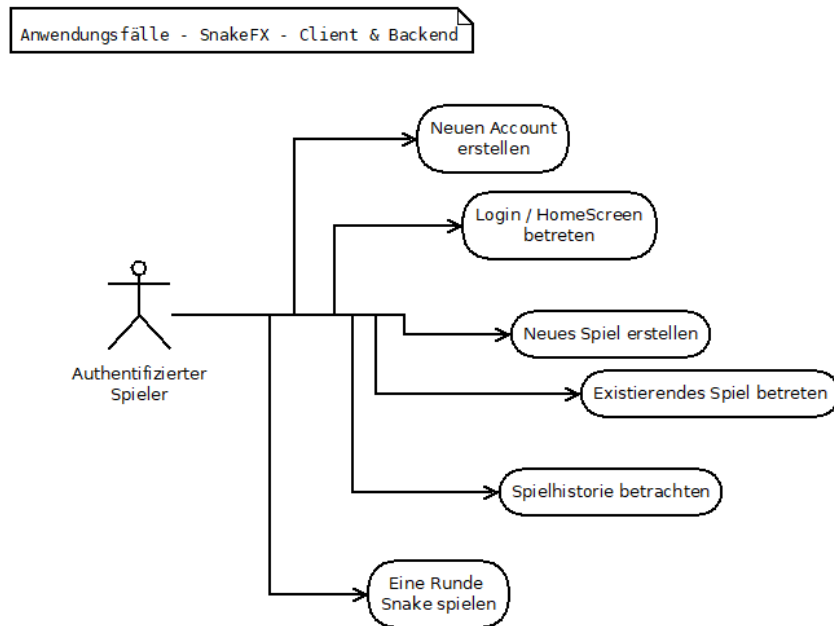
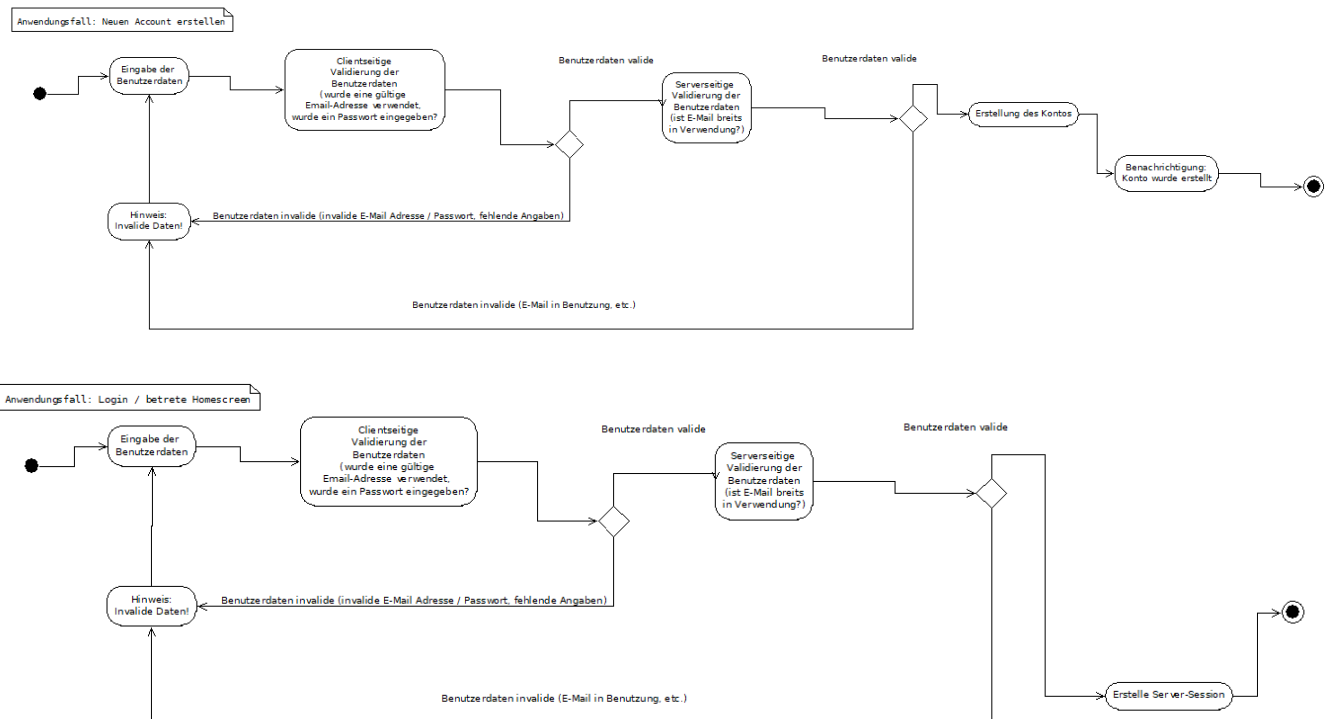
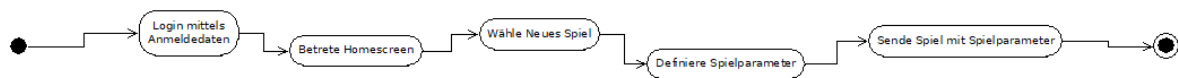


Abbildung 9 – Anwendungsfalldiagramm des P+F-Projekts

UML – Aktivitätsdiagramm



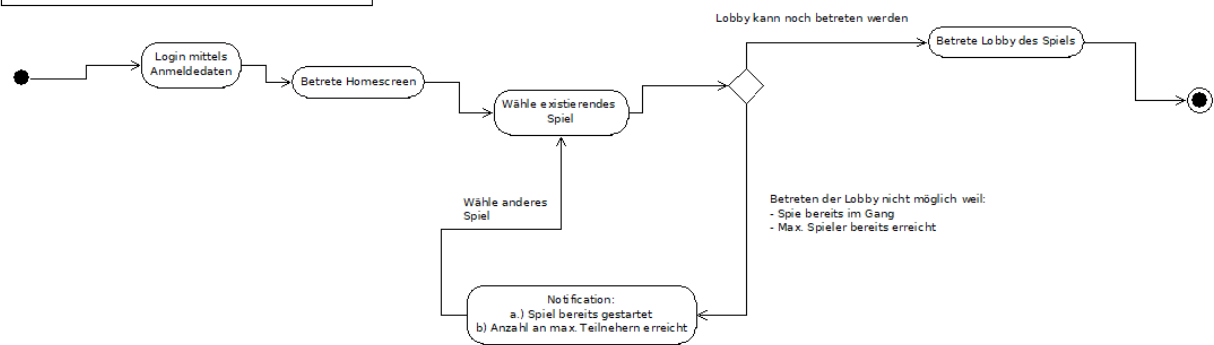
Anwendungsfall: Erstelle neues Spiel



Fragen:

- Kann hier noch irgendwas schiefgehen?
- Wenn ja, muss das in dieser Art von UML-Diagramm erwähnt werden?

Anwendungsfall: Existierendes Spiel in Lobby betreten



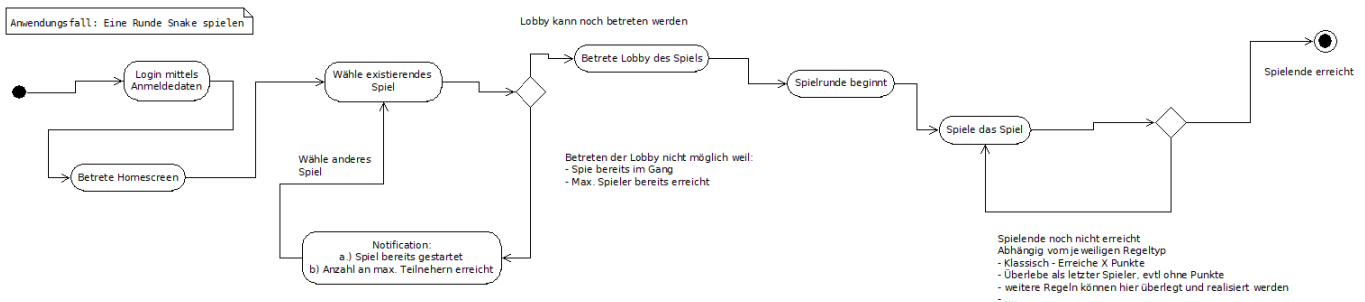
Anwendungsfall: Betrachte Spielhistorie



Fragen:

- Kann hier noch irgendwas schiefgehen?
- Wenn ja, muss das in dieser Art von UML-Diagramm erwähnt werden?

Anwendungsfall: Eine Runde Snake spielen



User-Interface – Frontend - SnakeFX

Die folgenden Abbildungen zeigen das Front-End des Projekts.

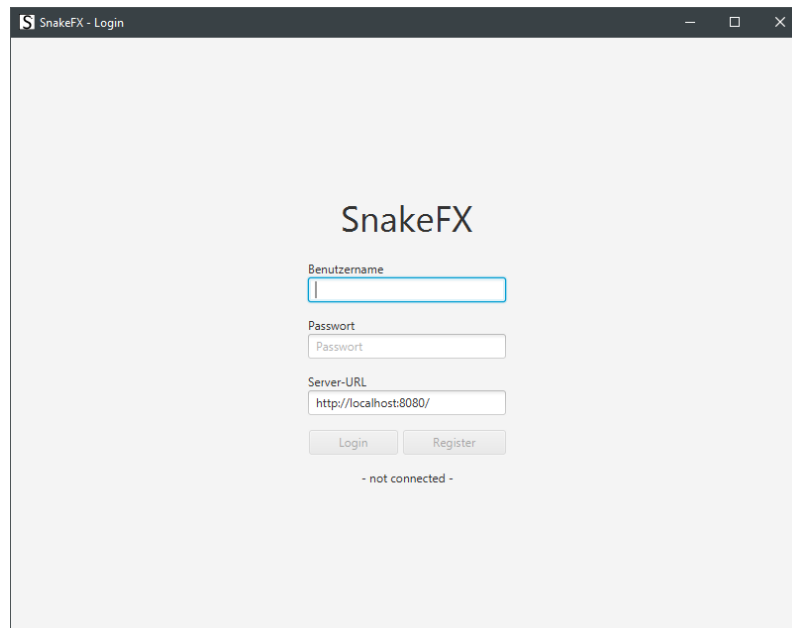


Abbildung 10 - Login- / Register-Screen der Anwendung

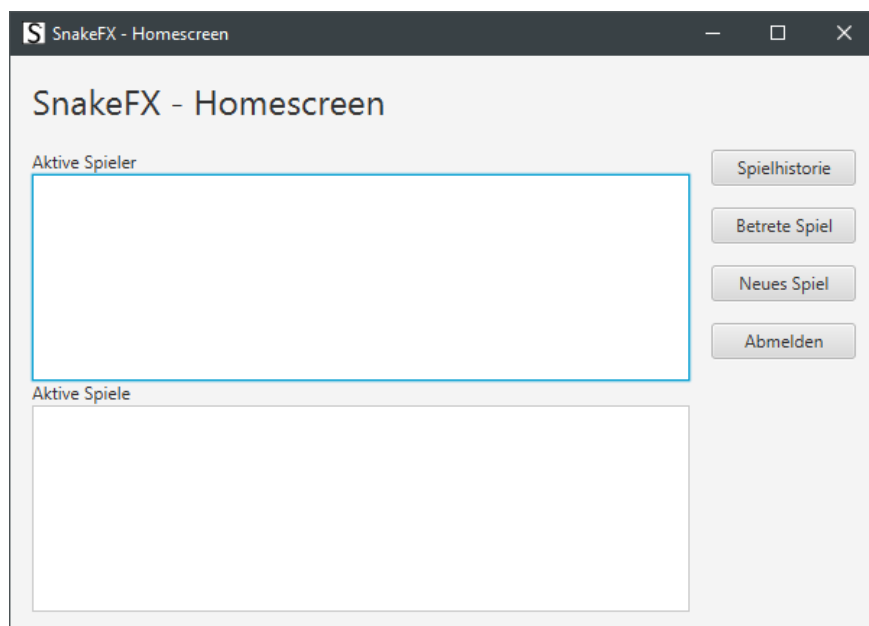


Abbildung 11 - Homescreeen der Anwendung - hier werden alle aktiven Spieler und alle aktiven Spiele aufgelistet.

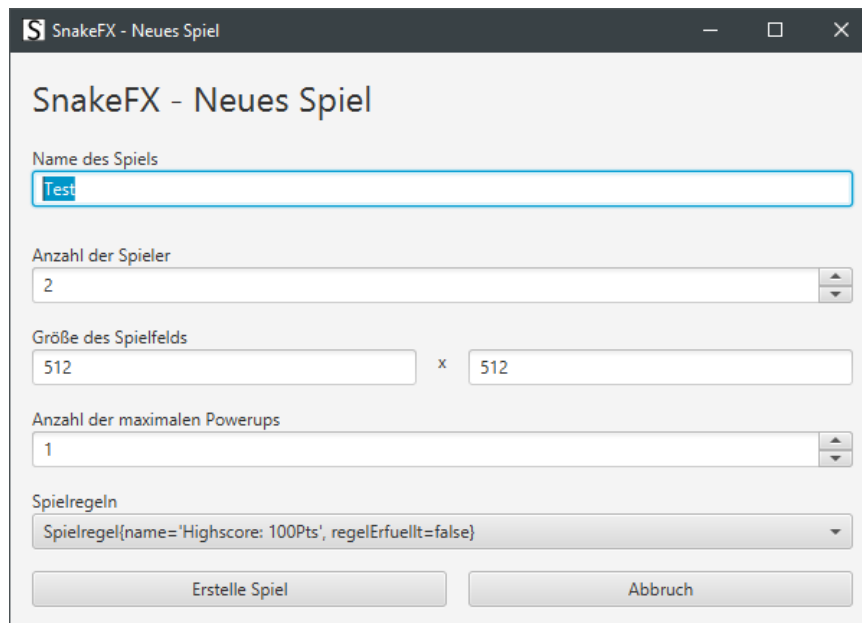


Abbildung 12 - New-Game Screen - hier können neue Spiele definiert und im Anschluss in der Lobby angemeldet werden.

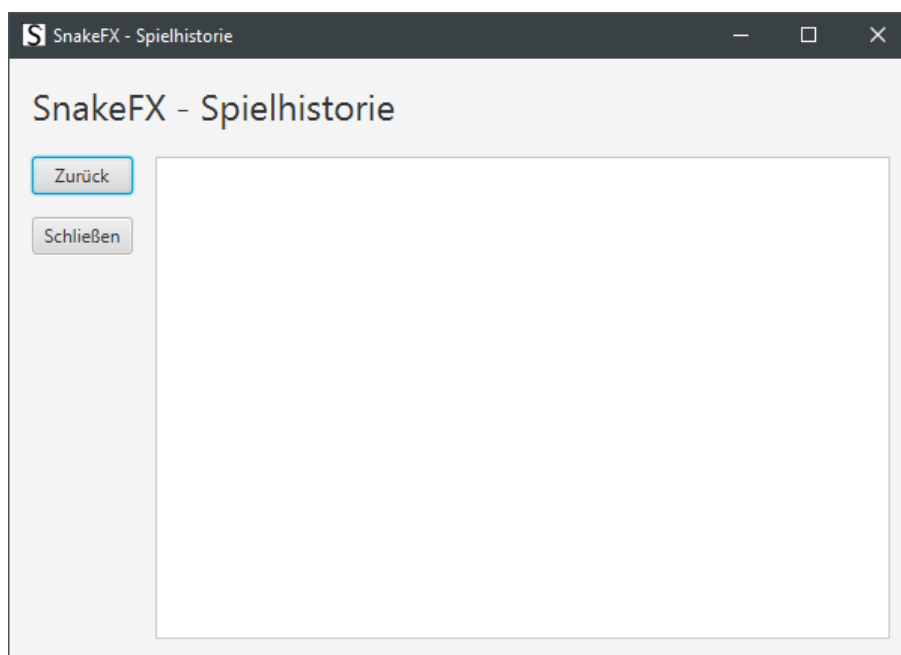


Abbildung 13 - Spielhistorie - In dieser Benutzeroberfläche können alle in der Vergangenheit gespielten Spiel betrachtet werden.

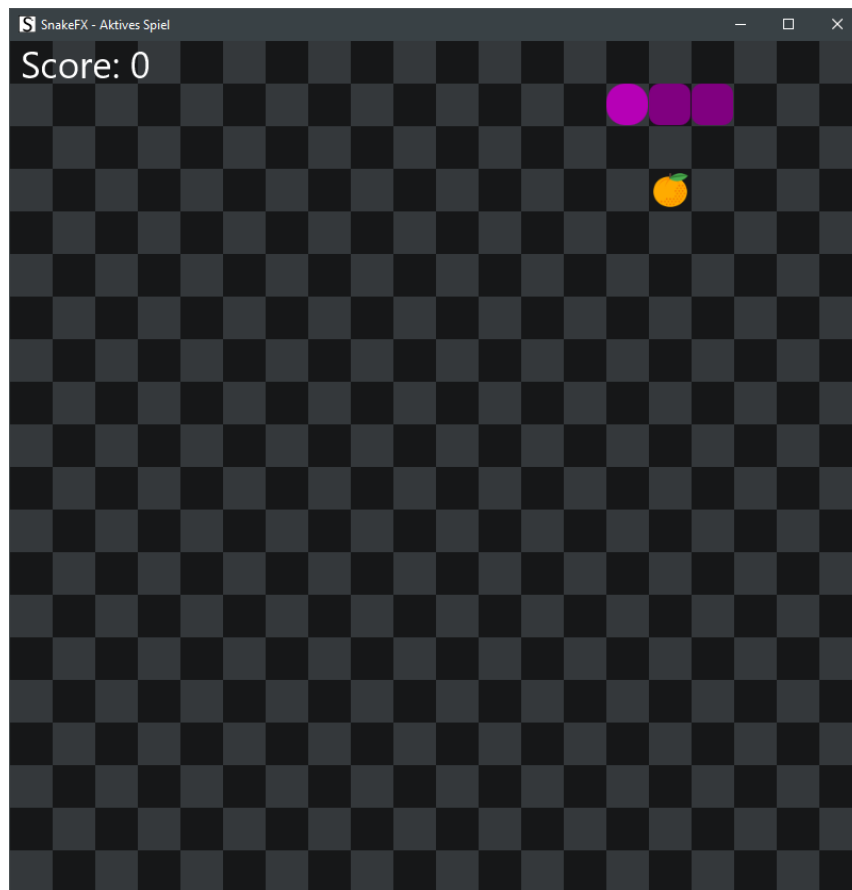


Abbildung 14 - Im aktiven Spiel wird die Implementierung des Snake-Spiels ausgeführt. Die Abbildung enthält das Spielbrett, sowie die Implementierung der Food-Mechanik sowie eine einfache KI mit zufälliger Wegfindung.

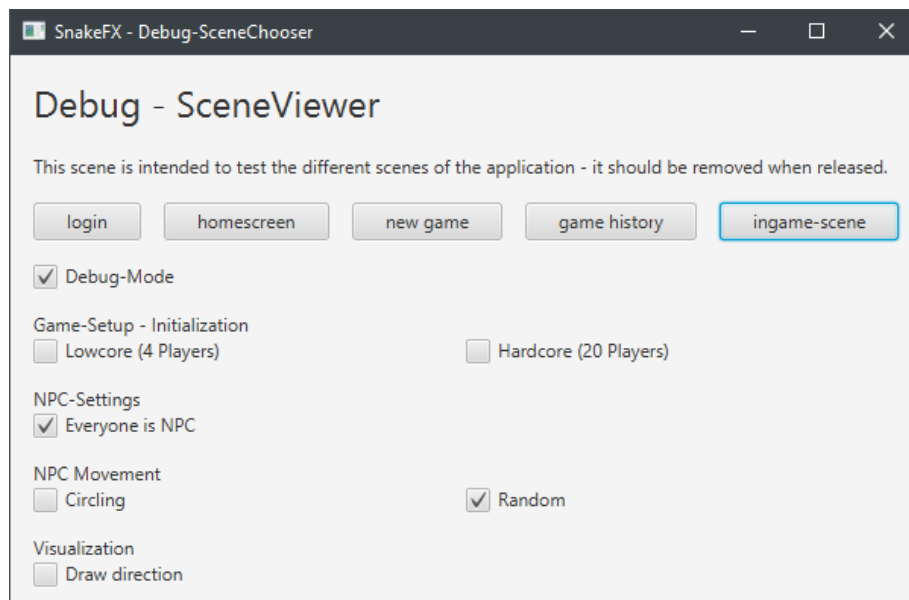


Abbildung 15 - der Debug-SceneViewer ermöglicht die direkte Initialisierung der einzelnen Benutzeroberflächen ohne vollständige Interaktion mit der Anwendung. Des Weiteren kann das Snake-Spiel über diese Benutzeroberfläche in verschiedene Test-Szenarien versetzt werden.