Software Entwicklung 2

Sommersemester 2019

JumpOrDie

Autoren

Agil Homar,	ah210
Christian Heinz,	ch148
Salome Wecks,	sw193
Jonas Leitner	jl121

https://gitlab.mi.hdm-stuttgart.de/jl121/se2-projekt

Kurzbeschreibung

Bei unserem Projekt handelt es sich um ein klassisches Jump and Run Spiel, welches wir für Software-Entwicklung 2 des Studiengangs Medieninformatik an der Hochschule der Medien programmiert haben.
Der Benutzer hat die Möglichkeit durch die Tastatur seine Spielfigur springen zu lassen, oder sich zu ducken. Das ist auch nötig, da mit der Zeit zufällig generierte Hindernisse auf ihn zu kommen und er ausweichen muss, um nicht zu kollidieren. Die Hindernisse haben verschiedene Formen bzw. Größen und kommen entweder am Boden oder in der Luft vor. Man muss sich also rechtzeitig entscheiden wann man springen will oder sich ducken sollte. Um den Spielspaß zu erhöhen, kommen mit der Zeit die Hindernisse näher bei einander und natürlich werden sie kontinuierlich schneller. Ziel ist es, möglichst lange zu überleben und einen hohen Score zu erspielen! Um den Anreiz auf einen guten Score zu erhöhen, gibt es eine Scoreliste auf der man sich mit anderen Spielern vergleichen kann und um den ersten Platz kämpft.
Viel Spaß!

Startklasse

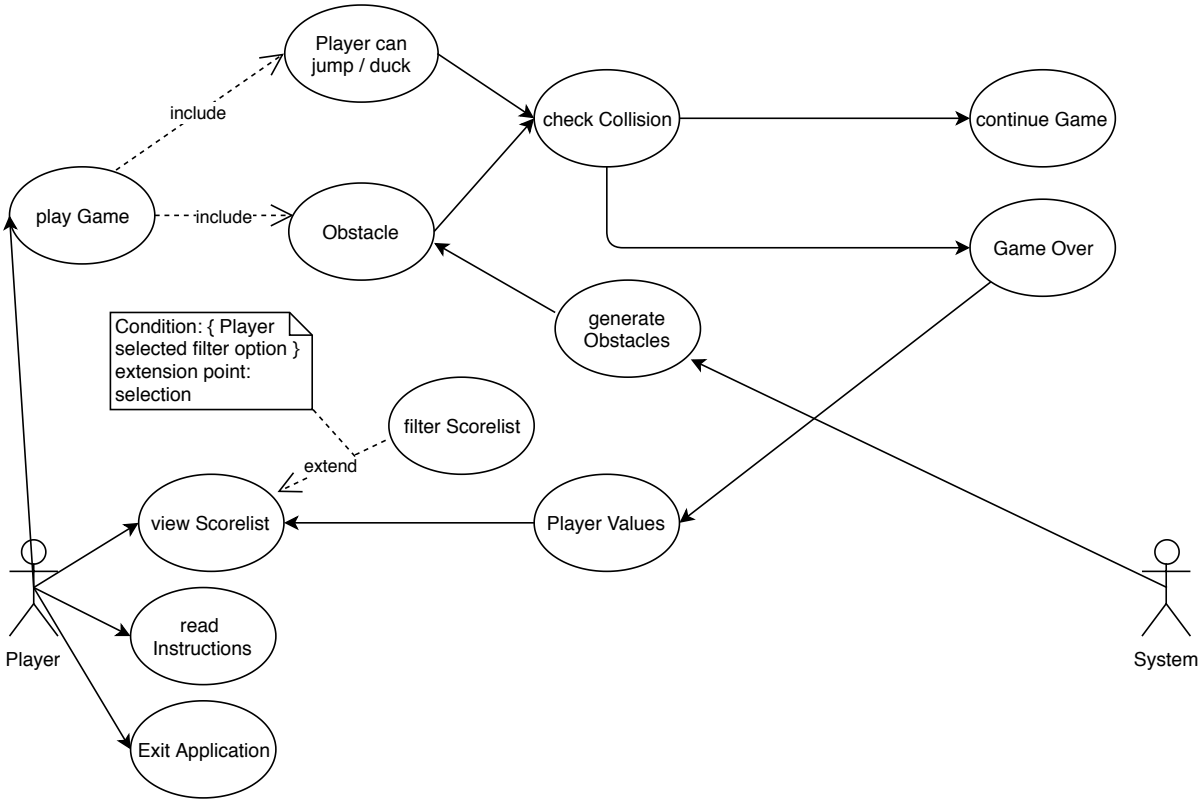
Die Main-Methode befindet sich in der Klasse: **App** (AgChSaJo.GUI.App)
(verwendet wurde der JDK 11)

Besonderheiten

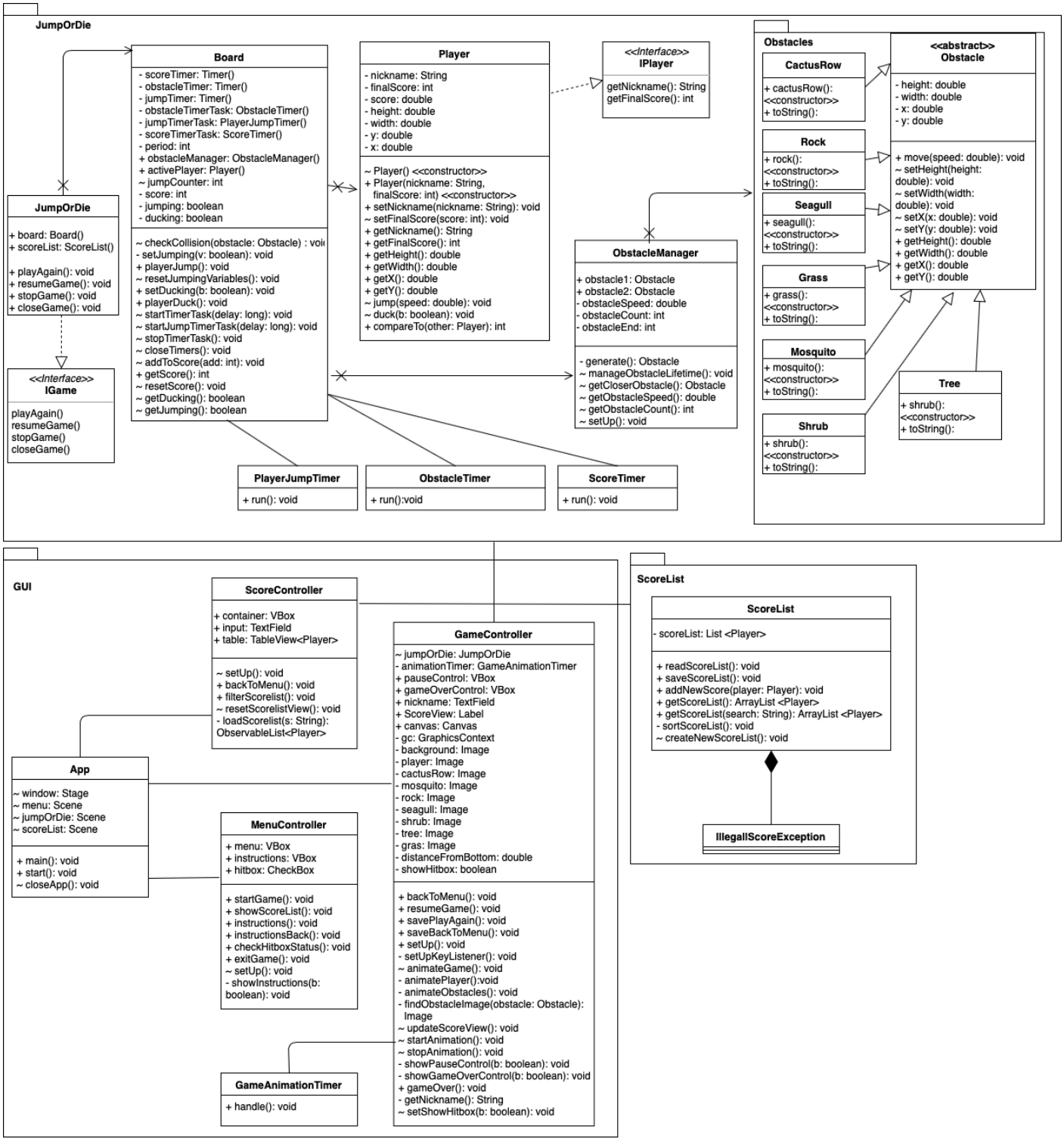
Instructions
Es könnte Sinn machen vor dem ersten Spiel kurz in den Menüpunkt Instructions zu schauen und sich mit der Steuerung vertraut zu machen.
Des Weiteren kann dort ausgewählt werden, ob man sich die exakten Hitboxen anzeigen lassen will oder nicht.

Gameplay
Es könnte sein, dass einem das Gameplay am Anfang etwas schwieriger vorkommt, als nach ein paar Hindernissen, da sich die Geschwindigkeit steigert und das Spiel flüssiger wirkt. Hier könnte man eventuell noch an den Parametern schrauben.
Feature: Durch ‘ESC‘ kann das Spiel pausiert werden.

UML Use Case Diagramm



UML Klassendiagramm



Stellungnahmen

Architektur

Package-Struktur

Wir haben eine klare Aufteilung unserer Klassen in drei Hauptordner. Zum einen **GUI**, dort sind alle Klassen gespeichert, die mit dem Aufbau und der Interaktion des Graphischen User Interfaces zu tun haben. Der nächste Ordner **JumpOrDie**, beinhaltet die komplette Spiellogik wie der Name schon verrät. Wobei wir noch einen zusätzlichen Unterordner **Obstacles** angelegt haben. Dieser enthält die verschiedenen Hindernisse, die es im Spiel gibt. Der dritte Hauptordner heißt **ScoreList** und enthält eine Klasse, die unsere Scoreliste verwaltet und die dazugehörige *IllegalScoreException*, die bei einem Speicherversuch fallen kann.

Interfaces

Wir haben zwei eigene Interfaces erstellt. Zum einen **IGame,** welches vorgibt dass man ein Spiel starten, pausieren und wieder schließen kann. Ein Spiel hat auch immer einen Spieler, weshalb wir das Interface **IPlayer** implementiert haben. Dieses gibt vor, dass ein Spieler immer einen Nickname und einen Score haben muss. Dadurch ist gewährleistet, dass ein Ergebnis in der Scoreliste gespeichert werden kann. Durch diese zwei Interfaces ist es möglich, bei einer neuen Spielidee ein weiteres Spiel zur Applikation hinzuzufügen und eine Minigame Applikation zu erschaffen.

Des Weiteren verwenden wir das Comparable<> Interface von Java, um unsere Spieler dem Score nach zu sortieren.

Vererbung

Vererbung verwenden wir bei unseren Hindernissen. Es gibt eine abstrakte Klasse *Obstacle*. Diese beinhaltet alle Methoden und Attribute, die ein Hindernis können muss und hat. Von dieser leiten wir dann unsere verschiedenen Hindernisse ab. Diese unterscheiden sich dann in Breite, Größe und Höhe, in der sie erscheinen. Ein paar Beispielklassen sind: *CactusRow*, *Mosquito*, *Rock* und *Shrub*.

Clean Code

Alle Klassen wurden so restriktiv wie nur möglich gestaltet. Attribute eines Objektes bekommt man durch *getter*-Methoden und dabei wiederum eine Kopie und nicht die direkte Referenz. Es gibt kaum eine statische Methoden. Java Code Konventionen wurden eingehalten.

GUI

Wie bereits erwähnt befinden sich die Controller für die GUI im Ordner **GUI**. Wir haben drei verschiedene Szenen. Das Menu, das Game und den Score. Deren Grundgerüste wurden alle mit FXML erstellt. Die FXML Dateien befinden sich im Ordner **resources.fxml**. Durch die Klasse *App* werden die Szenen miteinander verbunden. Zusätzlich gibt es eine Klasse *GameAnimationTimer*, welcher vom *AnimationTimer* erbt. Durch diesen ist es uns möglich unser Spiel dynamisch anzeigen zu lassen, da er unsere „Canvas“ fortlaufend aktualisiert.

Logging

Logger sind in allen wichtigen Klassen mit Funktion vorhanden. Wir verwenden verschiedene Log-Stufen. **Error** wird zum Beispiel verwendet, wenn es nicht möglich war das ScoreList-File zu lesen, da es nicht gefunden wurde oder beschädigt ist. Wenn dann zum Beispiel eine neue Scoreliste erstellt werden muss bekommt man dafür eine Meldung mit der Stufe **Info**. Info verwenden wir des Weiteren, wenn zum Beispiel ein neues Spiel gestartet wird, ein GameOver stattgefunden hat oder wenn die Scoreliste erfolgreich ins File gespeichert wurde. Wenn die *IllegalScoreException* fliegt, catchen wir diese und loggen in diesem Fall auch mit Info, da nur der gespielte Score nicht gespeichert werden konnte(Eventuell erwünscht vom Benutzer). Die Log-Stufe **Debug** verwenden wir unter anderem wenn und was für ein neues Hindernis erstellt wird. Außerdem wenn der Spieler seinen Status ändert, wie zum Beispiel beim Ducken oder Springen.

Exceptions

Wir haben eine *IllegalScoreException* implementiert, welche von *IllegalArgumentException* erbt. Es handelt sich also um eine *RuntimeException*. Diese fliegt, falls versucht wird einen illegalen Score zu speichern d.h. falls kein Nickname vorhanden ist, oder ein negativer Score erscheint, welcher nicht erspielbar ist.

UML

Das Klassendiagramm und das Use-Case Diagramm befinden sich auf den Seiten 3 und 4. Zusätzlich wurden sie als PDF ins Repository geladen.

Threads

Für unser Spiel verwenden wir verschiedene Timer für das Bewegen der Hindernisse, den Sprung des Spielers und die Animation. Jeder dieser Timer besitzt seinen eigenen Thread. Zusätzlich haben wir einen Timer erstellt, welcher jede Sekunde den Score um eins hochzählt. Ein anderer Timer, der sich um die Hindernisse kümmert, zählt ebenfalls den Timer um 10 hoch, nämlich für jedes Hindernis, welches der Spieler überwunden hat. Dieser Zugriff läuft synchronisiert ab. (Score in Klasse *Board*)

Streams und Lambda-Funktionen

Wir verwenden ParallelStream in unserer Klasse *ScoreList*. Wenn eine Anfrage durch *getScoreList*() kommt, kann in diesem Fall noch ein Suchbegriff mitgegeben werden. Je nachdem wird der Stream zuerst gefiltert, dann sortiert und danach durch einen Collector als Liste zurückgegeben. Lambda-Funktionen sind unter anderem bei den Streams verwendet wie auch beim KeyListener in der Klasse *GameController*.

Factories

Wir haben eine ObstacleFactory implementiert. Es gibt eine Klasse *ObstacleManager*, welche den Lebenszyklus und das Erstellen der Hindernisse verwaltet. Die Factory ist in der Methode *generate()* zu finden und erstellt zufällig verschiedene Hindernisse.

Dokumentation und Tests

Alle wichtigen Methoden und Codeschnipsel, die nicht selbsterklärend sind wurden dokumentiert.
Zum Testen wichtiger Funktionen haben wir JUnit Tests erstellt. Diese sind im Ordner **test** zu finden. Zu den Klassen haben wir jeweils korrespondierende Testklassen erstellt, die wichtige Funktionen der Klasse (auf Herz und Nieren) durchtesten. Es wurden ebenfalls Negativtests implementiert. Zum Beispiel wird bei der *ScoreList* getestet, ob wie zu erwarten eine *IllegalScoreException* fliegt, wenn versucht wird einen unzulässigen Score zu speichern.

Bewertungsbogen

Vorname	Nachname	Kürzel	Matrikeln	Projekt	Arc.	Clean Code	Doku	Tests	GUI	Logging/Except.	UML	Threads	Streams
Jonas	Leitner	jl121	37427	JumpOrDie	3	2	3	3	3	3	2	3	3
Christian	Heinz	ch148	37562	JumpOrDie	3	2	3	3	3	3	2	3	3
Agil	Homar	ah210	37478	JumpOrDie	3	2	3	3	3	3	2	3	3
Salome	Wecks	sw193	37495	JumpOrDie	3	2	3	3	3	3	2	3	3