SnakeDart

Referenzspiel und -dokumentation für das Modul "Webtechnologie Projekt"

Prof. Dr. Nane Kratzke

SoSe 2016

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Programm-Listings

1 Einleitung

Am Beispiel einer Spielentwicklung sollen sie eine Auswahl relevanter Webtechnologien wie bspw. **HTML**, **DOM-Tree**, **HTTP** Protokoll, **REST**-Prinzip sowie die Trennung in **client**- und **serverseitige Logik** spielerisch kennenlernen. Das Spiel selber ist dabei weitesgehend clientseitig zu realisieren und soll nur für die Speicherung von Spielzuständen, wie bspw. Highscores, auf REST-basierte Services zugreifen. Selbst ohne Storagekomponente soll ihr Spiel spielbar sein (einzig und allein das Speichern von Spielzuständen ist natürlich eingeschränkt).

Bei der Suche nach geeigneten Spielideen lohnt es sich immer mal wieder bei alten Arcade Klassikern wie Tetris, Space Invader, Pong, etc. zu suchen und sich inspirieren zu lassen. Um Komplexität und Aufwand im Griff zu behalten, ist ihnen eine REST-basierte Storagekomponente vorgegeben. Sie sollen diese nur in Dart nachimplementieren. Ferner soll ihr Spiel dabei auf einem zweidimensionalen Raster basieren und keinen Canvas zur Darstellung nutzen. Sie sollen ein Single Player Game schreiben. Um die Komplexität nicht im Rahmen zu halten, sollten sie daher mit Ansätzen vorsichtig sein, die auf Multi Player Games beruhen und Player durch "künstliche Intelligenzen" ersetzen. Dies kann sehr schnell zu erheblicher Komplexität führen, denken Sie nur mal an Schach!

Ihr Spiel soll dabei auf den Einsatz von Canvas verzichten, sondern ausschließlich auf DOM-Tree Manipulationen beruhen, um den Spielzustand darzustellen. Das Spiel sollte daher auf einem einfach 2D Raster spielbar sein[1]. Zum einen reduziert dies die Komplexität möglicher Spielkonzepte, zum anderen sollen sie durch die Aufgabenstellung dazu gezwungen werden, sich intensiv mit dem DOM-Tree eines HTML-Dokuments und dessen clientseitiger Manipulation auseinander setzen. Dies würden sie nicht, wenn Sie Canvas nutzen würden (sie würden dann sich intensiv mit Grafikbibliotheken und ggf. Game Engines auseinandersetzen, das ist jedoch nicht Ziel dieser Veranstaltung[2]).

Sie werden sehen, dass sich einfache bis mittelkomplexe Spiele auch mit absoluten Web-Basistechnologien, die jeder moderne Webbrowser anbietet, realisieren lassen. Ich hoffe sie erhalten dadurch ein besseres Gefühl zwischen der Trennung von client- und serverseitiger Logik, sowie was mit Webtechnologien grundsätzlich alles machbar ist.

Hinweise für die Durchführung ihres Webtechnologie Projekts sind in vorliegender Dokumentation in folgender Form gekennzeichnet.

> **Hinweis:** Solche Boxen kennzeichnen Hinweise für ihr Webtechnologie-Projekt.

Es gilt z.B. der folgende Hinweis:

> **Hinweis:** Vorliegende Dokumentation erläutert einerseits den Aufbau eines Beispielspiels (SnakeGame). In zweiter Funktion dient die Dokumentation gleichermaßen als Anhalt für Sie, um Ihr eigenständig entwickeltes Spiel nachvollziehbar zu dokumentieren.

Diese Dokumentation erläutert die durch Ihr Spiel zu erfüllenden Anforderungen und das Beispielspielkonzept des Beispielgames **SnakeGame** im Kapitel 2. Die Umsetzung des Spielkonzepts in eine Model-View-Controller basierte Architektur wird im Kapitel 3 dargestellt. In diesem Kapitel wird auch die REST-basierte Storagekomponente erläutert, die die klassische MVC-Architektur um eine Storagekomponente erweitert. Exemplarisch wird ferner der sinnvolle Einsatz von UML-Diagrammen gezeigt, um angewendete Gestaltungsprinzipien der Software zu veranschaulichen. Das Kapitel 4 befasst sich mit dem Einsatz von deskriptiven Dateiformaten, um Spiellevel und Spielparameter zu definieren. Auf den ersten Blick wirkt dies nach Mehraufwand. Sie werden jedoch sehen, dass es durchaus Sinn machen kann, Parameter nicht hardcodiert im Spiel einzubauen, sondern diese in externe Konfigurationsformate auszulagern. Insbesondere in der Schlussphase werden sie es ggf. zu schätzen wissen, dass Sie Spielparameter unabhängig vom Code ändern können, um die Spielbarkeit ihres Spiels zu optimieren. Im Kapitel 5 geht es um den Nachweis der Anforderungen, den sie systematisch

[1] Das Raster soll dabei nicht zu groß werden. Im Extrem könnten sie ansonsten 1 Pixel große Felder definieren und sich künstlich ein Canvas Element "nachbauen".

[2] Wenn Sie so etwas interessiert überlegen Sie das Wahlpflichtmodul "Game Programming" des Studiengangs ITD zu belegen.

(wenn auch aufgrund der Kürze der Zeit im wesentlichen nur argumentativ) Anforderung für Anforderung erbringen sollen. Es bietet sich an, dies nicht erst am Ende ihrer Spieleentwicklung zu machen, sondern begleitend. **Insbesondere da dieses Kapitel sehr viele und relevante Informationen enthält, die im Rahmen der Notenfindung herangezogen werden.**

Auch wenn es nur ein kleines Spiel ist, das sie entwickeln sollen, werden sie im Verlaufe dieses Projekts die typischen Phasen der Softwareentwicklung von der Anforderungserhebung, über Architektur und Implementierung bis zur Nachweisführung durchlaufen und dokumentieren.

Ich wünsche Ihnen nun viel Erfolg und Spaß bei der Entwicklung ihres Webgames. Die besten Spiele der vergangenen Jahre finden Sie in einer Hall-of-Fame unter folgendem Link `http://www.nkode.io/assets/webtech-hall-of-fame/index.html`. Auch ihr Spiel kann in diese Hall-of-Fame eintreten. Dafür gelten die folgenden Regeln:

1. Ihr Team muss das Modul mit einer guten oder sehr guten Note abschließen.
2. Sie müssen erlauben, dass ihr Spiel der Öffentlichkeit zur Verfügung gestellt werden darf.
3. Die fünf besten Spiele werden in der Google+ Dartisans Community zu einer öffentlichen Abstimmung eingestellt.
4. Die Google+ Dartisans Community stimmt über die besten Spiele ab.
5. Die drei best bewerteten Spiele (Stimmenanzahl) werden in die "Hall of Fame" aufgenommen.
6. Das **beste Spiel** des Semesters erhält einen **Notenbonus**.

2 Anforderungen und abgeleitetes Spielkonzept

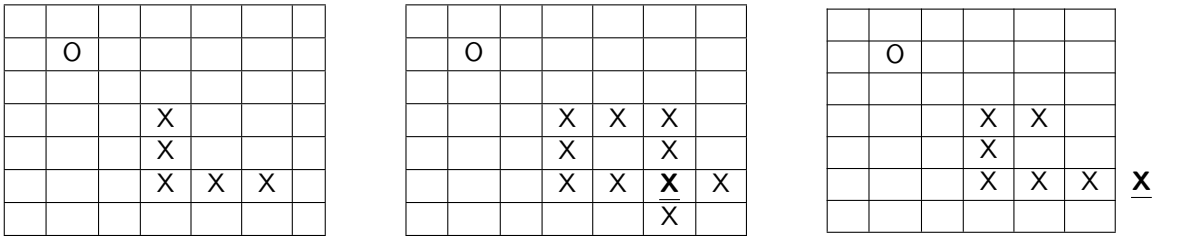
Im Verlaufe des Webtechnologie Projekts sollen Sie ein clientseitiges Spiel entwickeln, welches in einem Webbrowser spielbar sein soll. Das Spiel kann sich softwaretechnisch am hier dokumentierten **SnakeGame** (Github: `https://github.com/nkratzke/dartsnake`) orientieren. Sie sollen jedoch ein eigenständiges Spielkonzept entwickeln und umsetzen.

2.1 Anforderungen

Ihr Spiel soll folgende Anforderungen erfüllen. Sie sollen dabei nachvollziehbar dokumentieren, wie diese Anforderungen durch ihre Realisierung umgesetzt werden. Das Spiel soll folgende in Tabelle 1 aufgeführten **funktionalen Anforderungen**, **Dokumentationsanforderungen** und **technischen Randbedingungen** erfüllen.

Id	Kurztitel	Anforderung
AF-1	Einplayer Game	Das Spiel soll ein Einplayer Game sein *(Mehrplayer Konzepte können als Einplayer Game realisiert werden, wenn Spieler durch "künstliche Intelligenzen" gesteuert werden. Beachten Sie dabei bitte, dass abhängig vom Spielkonzept die Komplexität des Spiels erheblich steigt, denken sie bspw. an Schach. Es bietet sich an, sich von alten Arcade Klassikern inspieren zu lassen.)*
AF-2	2D Game	Das Spiel soll konzeptionell auf einem 2D-Raster basieren.
AF-3	Levelkonzept	Das Spiel sollte ein Levelkonzept vorsehen.
AF-4	Parametrisier-ungskonzept	Das Spiel sollte ein Parameterisierungskonzept für relevante Spielparameter vorsehen.
AF-5	REST-basierter Storage	Das Spiel soll auf einen JSON-basierte Storageservice zugreifen können, um Spielzustände (z.B. Highscores und ähnliches) speichern zu können. Das Spiel muss bei nicht erreichbarem Storage-Service (Gamekey Service) weiterhin spielbar sein. Einzig und allein die Speicherfunktionalitäten dürfen natürlich eingeschränkt sein. *(Für den Storage-Service (Gamekey-Service) gibt es eine Referenz-Implementierung:* `https://bitbucket.org/nanekratzke/gamekey`*)*
AF-6	Desktop Browser	Das Spiel muss in Desktop Browsern spielbar sein.
AF-7	Mobile Browser	Das Spiel muss auf SmartPhone Browsern spielbar sein.
		Dokumentationsanforderungen
D-1	Dokumenta-tionsvorlage	Die Dokumentation soll sich an vorliegender Vorlage orientieren.
D-2	Projekt-dokumentation	Das Spiel muss geeignet dokumentiert sein, so dass es von projektfremden Personen fortgeführt werden könnte.
D-3	Quelltext-dokumentation	Der Quelltext des Spiels muss geeignet dokumentiert sein und mittels schriftlicher Dokumentation erschließbar und verständlich sein.
D-4	Libraries	Alle verwendeten Libraries sind aufzuführen und deren Notwendigkeit zu begründen.
		Technische Randbedingungen
TF-1	No Canvas	Die Darstellung des Spielfeldes sollte ausschließlich mittels DOM-Tree Techniken erfolgen. Die Nutzung von Canvas-basierten Darstellungstechniken ist **explizit** untersagt.
TF-2	Levelformat	Level sollten sich mittels deskriptiver Textdateien definieren lassen (z.B. mittels CSV, JSON, XML, etc.), so dass Level-Änderungen ohne Sourcecode-Änderungen des Spiels realisierbar sind.
TF-3	Parameterformat	Spielparameter sollten sich mittels deskriptiver Textdateien definieren lassen (z.B. mittels CSV, JSON, XML, etc.), so dass Parameter-Änderungen ohne Sourcecode-Änderungen des Spiels realisierbar sind.
TF-4	HTML + CSS	Der View des Spiels darf ausschließlich mittels HTML und CSS realisiert werden.
TF-5	Gamelogic in Dart	Die Logik des Spiels muss mittels der Programmiersprache Dart realisiert werden.
TF-6	Storagelogic in Dart	Der Storageservice muss REST-basiert sein und mittels der Programmiersprache Dart realisiert werden.
TF-7	Storage Referenz	Das Spiel soll gleichermaßen mit der Referenzimplementierung des Storageservice als auch der Eigenrealisierung des Storageservice funktionieren.
TF-8	Storage Referenztest	Die Eigenrealisierung des Storageservices muss den Referenztest der Referenzimplementierung implementieren. *(Nicht erfolgreiche Testfälle sind explizit auszuweisen, zu dokumentieren und zu begründen!)*
TF-9	Browser Support	Das Spiel muss im Browser Chromium/Dartium (native Dart Engine) funktionieren. Das Spiel muss ferner in allen anderen Browserrn (JavaScript Engines) ebenfalls in der JavaScript kompilierten Form funktionieren (geprüft wird ggf. mit Safari, Chrome und Firefox).
TF-10	MVC Architektur	Das Spiel sollte einer MVC-Architektur folgen.
TF-11	Erlaubte Pakete	Erlaubt sind alle dart:* packages, sowie das Webframework start.
TF-12	Verbotene Pakete	Verboten sind Libraries wie Polymer oder Angular. *(Sollten Sie Pakete verwenden wollen, die außerhalb der erlaubten Pakete liegen, holen Sie sich das Go ab, begründen sie bitte, wieso sie das Paket benötigen).*
TF-13	No Sound	Das Spiel muss keine Sounds unterstützen.

Tabelle 1: Anforderungen



(a) Spielfeld (b) Verknotete Schlange (c) Schlange außerhalb des Spielfeldes

Abbildung 1: Spielprinzip von SnakeGame

2.2 Spielkonzept des Snake Games

Das SnakeGame dient im wesentlichen als Anschauungsgegenstand für Sie und deckt wesentliche (aber nicht alle) aufgestellten Anforderungen (vgl. Kapitel 5) ab. Konzeptionell basiert es auf einem quadratischen n x n Spielfeld, auf dem sich eine Maus (O) und eine Schlange (X) wachsender Länge befindet. Ein Spieler kann eine Schlange (X) über das Spielfeld mittels Betätigen von Cursortasten bewegen. Ziel der Schlange ist es, möglichst viele Mäuse zu fangen. Hierzu muss der Spieler die Schlange zu einer Maus bewegen. Berührt der Kopf der Schlange die Maus, frisst die Schlange die Maus und die Schlange wird um ein Element länger. Es wird in diesem Fall per Zufall eine neue Maus auf dem Spielfeld erzeugt. Das Spiel wird mit jeder gefressenen Maus schwieriger, denn

- die Länge der Schlange erhöht sich mit jeder gefressenen Maus um eins,
- die Geschwindigkeit der Schlange erhöht sich ebenfalls mit jeder gefressenen Maus.

Die Schlange kann sich dabei nur vorwärts bewegen (d.h. nicht nach hinten kriechen oder zu einer der beiden Seiten rollen). Die Steuerung der Schlange wirkt nur auf den Kopf. Die restlichen Elemente des Schlangenkörpers folgen einander.

Das Spiel ist beendet, wenn einer der beiden folgenden GameOver Bedingungen eintreten:

- Die Schlange "verknotet" sich (d.h. mindestens zwei Elemente des Schlangenkörpers liegen auf demselben Element des Spielfeldes, vgl. Abb. 1(b))
- Die Schlange verlässt das Spielfeld (d.h. mindestens ein Element der Schlange befindet sich außerhalb des Spielfeldes, vgl. Abb. 1(c)).

SnakeGame kann auf vielfältige Arten variiert werden. Z.B. könnte

- mit mehr als einer Maus gespielt werden,
- Mäuse könnten sich bewegen,
- Mäuse könnten ihre Geschwindigkeit variieren,
- ein Teil der Mäuse könnte Tollwut haben (vergiftet sein, neue Game Over Bedingung)
- usw.

Die nachfolgende Architektur ist so gestaltet, dass solche Variationen nachträglich ergänzt werden könnten.

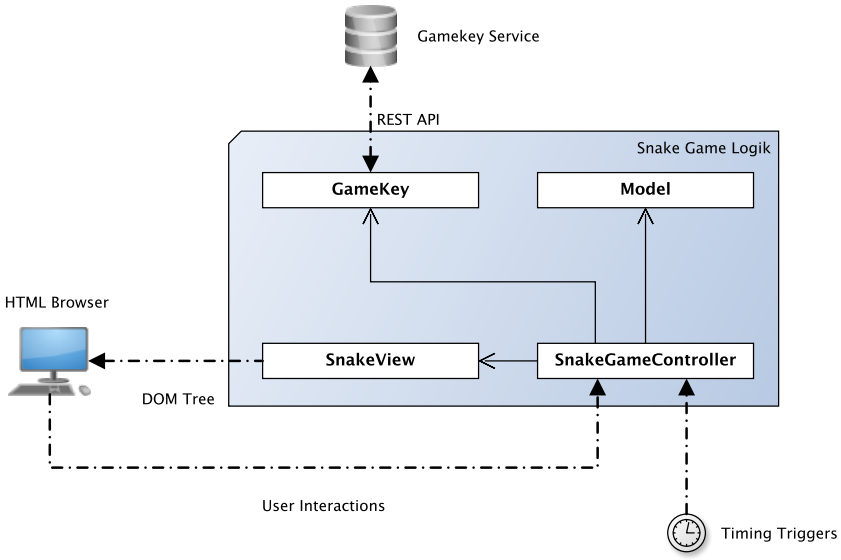


Abbildung 2: Architektur

3 Architektur und Implementierung

Abbildung 2 zeigt die Architektur von SnakeGame im Überblick. Die Architektur folgt dem bewährten Model-View-Controller Prinzip, ist jedoch um eine REST-basierte Storage Anbindung ergänzt. Softwaretechnisch gliedert sich die Spiellogik so in mehrere Komponenten (Klassen) mit spezifischen funktionalen Verantwortlichkeiten. Eine zentrale Rolle für die Spielsteuerung hat der Controller (Klasse `SnakeGameController`). Der Controller kann

- Nutzerinteraktionen (insbesondere Betätigen von Cursortasten) sowie
- Zeitsteuerung (insbesondere einen Trigger zur Bewegung der Schlange und einen Trigger zur Bewegung der Mäuse)

erkennen und in entsprechende Modelinteraktionen umsetzen. Der Controller wird detailliert im Abschnitt 3.3 erläutert.

Der `SnakeView` kapselt den DOM-Tree und bietet entsprechende Manipulationsmethoden für den Controller an, um sich verändernde Spielzustände im Browser zur Anzeige zu bringen. Der View wird im Abschnitt 3.2 erläutert.

Konzeptionell wird SnakeGame in einem Model abgebildet. Das Model ist komplexer und gliedert sich in mehrere logische Entities, die sich aus dem Spielkonzept des Abschnitts 2.2 ableiten und im Abschnitt 3.1 erläutert werden.

Zur Speicherung von Spielzuständen (bei SnakeGame nur Speicherung von Highscores) kann der Controller auf eine sogenannte Gamekey API zurürckgreifen. Die Gamekey API kapselt in Form der Klasse `GameKey` eine REST basierte Schnittstelle eines externen Gamekey Service. Mittels des Gamekey Service ist es möglich, Spielzustände eines Spiels in Form von JSON zu speichern und abzufragen. Der Gamekey Service und seine Anbindung wird im Abschnitt 3.4 erläutert.

Hinweis: Der Gamekey Service kann von beliebigen Spielen genutzt werden, um Spielzustände in einem JSON Format zu speichern. D.h. auch Sie sollen Ihr Spiel an den Gamekey Service anbinden.

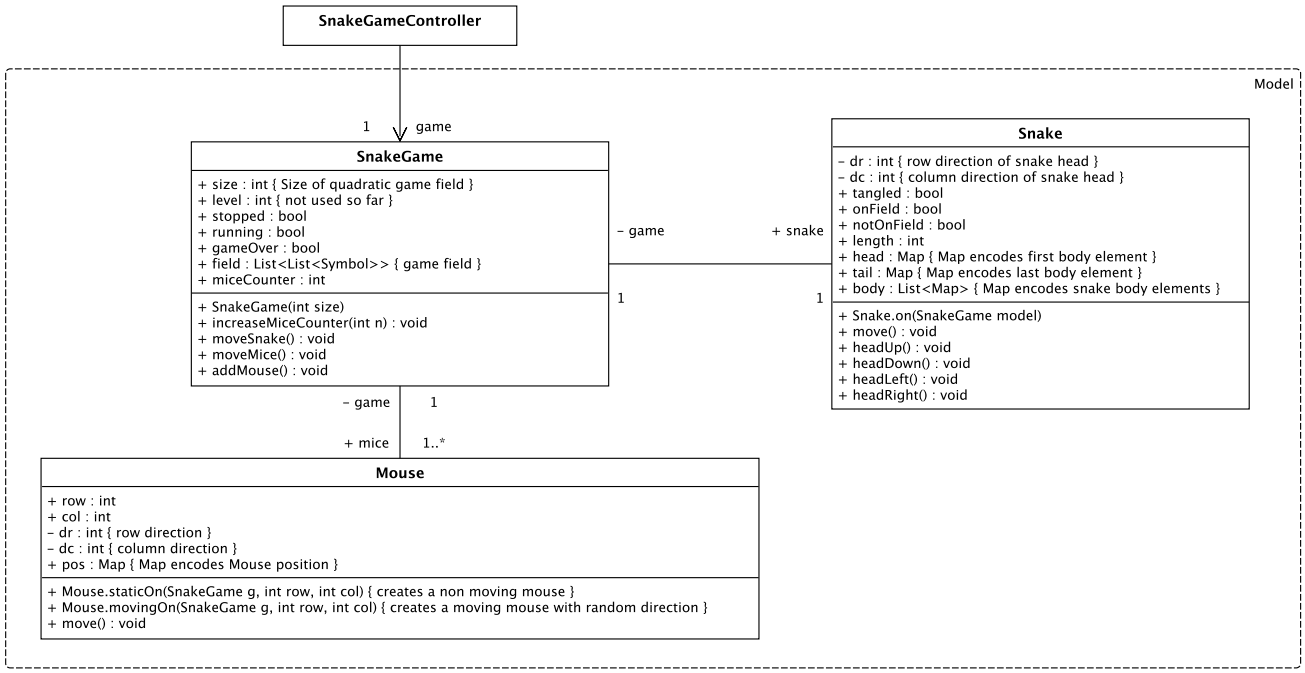


Abbildung 3: Klassendiagramm (Model und Controller)

3.1 Model

Aus dem Spielkonzept des Abschnitts 2.2 wurden eine Schlange (`Snake`) und eine Maus Entity (`Mouse`) abgeleitet. Ein Spiel (`SnakeGame`) besteht dabei aus einer Schlange (`Snake`, vgl. Abschnitt 3.1.2) und mindestens einer Maus (`Mouse`, vgl. Abschnitt 3.1.3). Das Klassendiagramm des Models ist in Abb. 3 gezeigt.

Hinweis: Aufgrund der geringen Komplexität des SnakeGame (deswegen wurde es als Anschauungsgegenstand gewählt) ist das Model sehr übersichtlich. Es ist unwahrscheinlich, dass ihr Model ähnlich 'klein' ausfallen wird (es sei denn sie wählen ein Spiel geringer Komplexität und beabsichtigen nicht das **Notenspektrum** nach oben hin auszunutzen). Üblicherweise hat das Model den größten Codeumfang, was logisch ist, denn hier wird ja die Spiellogik realisiert.

3.1.1 `SnakeGame`

Der Controller interagiert dabei nur mit dem `SnakeGame` und nicht mit dahinter liegenden Entities. Auf diese Weise ist es möglich, das Spiel um weitere Entities oder Entityvarianten zu erweitern, ohne dass der Controller geändert werden müsste. Das `SnakeGame` kann dabei über folgende Attribute dem Controller Aufschluss über den aktuellen Spielzustand geben:

- `size` liefert die quadratische Spielfeldgröße in Feldern.
- `level` bezeichnet den aktuellen Level, in dem sich das Spiel zum Zeitpunkt der Abgabe befindet (dies wird bislang im SnakeGame nicht genutzt).

- Mittels `stopped` und `running` kann in Erfahrung gebracht werden, ob das Spiel gerade läuft oder gestoppt ist (mittels `stopped` wäre so z.B. eine Pause Funktionalität möglich, die bislang ebenfalls nicht genutzt wird).
- `gameOver` gibt darüber Aufschluss, ob eine der definierten Game Over Bedingungen (verknotete Schlange, Schlange nicht mehr auf Spielfeld) eingetreten ist.
- Der `miceCounter` gibt an, wieviele Mäuse die Schlange bereits gefressen hat.

Ein `SnakeGame` Objekt

- kann mittels des Konstruktors erzeugt werden. Dabei wird die Spielfeldgröße (Default 30 x 30) übergeben.
- Der Controller kann dem Model mittels `moveSnake()` und `moveMice()` mitteilen, dass die Schlange bzw. alle Mäuse sich um einen Schritt bewegen sollen.
- Mittels `addMouse()` kann eine zusätzliche Maus dem Spiel hinzugefügt werden. Normalerweise sollte dies innerhalb des Models geschehen. Denkbar sind jedoch auch für Spielerweiterungen und Varianten besondere Ereignisse, die vom Controller erkannt werden (z.B. ein "Es regnet Mäuse Event"), die in `addMouse()` Aufrufe durch den Controller umgesetzt werden könnten. Dies wird jedoch im aktuellen Implementierungsstand des SnakeGame nicht genutzt.
- Mittels der Methode `increaseMiceCounter()` kann mitgezählt werden, wieviel Mäuse die Schlange bereits gefressen hat. Sie wird von der `move()` Methode der `Snake` Klasse aufgerufen, wenn erkannt wurde, dass eine Maus gefressen wurde.

3.1.2 `Snake` Entity

Der Zustand einer Schlange (`Snake`) besteht aus einer Liste von Body Elementen (`body` der Schlange). Der erste Eintrag dieser Liste bezeichnet den Kopf (`head`), der letzte Eintrag das Ende (`tail`) der Schlange. Jedes einzelne body Element wird dabei als Map ausgedrückt. Eine Schlange ist eine Liste solcher (aufeinander folgender) Elemente:

```
1 {
2   'row' : int, // Zeile
3   'col' : int // Spalte
4 }
```

Listing 1: Codierung eines Schlangenkörperelements

Die Bewegungsrichtung des Kopfs der Schlange wird mittels der Attribute `dr` (Bewegung entlang der Zeile, row direction $dr \in \{-1, 0, 1\}$) und `dc` (Bewegung entlang der Spalte, column direction $dc \in \{-1, 0, 1\}$) ausgedrückt. Das berechnete Attribute `tangled` gibt an, ob die Schlange verknotet (Game Over Bedingung) ist. Die berechneten Attribute `(not)OnField` geben an, ob die Schlange sich noch auf dem Spielfeld befindet (weitere Game Over Bedingung).

Mittels der Methoden `headUp()`, `headDown()`, `headLeft()` und `headRight()` kann der Schlange die Bewegungsrichtung (nach oben, nach unten, nach links, nach rechts) vorgegeben werden, in die sie sich im nächsten Schritt (und allen folgenden) zu bewegen hat.

Mittels der `move()` Methode wird die Schlange veranlasst ihren nächsten Schritt zu machen. Innerhalb der `move()` Methode wird geprüft, ob sie dabei eine Maus erwischt hat und diese fressen kann.

3.1.3 `Mouse` Entity

Der Zustand einer Maus (`Mouse`) besteht aus

- der aktuellen Zeile (`row`),
- der aktuellen Spalte (`col`) auf dem sich die Maus befindet,
- Bewegung entlang der Zeile, row direction `dr` $\in \{-1, 0, 1\}$) und
- Bewegung entlang der Spalte, column direction `dc` $\in \{-1, 0, 1\}$).

Das berechnete Attribut `pos` ergibt sich aus `row` und `col`, die als Map in demselben Format zurückgegeben werden, wie auch Schlangenkörperelemente (vgl. Listing 1) codiert werden.

Ein Mausobjekt kann mittels des Konstruktors `Mouse.staticOn()` als sich nicht bewegende Maus (`dr`=**0** und `dc`=**0**) und mittels des Konstruktors `Mouse.movingOn()` als sich zufällig in eine Richtung bewegende Maus erzeugt werden.

Mittels der Methode `move()` kann einer Maus mitgeteilt werden, dass sie den nächsten Schritt machen soll. Die Methode sorgt ferner dafür, dass die Maus auf dem Spielfeld bleibt, indem sie bei Berührung des Spielfeldrands den entsprechend **dr** oder **dc** in die andere Richtung dreht.

3.2 View

Der View dient der Darstellung des Spiels für den Spieler. Er besteht im Kern aus einem HTML-Dokument (siehe Abschnitt 3.2.1) und einer clientseitigen Logik, die den DOM-Tree des HTML-Dokuments manipuliert (siehe Abschnitt 3.2.2).

3.2.1 HTML-Dokument

Der View wird im Browser initial durch folgendes HTML-Dokument erzeugt. Im Verlaufe des Spiels wird der DOM-Tree dieses HTML-Dokuments durch die Klasse `SnakeView` manipuliert (siehe Abbildung 5), um den Spielzustand darzustellen und Nutzerinteraktionen zu ermöglichen. Die Klasse `SnakeView` wird dabei durch das Script `snakeclient.dart` als clientseitige Logik geladen.

```
1  <html>
2    <head>
3      [...]
4      <title>Snake Dart</title>
5      <link rel="stylesheet" type="text/css" href="style.css">
6    </head>
7
8    <body>
9      <div id="warningoverlay"></div>
10     <div id="overlay"></div>
11
12     <div class="container">
13       <h1 id="title">Snake Dart</h1>
14       <div id='welcome'>
15         <div id='highscore'></div>
16         <span id="start">Start</span>
17       </div>
18       <div id='message'>
19         <div id='gameover'></div>
```

```
20          <div id='reasons'></div>
21        </div>
22
23        <table id='snakegame'></table>
24
25        <div id='controls'>
26          <span id='points'></span>
27        </div>
28
29      [...]
30
31    </div>
32
33    <script type='application/dart' src='snakeclient.dart'></script>
34    <script src='packages/browser/dart.js'></script>
35  </body>
36 </html>
```

Listing 2: HTML Basisdokument des Spiels

Dieses HTML-Dokument wird genutzt, um darin das Spiel einzublenden. Abbildung 4 zeigt dabei zwei typische Screenshots.

3.2.2 `SnakeView` als Schnittstelle zum HTML-Dokument

Folgende Elemente haben dabei eine besondere Bedeutung und können dabei über entsprechende Attribute der Klasse `SnakeView` (siehe Abb. 5) angesprochen werden.

- Das Element mit dem Identifier `#warningoverlay` wird genutzt, um generelle Warnungen einzublenden (bspw. dass keine Verbindung zum Gamekey Service aufgebaut werden konnte).
- Das Element mit dem Identifier `#overlay` bezeichnet eine Overlay Ebene. Diese wird mittels CSS absolut positioniert und dient insbesondere dazu, dass Highscore Speicherformular des Games einzublenden (vgl. Abb. 4, linker Screenshot).
- Das Element mit dem Identifier `#highscore` dient dazu den Highscore einzublenden.
- Das Table-Element mit dem Identifier `#snakegame` dient dazu das Spielfeld einzublenden. Es wird mittels CSS so gestaltet, dass die Schlange als Hintergrund des Spielfelds dient.

Alle CSS-Gestaltungen werden in der `style.css` vorgenommen. Die Applikationslogik wird über das `snakeclient.dart` Script geladen. Für nicht Dart-fähige Browser wird ergänzend gem. den Dart-Konventionen die `dart.js` geladen, um Dart Logik in Javascript Engines zur Ausführung bringen zu können.

Objekte der Klasse `SnakeView` agieren nie eigenständig, sondern werden grundsätzlich von Objekten der Klasse `SnakeGameController` genutzt (vgl. Abb. 5), um View Aktualisierungen vorzunehmen (vgl. auch Sequenzdiagramme in den Abschnitten 3.3.1, 3.3.2 und 3.3.3). Der Controller kann sich hierzu folgender Methoden und Attribute bedienen, um den DOM-Tree nicht selber manipulieren zu müssen (Separation of Concerns):

- Die `update()` Methode aktualisiert den Spielzustand im Table-Element `#snakegame`. Sie wird durch den Controller im Verlaufe der `snakeTrigger` und `miceTrigger` Verarbeitung aufgerufen (vgl. auch Abb. 6).
- Die `generateField()` Methode erzeugt eine Tabelle zur Darstellung des quadratischen Spielfeldes. Dieses wird in den DOM-Tree in das TABLE-Element mit der Id `#snakegame` eingeblendet.

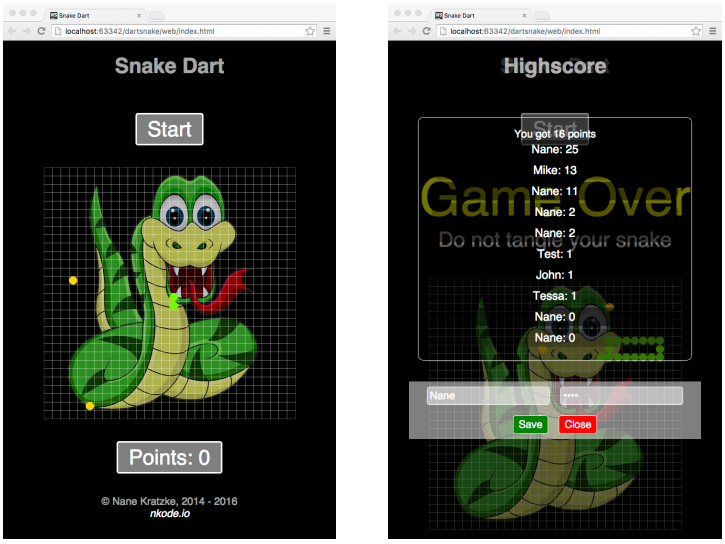


Abbildung 4: Screenshots des SnakeGames

- Highscore und Highscore Speicherung kann mittels der Methode `showHighscore()` als Formular in das DIV-Element mit der Id `#overlay` des DOM-Tree eingeblendet werden.
- Mittels der `warn()` Methode können innerhalb des Highscore Formulars Warnungstexte eingeblendet werden (bspw. falsches Password eingegeben, Spieler nicht vorhanden, etc.)
- Die Methode `closeForm()` blendet das Highscore Speicherformular wieder aus dem DOM-Tree aus (d.h. das Formular wird geschlossen).
- Mittels der Attribute `user` und `password` kann ein `SnakeGameController` im Verlaufe der Highscore Verarbeitung auf die Input-Eingaben des Highscore Formulars (siehe Abb. 4, rechte Screenshot) zugreifen.

3.3 Controller

Der Controller ist für die Ablaufsteuerung des Spiels zuständig. Er verarbeitet dabei gem. Abb. 2 sowohl zeitgesteuerte Events, als auch Nutzerinteraktionen. Während des Spiels hat der Spieler die Möglichkeit die Schlange (`Snake`) mittels der Cursortasten zu bewegen. Diese Steuerung ist recht naheliegend und kann wird am einfachsten durch das Listing 3 erläutert.

```
1    window.onKeyDown.listen((KeyboardEvent ev) {
2      if (game.stopped) return;
3      switch (ev.keyCode) {
4        case KeyCode.LEFT: game.snake.headLeft(); break;
5        case KeyCode.RIGHT: game.snake.headRight(); break;
6        case KeyCode.UP:  game.snake.headUp(); break;
7        case KeyCode.DOWN: game.snake.headDown(); break;
8      }
9    });
```

Listing 3: Steuerung der Schlange

Dies dreht im wesentlichen nur den Kopf der Schlange in die Richtung, in die sie sich im nächsten Zug zu bewegen hat. Komplexer sind die zeitgesteuerten Abläufe, die die Dynamik des Spiels realisieren. Der grundsätzliche Prozess sich

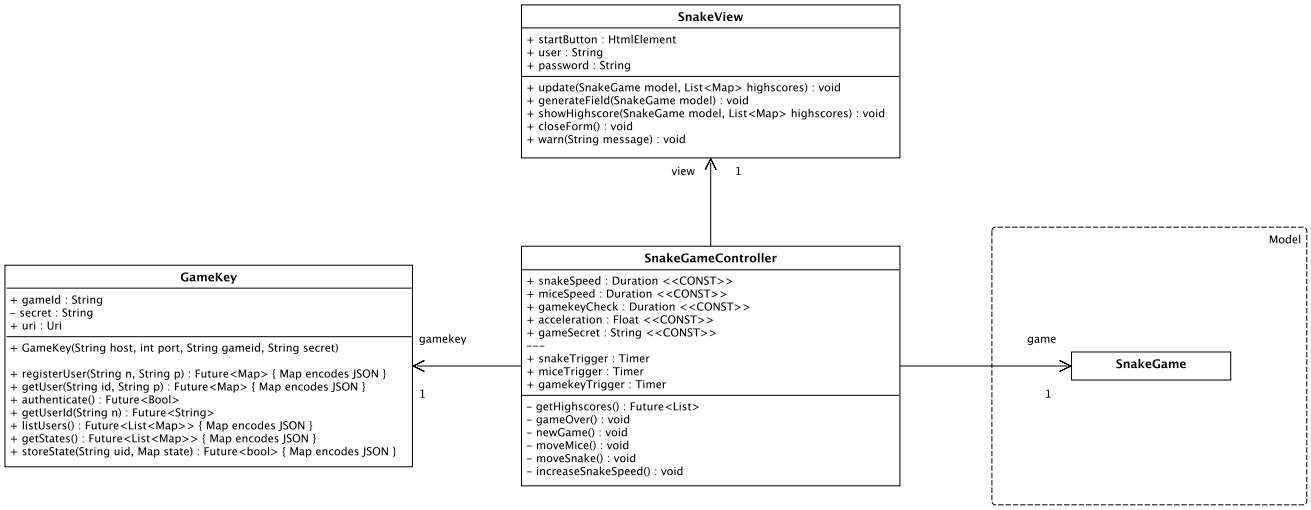


Abbildung 5: Klassendiagramm (View-Controller und Gamekey)

ändernder Spielzustände und deren Abbildung auf dem View, wird in Abschnitt 3.3.1 erläutert. Noch komplexer sind erstaunlicherweise die Abläufe im Falle des Game Overs. Zum einen müssen hier die zeitgesteuerten Events deaktiviert werden, zum anderen muss der Controller eng mit dem View und dem Gamekey Service interagieren, um ggf. erreichte Highscores zu speichern und noch nicht registrierte Nutzer gesondert im Gamekey Service zu registrieren. Diese GameOver Abläufe werden aufgrund ihrer Komplexität in den Abschnitten 3.3.2 und 3.3.3 gesondert erläutert.

3.3.1 Laufendes Spiel

Durch den `snakeTrigger` und den `miceTrigger` werden periodisch die Abläufe in Abbildung 6 angestoßen. Beide Abläufe sind identisch, daher wird nur der Kontrollfluss der Schlange erläutert. Der Kontrollfluss der Mäuse (im aktuellen Zustand wird nur eine sich nicht bewegende Maus genutzt) ist (bis auf eine fehlende Beschleunigung der Mausgeschwindigkeit) analog.

Obwohl nur eine sich nicht bewegende Maus genutzt wird, wäre es ein leichtes, das bestehende Spiel, so zu erweitern, dass die Schlange mehrere Mäuse jagen könnte. Das Spiel ist grundsätzlich dafür vorbereitet. Man müsste nur den Konstruktor des `SnakeGame` wie folgt erweitern.

Hinweis: Es ist häufig nicht sonderlich schwer, ein Spiel flexibel zu gestalten, wenn man sich Anfangs ein paar Gedanken macht und nicht nur versucht ein spezifisches Problem zu lösen.

Obwohl klassisch im SnakeGame nur eine sich nicht bewegende Maus genutzt wird, wäre es ein leichtes, das bestehende Spiel so zu erweitern, dass die Schlange mehrere Mäuse jagen könnte. Das Spiel ist grundsätzlich dafür vorbereitet. Man müsste nur den Konstruktor der Klasse `SnakeGame` wie folgt erweitern.

```
1 SnakeGame(this._size) {
2   start();
3   _snake = new Snake.on(this);
4   addMouse();
5   // addMouse(); // Um mit zwei Maeusen zu spielen
6   // addMouse(); // Um mit drei Maeusen zu spielen, ...
7   stop();
8 }
```

Listing 4: Konstruktor der Klasse SnakeGame

Die SnakeGame Implementierung sieht sogar sich bewegende Mäuse vor. Das ist der Grund warum es einen miceTrigger gibt, der bei sich nicht bewegenden Mäusen eigentlich nicht erforderlich wäre. Sie können SnakeGame mit sich bewegenden Mäusen spielen, wenn sie die addMouse() Methode der Klasse SnakeGame wie folgt abändern.

```
void addMouse() {
  if (stopped) return;
  Random r = new Random();
  final row = r.nextInt(_size);
  final col = r.nextInt(_size);
  //_mice.add(new Mouse.staticOn(this, row, col));
  _mice.add(new Mouse.movingOn(this, row, col));
}
```

Listing 5: Hinzufügen von Mäusen mittels der Methode addMouse() der Klasse SnakeGame

Auch wenn das SnakeGame in der vorliegenden Implementierung kein Level-System vorsieht, sollte ersichtlich sein, dass man auf Basis dieser beiden Variationsmöglichkeiten die Schwierigkeit des Spiels zusätzlich zur steigenden Geschwindigkeit der Schlange in unterschiedlichen Leveln variieren könnte.

Der snakeTrigger und miceTrigger feuern periodisch und der SnakeGameController ruft mit jedem Trigger Event die moveSnake() bzw. die moveMice() Methode auf, wie in Abb. 6 gezeigt. Als erstes wird geprüft, ob das Spiel einen Game Over Zustand erreicht hat. Sollte dies der Fall sein, wird gem. Abschnitt 3.3.2 oder 3.3.3 fortgefahren. Solange das Spiel noch läuft, ruft der Controller je nach Trigger die moveSnake() oder die moveMice() Methode des Model (SnakeGame) auf (die es wiederum an die move() Methoden der Klassen Snake bzw. Mouse delegieren, nicht in Abb. 6 dargestellt).

Auf diese Weise wird das Model in seinen nächsten Zustand überführt. Anschließend benachrichtigt der Controller den SnakeView über das Eintreten eines neuen Model Zustands und veranlasst so die Aktualisierung der Oberfläche mittels der update() Methode des Views.

3.3.2 Game Over bei registriertem Spieler

Im Falle das während der Bearbeitung des snakeTrigger oder des miceTrigger festgestellt wird, dass sich der GameOver Zustand im Spiel eingestellt hat, erfolgen zwei Abläufe, die gestartet werden, je nachdem ob es sich beim Spieler um einen registrierten Spieler (siehe Abschnitt 3.3.2) oder nicht registrierten Spieler (siehe Abschnitt 3.3.3) handelt.

Im Falle eines registrierten Spieler ist der Kontrollfluss wie in Abbildung 7 gezeigt.

Im Verlaufe des zeitgesteuerten Modelltriggerings (moveSnake() bzw. moveMice()) wird irgendwann der Game Over Zustand des Models mittels der gameOver() Methode festgestellt. In diesem Fall wird die Game Over Behandlung durch den SnakeGameController vorgenommen.

Hierzu veranlasst der SnakeGameController das SnakeGame mittels der Methode stop() in den stopped Status zu wechseln. Anschließend wird der SnakeView für ein letztes update() benachrichtigt. Anschließend wendet sich der SnakeGameController an die GameKey API um alle gespeicherten Highscores mittels getStates() anzufordern. Die GameKey Klasse leitet diese Anfrage an den GameKey Service über die in Abb. 2 gezeigte REST Schnittstelle (vgl. auch Tabelle 2) weiter. Diese Highscores werden sortiert und mittels showHighscore() an den SnakeView geleitet, damit dieser ein Formular zur Highscore-Speicherung auflegt (siehe auch Abb. 4(rechts)). Im Highscore Formular sind Eingabefelder

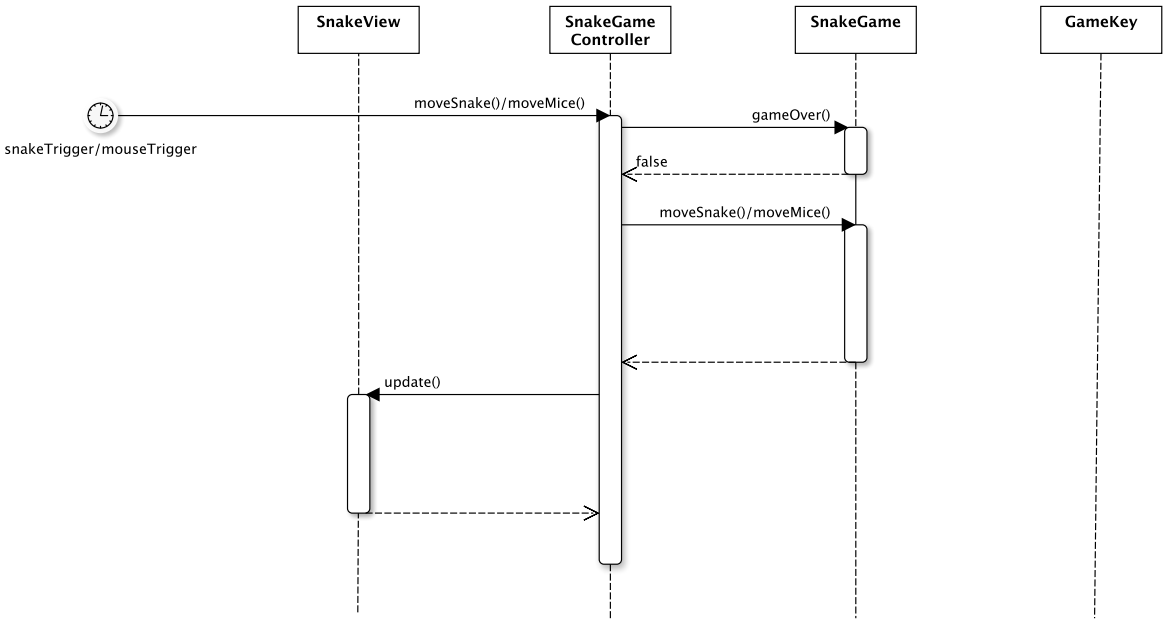


Abbildung 6: Sequenzdiagram (laufendes Spiel)

für User und Passwort Informationen sowie ein Save und Cancel Button vorhanden. Es werden entsprechende Listener registriert.

Die erste Phase der Verarbeitung ist damit beendet. Der View befindet sich jetzt in einem Zustand, dass ein Spieler seinen Namen und Passwort eingeben kann und seinen Highscore mittels Save speichern kann (bzw. das Speichern mittels Cancel abbrechen kann, wird hier nicht näher beleuchtet, weil trivial).

Die zweite Phase der Verarbeitung beginnt mit der Betätigung des **Save Buttons** durch den Spieler. In diesem Fall übernimmt wieder der Controller und fragt über den `SnakeView` Spieler (`user`) und Passwort (`pwd`) Daten aus den Eingabefeldern ab. Mittels dieser Daten wendet er sich mittels `getUserId()` an die `GameKey` API, um die Id des Spielers in Erfahrung zu bringen. Sollte der Spieler im GameKey Service existieren und das korrekte Passwort angegeben haben, so wird eine User Id zurück gegeben, die benötigt wird, um einen Status mittels `storeState()` im GameKey Service für einen Spieler zu speichern. Im erfolgreichen Fall des Speicherns wendet sich der `SnakeGameController` abschließend mittels `closeForm()` an den `SnakeView` um das Formular zu schließen und anschließend ein neues Spiel mittels `newGame()` zu starten.

3.3.3 Game Over bei nicht registriertem Spieler

Im Falle eines nicht registrierten Spieler ist der Kontrollfluss wie in Abbildung 8 gezeigt. Der Ablauf ist grundsätzlich wie in Abschnitt 7 gezeigt, jedoch muss die zweite Phase durch eine dritte Phase der Verarbeitung (Spieler Erzeugung) ergänzt werden.

Im Falle das ein Spieler das Spiel zum ersten Mal spielt (und vorher noch keine anderen an den GameKey Service angebunden Spiele gespielt hat) kann `getUserId()` keine User Id zurückgeben, denn der Spieler existiert ja noch nicht. In diesem Fall blendet der `SnakeGameController` mittels der `warn()` Methode im **SnakeView** eine entsprechende Warnung ein, dass der Spieler noch nicht existiert. Zusätzlich werden zwei Buttons (Create User, Cancel) eingeblendet, die bei entsprechenden Listenern registriert werden. Die zweite Phase ist damit um das storeState() gekürzt.

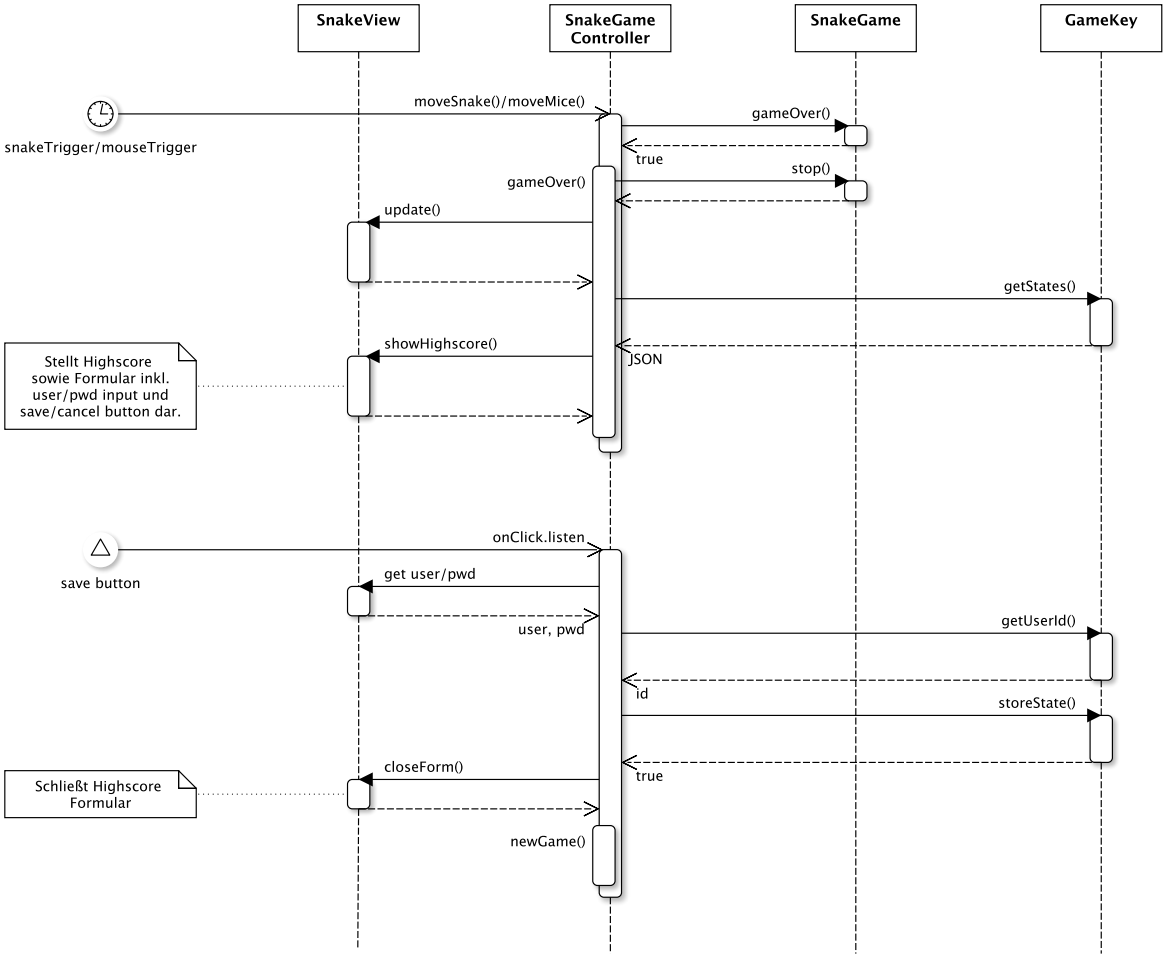


Abbildung 7: Sequenzdiagram (Game Over bei registriertem Spieler)

Die dritte Phase der Verarbeitung kann über den **Create Button** durch den Nutzer angestoßen werden. In diesem Fall übernimmt wieder der Controller und registriert den Spieler mittels `registerUser()` über die `GameKey` API beim GameKey Service. Im erfolgreichen Fall wird dem User eine Id zugewiesen, die dann dazu genutzt werden kann, den Highscore mittels `storeState()` wie im Abschnitt 3.3.2 erläutert zu speichern. Anschließend wird das Formular mittels `closeForm()` geschlossen und ein neues Spiel gestartet (`newGame()`).

3.4 REST-basierte Storage Lösung

Hinweis: Die Referenzimplementierung des Gamekey Service in Ruby, dient nur dazu, dass Spiel und Storage parallel durch Ihr Team entwickelt werden kann. Denken sie über eine entsprechende Aufgabenverteilung nach. Eine wesentliche Aufgabe ist es, dass sie die Storage Referenzlösung als REST-basierten Dart Service nachimplementieren, um nachzuweisen, dass Sie in der Lage sind Dart auch serverseitig (Storage) und nicht nur clientseitig (Spiel) einzusetzen. Gleichzeitig weisen Sie so nach, dass sie sich hinreichend in REST-Prinzipien und das HTTP Protokoll eingearbeitet haben.

Ihr Spiel muss sowohl mit der Referenzimplementierung in Ruby als auch mit ihrer Eigenentwicklung in Dart funktionieren.

3.4.1 Referenzimplementierung (GameKey Service, Ruby)

Die Referenz-Storage Lösung ist bewusst in Ruby und nicht in Dart implementiert und findet sich hier: `https://bitbucket.org/nanekratzke/gamekey`. Sie finden unter diesem Link auch eine kurze Online Dokumentation, wie der GameKey command line client funktioniert.

Sie können sich die Ruby Implementierung als Git Repository clonen:

```
git clone https://nanekratzke@bitbucket.org/nanekratzke/gamekey.git
```

Ihre Aufgabe ist es (neben der Entwicklung des eigentlichen Spiels), diese Referenzimplementierung mit identischer Verhaltensweise in Dart nachzuimplementieren.

Der GameKey Service kann auf Systemen auf denen Ruby (Version 2.2 oder höher) installiert ist, mittels

```
gem install gamekey
```

installiert werden. Anschließend kann der Service kann mittels

```
gamekey start
```

gestartet werden. Mittels

```
gamekey help
gamekey help <command>
```

erhalten sie weitere Nutzungshinweise für die einzelnen command line Befehle.

Der Gamekey-Service kann theoretisch als In-Memory Key-Value Lösung betrieben werden. Alle Schreiboperationen werden jedoch in der Referenzimplementierung in eine `gamekey.json` Datei serialisiert (das senkt die Performance, man erhält dafür aber Persistenz und Debug-Möglichkeiten). Diese Datei (letztlich nur eine 1:1 JSON Serialisierung der internen Datenstruktur zum Zeitpunkt der letzten Schreiboperation) können Sie für Test und Debugging-Zwecke heranziehen.

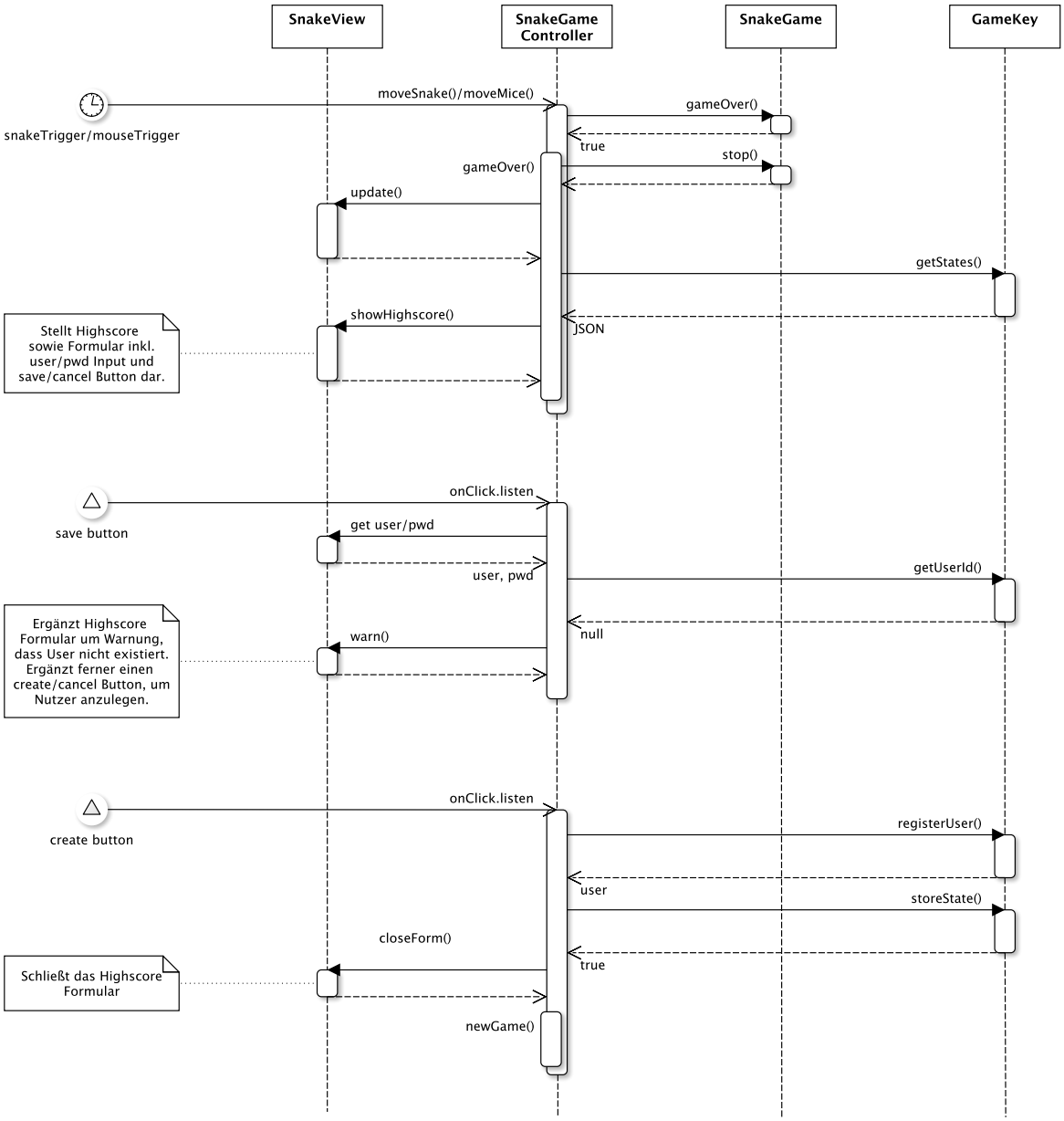


Abbildung 8: Sequenzdiagram (Game Over bei nicht registriertem Spieler)

Details der REST-basierten Schnittstellen finden sich in der `lib/gamekey.rb` in der Methode `Gamekey::api()`. Hier werden alle URLs und zulässigen HTTP Methoden sowie Rückgabecodes erklärt. Nachfolgende Tabelle 2 fast die REST-basierte Schnittstelle für einen ersten Überblick zusammen.

Ressource	METHOD	Erklärung
`/user`	`POST`	Erstellt eine neue User Resource (Spieler eines Games).
`/users`	`GET`	Liefert alle User als JSON Liste.
`/user/:id`	`GET`	Liest User Daten zu einem User mit der ID :id aus. User Daten werden als JSON zurückgegeben. Der User muss sich mit seinem Passwort authentifizieren.
`/user/:id`	`PUT`	Aktualisiert User Daten für einen User mit der ID :id. Aktualisierte User Daten werden als JSON zurückgegeben. Der User muss sich mit seinem Passwort authentifizieren.
`/user/:id`	`DELETE`	Löscht User Ressource für einen User mit der ID :id. Der User muss sich mit seinem Passwort authentifizieren. Diese Operation löscht auch alle GameStates des Users!
`/game`	`POST`	Erstellt eine neue Game Resource (Games).
`/games`	`GET`	Liefert alle Games als JSON Liste.
`/game/:id`	`GET`	Liest Game Daten zu einem Game mit der ID :id aus. Game Daten werden als JSON zurückgegeben. Das Game muss sich mit seinem Secret authentifizieren.
`/game/:id`	`PUT`	Aktualisiert Game Daten für ein Game mit der ID :id. Aktualisirte Game Daten werden als JSON zurückgegeben. Das Game muss sich mit seinem Secret authentifizieren.
`/game/:id`	`DELETE`	Löscht Game Ressource für ein Game mit der ID :id. Das Game muss sich mit seinem Secret authentifizieren. Diese Operation löscht auch alle GameStates des Games!
`/gamestate/:gid/:uid`	`GET`	Liefert alle GameState Ressourcen eines Games mit der ID :gid für den User mit der ID :uid. Das Game muss sich mit seinem Secret authentifizieren. Alle GameStates werden als JSON Liste zurückgegeben.
`/gamestate/:gid`	`GET`	Liefert alle GameState Ressourcen eines Games mit der ID :gid für alle. Das Game muss sich mit seinem Secret authentifizieren. Alle GameStates werden als JSON Liste zurückgegeben.
`/gamestate/:gid/:uid`	`POST`	Speichert einen GameState für ein Spiel mit der ID :gid und einem User mit der ID :uid. Das Game muss sich mit seinem Secret authentifizieren. Der User muss sich mit seinem Passwort authentifizieren.

Tabelle 2: REST Schnittstelle des Gamekey Service

Der Gamekey Service beinhaltet ferner einen Satz an automatisierten Referenztests, mit der Sie prüfen können, ob ihre Dart Implementierung des Gamekey Service korrekt ist und sich verhält wie die Referenzlösung.

```
gamekey test --host http://my.gamekey.implementation:8080
```

Die Tests die ihre Dartimplementierung bestehen muss, finden sich in der Datei `lib/reference.rb` der Referenzimplementierung. Nicht erfolgreiche Testfälle werden mit Identifier ausgegeben. Anhand des Identifiers können sie in der Referenzimplementierung herausfinden, welcher Testfall nicht erfolgreich war. Auch die Referenzimplementierung ist gegen diese Testfälle geprüft worden.

Ihre Referenzimplementierung muss ferner mit dem GameKey command line client wie die Referenzimplementierung funktionieren.

Hinweis: Ihnen wird ein funktionsfähiger GameKey Service in der Ruby Referenzimplementierung für die Laufzeit des Projekts zur Verfügung gestellt. Den Endpoint finden sie in Moodle. Ihr Game muss aber ebenfalls mit ihrer Dart Implementierung des Gamekey Service funktionieren.

3.4.2 Client-seitige Anbindung an den GameKey Service (Dart Schnittstelle)

Hinweis: Die Gamekey Dart Schnittstelle unterstützt nicht die volle Funktionalität der REST Schnittstelle. Bspw. ist es nicht möglich Spielernamen zu ändern, Spieler zu löschen, oder dass Passwort eines Spielers zu ändern. Es wird daher absehbar nicht für ihr Spiel ausreichen, einfach nur die `gamekey.dart` Datei für ihr Spiel zu übernehmen.

Hinweis: Vergessen Sie bitte nicht! Sie müssen sowohl die **clientseitige Dart API** als auch die **serverseitige Storage Lösung** in Dart entwickeln. Ihre clientseitige Dart API muss auch mit der Rubyimplementierung funktionieren. Sie können die Referenztests der Ruby Implementierung problemlos gegen ihre Dart-Implementierung laufen lassen, um zu prüfen, ob sich ihr Dart Service genauso verhält, wie der Ruby Referenz-Service. D.h. sie müssen keine eigenen Testfälle für die Storage Lösung entwickeln (sie dürfen aber natürlich).

Der GameKey Service wird in SnakeDart mittels eines Objekts der Klasse `GameKey` (vgl. Abb. 5) angesprochen. Die Klasse `GameKey` dient dem Controller dazu mit dem GameKey Service interagieren zu können, ohne die Details der REST Schnittstelle kennen zu müssen.

Ein `GameKey` Objektzustand umfasst letztlich nicht mehr

- als die `gameid` des Spiels (wird bei Registerieren des Spiels vergeben, siehe Online Dokumentation des Gamekey Service)
- das `secret` des Spiels (wird mit der gameid zur Authentifizierung benötigt)
- sowie der `uri` des GameKey Service Endpoints.

Ein `GameKey` Objekt kann mittels eines Konstruktors erzeugt werden. Anschließend ist dieses Objekt in der Lage mit dem GameKey Service über eine REST-Schnittstelle (vgl. Tabelle 2) zu kommunizieren. Es kapselt damit im wesentlichen die GameKey REST API für den `SnakeGameController` in Form folgender Methoden:

- `registerUser()` dient dazu, einen neuen Nutzer im GameKey Service zu registrieren (hierzu muss Nutzername und Passwort bekannt sein).
- `getUser()` dient dazu, detaillierte Informationen über einen Nutzer abzurufen.
- `authenticate()` dient dazu, einen Nutzer anhand seines Namens und seines Passworts zu authentifizieren.
- `getUserId()` dient dazu, auf Basis eines Nutzernamens die zugehörige Id des Nutzers zu bestimmen (da ein Nutzer diese kryptische Id normalerweise niemals bewusst eingeben würde)
- `listUsers()` liefert eine JSON List mit öffentlichen Informationen über alle im GameKey registrierten Nutzer.
- `getStates()` liefert eine JSON List mit allen für ein Game gespeicherten Spielzustände.
- `storeState()` speichert für einen Nutzer einen JSON State (der mittels `getStates()` nachträglich wieder abgerufen werden kann).

Alle genannten Methoden arbeiten asynchron (da sie auf Netzwerkzugriffen beruhen) und lieferen daher `Futures` zurück. Die oben genannten Methoden decken nur den Teil der REST API ab, die für das SnakeGame erforderlich sind. Die REST API wird nicht komplett durch diese Dart Schnittstelle genutzt.

4 Level- und Parametrisierungskonzept

Hinweis: SnakeDart sieht kein Levelkonzept (bis auf eine triviale Beschleunigung der Schlange) und nur ein rudimentäres Parameterisierungskonzept vor. Sie sollten für ihr Spiel jedoch beides in deutlich umfangreicherer Form vorsehen. Ihnen dient dieses Kapitel daher primär als Platzhalter beides zu beschreiben.

4.1 Levelkonzept

Da SnakeGame nur als Anschauungsgegenstand dient, wurde auf die Implementierung eines Levelkonzepts verzichtet.

Hinweis: Sie sollten an dieser Stelle beschreiben, wie ihre Level definiert werden und nach welchen Kriterien sie steigende Schwierigkeitsgrade definieren. Gehen Sie dabei insbesondere auf die Art und Weise ein, mittels welcher Datenformate sie unterschiedliche Level konfigurieren. Vergessen Sie kein einprägsames Beispiel, um ihre Leveldefinition exemplarisch zu erläutern.

Hinweis: Ihre Angaben müssen so detailliert sein, dass jemand, der nicht aus ihrem Team stammt, in der Lage ist, einen weiteren Level für das Spiel zu ergänzen und/oder bestehende Level abzuändern.

4.2 Parameterisierungskonzept

Da SnakeGame nur als Anschauungsgegenstand dient, wurde auf die Implementierung eines vollständigen Parametrisierungskonzepts verzichtet. Dennoch wurde bspw. die Anbindung an den Gamekey-Service mittels einer extermen Settings-Datei folgenden Formats umgesetzt.

```
{
  "gameid": "ba2ed888-8df1-400b-a510-86307d7bbd68",
  "host":  "52.28.67.138",
  "port":  8080
}
```

Listing 6: Spielparameter des `SnakeGames` (`gamekey.json`)

- Die `gameid` bezeichnet dabei die vom GamekeyService während der Registrierung zugewiesen Identifier
- `host` ist ein valider DNS-Name oder IP Adresse (String codiert), der angibt auf welchem Host der Gamekey Service läuft (da der Gamekey Service CORS enabled ist, muss dies nicht derselbe Host sein, von dem das Spiel geladen wurde)
- `port` ist ein Integer Wert, der den Port angibt auf dem der Gamekey Service lauscht

Diese Datei wird durch den `SnakeGameController` zu Beginn des Ladens des Spiels auf folgende Weise geladen und zur Herstellung der Gamekey-Connection genutzt. `gamekeySettings` ist dabei eine String-Konstante die lediglich auf die `'gamekey.json'` Datei verweist.

```
    // Download gamekey settings. Display warning on problems.
    HttpRequest.getString(gamekeySettings).then((json) {
      final settings = JSON.decode(json);

```

```
5         // Create gamekey client using connection parameters
6         this.gamekey = new GameKey(
7             settings['host'],
8             settings['port'],
9             settings['gameid'],
10            gameSecret
11        );
12
13        // Check periodically if gamekey service is reachable. Display warning if not.
14        this.gamekeyTrigger = new Timer.periodic(gamekeyCheck, (_) async {
15          if (await this.gamekey.authenticate()) {
16            view.warningoverlay.innerHtml = "";
17          } else {
18            view.warningoverlay.innerHtml =
19              "Could not connect to gamekey service. "
20              "Highscore will not working properly.";
21          }
22        });
23      });
```

Listing 7: Laden der Spielparameter

Hinweis: Es sollte ihnen ein Leichtes sein, diesen Ansatz für weitere spielrelevante Parameter zu erweitern. Sie ermöglichen sich damit nebenbei einfachere Anpassungsmöglichkeiten während der finalen Feintuningphase ihres Spiels.

Hinweis: Beachten sie bitte, dass sie ein Parametrisierungskonzept ggf. für ihr Spiel und einmal für Ihre Dart-Implementierung des Gamekey Storage Service benötigen. Sie müssen also ggf. zwei Parameterisierungskonzepte dokumentieren, bzw. erläutern, warum ein Parametrisierungskonzept ggf. nicht erforderlich ist.

Ergänzend zu den Angaben in der `gamekey.json` Datei sind diverse Konstanten im `SnakeGameController` (`lib/src/control.dart`) definiert (die grundsätzlich auf vergleichbare Art und Weise in die `gamekey.json` ausgelagert werden könnten).

- `snakeSpeed` gibt die Zeitspanne in Millisekunden an, die zwischen zwei Schlangenbewegungen liegen soll (je größer, je langsamer ist die Schlange)
- `acceleration` gibt die Reduktion der snakeSpeed an, wenn die Schlange eine Maus gefressen hat. 0.01 bedeutet dass die Schlangenbewegung mit jeder gefressenen Maus 1% schneller wird. Je größer dieser Wert wird, desto spürbarer ist die Beschleunigung mit jeder gefressenen Maus (desto schwieriger wird das Spiel).
- `miceSpeed` gibt die Zeitspanne in Millisekunden an, die zwischen zwei Mausbewegungen liegen soll (je größer, je langsamer ist die Maus, desto einfacher sind Mäuse für die Schlange zu fressen).
- `gamekeyCheck` gibt das Intervall in Sekunden an, in dem geprüft werden soll, ob der Gamekey Service erreichbar ist.
- `gameSecret` gibt das Secret des Games an, welches bei der Registrierung des Games beim GamekeyService angegeben wurde/bzw. durch den Command Line Client automatisch generiert wurde. Dieses Secret ist aus Gründen der Schlüsselverteilung hart in das Spiel codiert. Bei Änderungen des Secrets muss auch das Spiel aktualisiert werden. Von der Auslagerung des gameSecret in die `gamekey.json` wird jedoch aus Sicherheitsgründen abgeraten.

5 Nachweis der Anforderungen

Nachfolgend wird erläutert wie die in Kapitel 2 eingeführten funktionalen Anforderungen, die Dokumentationsanforderungen und die technischen Randbedingungen erfüllt bzw. eingehalten werden. Dies kann – wie nachfolgend geschehen – argumentativ erfolgen, und muss nicht durch eine Testfall-getriebene Nachweisführung erfolgen. Abschließend wird angegeben, wer im Team welche Verantwortlichkeiten hatte.

Hinweis: Aufgrund der Kürze der Zeit, sollen sie in diesem Projekt nur argumentativ und über ihr Spiel nachweisen, dass sie die einzelnen Anforderungen abgedeckt haben.

Hinweis: Verweisen Sie – falls möglich und sinnvoll – bitte bei ihrer Argumentation auf konkrete Assets (Klassen, Scripte, Kapitel der Dokumentation, Konfigurationsdateien, etc.) die ihre These der Erfüllung von Anforderungen stützen. **Behaupten Sie keine Erfüllung, die sie selber nicht sehen. Sie sind hier zu wahrheitsgemäßen und nachvollziehbaren Angaben verpflichtet! Ihre Argumente gehen in die Notenfindung ein.**

5.1 Nachweis der funktionalen Anforderungen

Hinweis: Nachfolgend erfolgt der Nachweis der Einhaltung funktionaler Anforderungen. Grundsätzlich gilt: Nur teilweise oder nicht Erfüllung von Anforderungen bringt **Notenabzüge** mit sich. Pro Anforderung sind Erläuterungen anzugeben, warum eine Anforderung als erfüllt, teilweise erfüllt oder nicht erfüllt betrachtet wird. **Nicht erläuterte Angaben, werden wie nicht erfüllt gewertet.**

Id	Kurztitel	Erfüllt	Teilw. erfüllt	Nicht erfüllt	Erläuterung
AF-1	Einplayer Game	×			SnakeGame ist ein Einpersonen Spiel, wie aus dem in Abschnitt 2.2 dargestellten Spielkonzepts hervorgeht.
AF-2	2D Game	×			SnakeGame wird auf einem 2D-Raster gespielt, wie aus dem in Abschnitt 2.2 dargestellten Spielkonzepts hervorgeht.
AF-3	Levelkonzept			×	Da SnakeGame nur als Anschauungsgegenstand dient, wurde auf die Implementierung eines Levelkonzepts verzichtet.
AF-4	Parametrisierungs-konzept		×		Da SnakeGame nur als Anschauungsgegenstand dient, wurde auf die Implementierung eines ausgereiften Parametrisierungskonzept verzichtet. Die Gamekey Anbindung wurden jedoch mittels einer Parameterdatei realisiert (siehe Abschnitt 4.2). Weitere Werte zur Verhaltensbeeinflussung, wurden jedoch als Konstanten in der Klasse SnakeGameController realisiert, so dass diese an zentraler Stelle – jedoch nur mit Änderung des Quellcodes — geändert werden können.
AF-5	REST-basierter Storage	×			Das Spiel kann auf die JSON-basierte Referenz-Storagelösung zugreifen (vgl. Abschnitt 3.4.2).
AF-6	Desktop Browser	×			Das Spiel ist in Desktop Browsern spielbar. Es wurde in Chrome, Safari und Firefox erfolgreich zur Ausführung gebracht, gespielt und jeweils ein Highscore gespeichert.
AF-7	Mobile Browser			×	Da SnakeGame nur als Anschauungsgegenstand dient, wurde auf die SmartPhone Unterstützung von Mobilebrowsern verzichtet.

Tabelle 3: Nachweis der funktionalen Anforderungen

5.2 Nachweis der Dokumentationsanforderungen

Hinweis: Nachfolgend erfolgt der Nachweis der Einhaltung der Dokumentationsanforderungen. Grundsätzlich gilt: Nur teilweise oder nicht Erfüllung von Anforderungen bringt Punktabzüge mit sich. Pro Anforderung sind Erläuterungen anzugeben, warum eine Anforderung als erfüllt, teilweise erfüllt oder nicht erfüllt betrachtet wird. **Nicht erläuterte Angaben, werden wie nicht erfüllt gewertet.**

Id	Kurztitel	Erfüllt	Teilw. erfüllt	Nicht erfüllt	Erläuterung
D-1	Dokumentations-vorlage	x			Vorliegende Dokumentation dient als Vorlage für Spieldokumentationen.
D-2	Projektdokumentation	x			Vorliegende Dokumentation erläutert die übergeordneten Prinzipien und verweist an geeigneten Stellen auf die Quelltextdokumentation.
D-3	Quelltext-dokumentation	x			Es wurden alle Methoden und Datenfelder, Konstanten durch Inline-Kommentare erläutert.
D-4	Libraries	x			Alle genutzten Libraries werden in der `pubspec.yaml` der Implementierung aufgeführt. Da nur die zugelassenen Pakete genutzt wurden, sind darüber hinaus keine weiteren Erläuterungen, warum welche Pakete genutzt wurden, erforderlich.

Tabelle 4: Nachweis der Dokumentationsanforderungen

Hinweis: Achten sie bitte darauf, dass sie beim Nachweis der erlaubten/verbotenen Pakete ggf. zwei Projekte berücksichtigen. Einmal ihr Spiel und einmal die Referenzimplementierung des Gamekey Storage Service. Sie müssen den Nachweis also ggf. für beide Projekte erbringen.

5.3 Nachweis der Einhaltung technischer Randbedingungen

Hinweis: Nachfolgend erfolgt der Nachweis der Einhaltung der vorgegebenen technischen Randbedingungen. Grundsätzlich gilt: Nur teilweise oder nicht Erfüllung von Anforderungen bringt Punktabzüge mit sich. Pro Anforderung sind Erläuterungen anzugeben, warum eine Anforderung als erfüllt, teilweise erfüllt oder nicht erfüllt betrachtet wird. **Nicht erläuterte Angaben, werden wie nicht erfüllt gewertet.**

Hinweis: Achten sie bitte darauf, dass sie beim Nachweis der erlaubten/verbotenen Pakete ggf. zwei Projekte berücksichtigen. Einmal ihr Spiel und einmal die Referenzimplementierung des Storage Service. Sie müssen den Nachweis also ggf. für beide Projekte erbringen.

Hinweis: Achten sie bitte bei der Storage Lösung darauf, dass Sie objektive Nachweise erbringen müssen, dass ihre Lösung alle Referenztests bestanden hat und das ihr Spiel sowohl mit der Referenzlösung als auch ihrer Dart Implementierung des Gamekey Service funktionieren muss!

Id	Kurztitel	Erfüllt	Teilw. erfüllt	Nicht erfüllt	Erläuterung
TF-1	No Canvas	×			Die Darstellung des Spielfeldes sollte ausschließlich mittels DOM-Tree Techniken erfolgen. Die Nutzung von Canvas-basierten Darstellungstechniken ist **explizit** untersagt. Die Klasse **SnakeView** nutzt keinerlei Canvas basierten DOM-Elemente.
TF-2	Levelformat			×	Da SnakeGame nur als Anschauungsgegenstand dient, wurde auf die Implementierung eines Levelkonzepts verzichtet.
TF-3	Parameterformat		×		Da SnakeGame nur als Anschauungsgegenstand dient, wurde auf die Implementierung eines ausgereiften Parametrisierungskonzept verzichtet. Die Gamekey Anbindung wurden jedoch mittels einer Parameterdatei realisiert (siehe Abschnitt 4.2). Weitere Werte zur Verhaltensbeeinflussung, wurden jedoch als Konstanten in der Klasse SnakeGameController realisiert, so dass diese an zentraler Stelle – jedoch nur mit Änderung des Quellcodes — geändert werden können.
TF-4	HTML + CSS	×			Der View des Spiels beruht ausschließlich auf HTML und CSS (vgl. web/index.html).
TF-5	Gamelogic in Dart	×			Die Logik des Spiels ist mittels der Programmiersprache Dart realisiert worden.
TF-6	Storagelogic in Dart			×	Da SnakeGame nur als Anschauungsgegenstand dient, wurde auf die Dart-Implementierung des Storageservice verzichtet.
TF-7	Storage Referenz		×		SnakeGame nutzt nur die Referenzimplementierung des Storageservice. Da die Eigenrealisierung des Storageservice jedoch in Dart nicht vorgenommen wurde, kann dieser Teil der Anforderung nicht nachgewiesen werden.
TF-8	Storage Referenztest			×	Es wurde keine Dart Implementierung der Referenz Storage Services vorgenommen.
TF-9	Browser Support	×			Das Spiel muss im Browser Chromium/Dartium (native Dart Engine) funktionieren. Das Spiel muss ferner in allen anderen Browserrn (JavaScript Engines) ebenfalls in der JavaScript kompilierten Form funktionieren (geprüft wird ggf. mit Safari, Chrome und Firefox).
TF-10	MVC Architektur	×			Das Spiel folgt durch Ableitung mehrerer Modell-Klassen (vgl. lib/src/model.dart), einer View Klasse (vgl. lib/src/view.dart) und dem zentralen Controller (vgl. lib/src/controller.dart) einer MVC-Architektur. Der GameKey Service (vgl. lib/src/gamekey.dart) wird ebenfalls diesem Prinzip unterworfen und wird nur vom Controller getriggert, genauso wie das Modell und der View. Der View greift zudem auf das Model nur lesend und nicht manipulierend zu.
TF-11	Erlaubte Pakete	×			Es sind nur dart:* packages, sowie das Webframework start genutzt worden. Siehe `pubspec.yaml` der Implementierung.
TF-12	Verbotene Pakete	×			Es sind keine Pakete, außer den erlaubten genutzt worden. Siehe `pubspec.yaml` der Implementierung.
TF-13	No Sound	×			Das Spiel hat keine Soundeffekte.

Tabelle 5: Nachweis der technischen Randbedingungen

5.4 Verantwortlichkeiten im Projekt

Hinweis: Sie müssen dokumentieren, welche Person für welche Projekt Ergebnisse verantwortlich war. Am besten machen Sie dies, indem sie relevante Assets identifizieren. Mittels einer Matrix lässt sich dann einfach dokumentieren, wer von Ihnen welche Assets verantwortlich und unterstützend erstellt hat. Dies kann bspw. wie nachfolgend gezeigt geschehen. **Diese Angaben werden im Rahmen der individuellen Notenfindung herangezogen.**

Komponente	Detail	Asset	Nane Kratzke	Heinzel Mann	Heinzel Frau	Anmerkungen
Model	Snake Game	`lib/src/model.dart`	V			beinhaltelt Inline Dokumentation
	Schlange	`lib/src/model.dart`	V		U	beinhaltelt Inline Dokumentation
	Maus	`lib/src/model.dart`	V	U		beinhaltelt Inline Dokumentation
View	HTML-Dokument	`index.html`	V			
	Gestaltung	`style.css`	V		U	
		`img/*`	V		U	
	Viewlogik	`lib/src/view.dart`	V	U		beinhaltelt Inline Dokumentation
Controller	Eventhandling	`lib/src/controller.dart`	V		U	beinhaltelt Inline Dokumentation
	Parametrisierung	`gamekey.json`	V		U	
	Level	-				nicht umgesetzt
GameKey API	REST Anbindung	`lib/src/gamekey.dart`	V	U		beinhaltelt Inline Dokumentation
Documentation	SnakeGame Report	`doc/*.*`	V			Lyx/Latex Report
GameKey Service (Ruby)	Command Line Client	`bin/gamekey`	V		U	test command von Heinzel Frau
	Dockerization	`Dockerfile`	V		U	
	Gamekey Server	`lib/*.rb`	V			beinhaltelt Inline Dokumentation
	Referencetests	`lib/reference.rb`	V	U		beinhaltet inline Dokumentation
	Dokumentation	`README.md`	V			Online Dokumentation

V = verantwortlich (hauptdurchführend, kann nur einmal pro Zeile vergeben werden)
U = unterstützend (Übernahme von Teilaufgaben)

Tabelle 6: Projektverantwortlichkeiten

Hinweis: Bei der Dokumentation bietet es sich an, diese ggf. nochmals in Kapitel aufzuteilen, um sie besser Einzelpersonen zuordnen zu können.

Hinweis: Wenn Sie eine Aufgabenzuordnung wählen, die im wesentlichen besagt, dass alle alles gemacht haben, zeigt das eigentlich nur, dass sie recht planlos agiert haben. Achten Sie darauf, dass die Verantwortlichkeiten und Unterstützungen gleichmäßig und fair im Projekt verteilt werden. Grundsätzlich bekommen alle im Team dieselbe Note, wenn sich jedoch Verantwortlichkeiten bei einer Person häufen, ist klar, dass diese die Hauptlast im Projekt getragen hat. Wenn Personen nur unterstützend tätig waren, ist klar, dass diese durch das Team mit durchgezogen wurden. **Solche Fälle haben natürlich Auswirkungen auf die Note!** Im oben stehenden Fall ist recht deutlich, dass die Hauptlast wohl nicht Heinzel Mann und Heinzel Frau getragen haben.