24.1.2022

Tim Krauth

Projektdokumentation

M226b Tim Krauth

Inhalt

[Ausgangslage 2](#_Toc93941484)

[Projektmethode 2](#_Toc93941485)

[Informieren 3](#_Toc93941486)

[Planen 4](#_Toc93941487)

[Entscheiden 5](#_Toc93941488)

[Realisieren 5](#_Toc93941489)

[Abstrakte Klasse, Vererbung 5](#_Toc93941490)

[Interface 6](#_Toc93941491)

[Steuerung und Border Frogger 7](#_Toc93941492)

[Zustandsautomat 7](#_Toc93941493)

[Korrigieren 8](#_Toc93941494)

[Auswerten 10](#_Toc93941495)

# Ausgangslage

Im Rahmen des Moduls 226b haben wir den Auftrag erhalten, in Einzelarbeit ein auf Java basierendes Programm, mit dem Processing Framework zu erarbeiten.

Ich konnte das Programm frei wählen und habe mich für ein Spiel namens Frogger entschieden, das ich aus meiner Kindheit noch kannte. In dem Spiel geht es darum, dass man den Frosch über die Strasse und Bäume bis zur Seerose steuert, ohne dabei unter das Auto oder ins Wasser zu kommen.

Das Projekt wird mit einem Bewertungsraster bewertet und beinhaltet:

* Dokumentation
* Arbeitsjournal
* Zeitplan
* Code

# Projektmethode

Ich habe mich dazu entschieden, meine Dokumentation basierend auf IPERKA durchzuführen. Dies ist eine Projektmethode, welche sehr oft für Einzelarbeiten wie diese verwendet wird. Jeder der sechs Buchstaben aus dem Wort IPERKA steht für einen Arbeitsschritt.

* Informieren
* Planen
* Entscheiden
* Realisieren
* Korrigieren
* Auswerten

Damit diese Methode funktioniert, muss die Reihenfolge beachtet werden und darf kein schritt ausgelassen werden.

# Informieren

Beim Informieren, habe ich mir die Kriterien genau angeschaut und überlegt, was ich alles in meinem Projekt brauche.

Anhand der Kriterien bewertet werden:

* Eine Fachliche Dokumentation
* Die Dokumentation innerhalb des Codes
* Die Codeversierung
* Das Arbeitsjournal
* Die Zeitplanung

Danach habe ich mir über die Umsetzung Gedanken gemacht und mein UML-Klassendiagramm gemacht

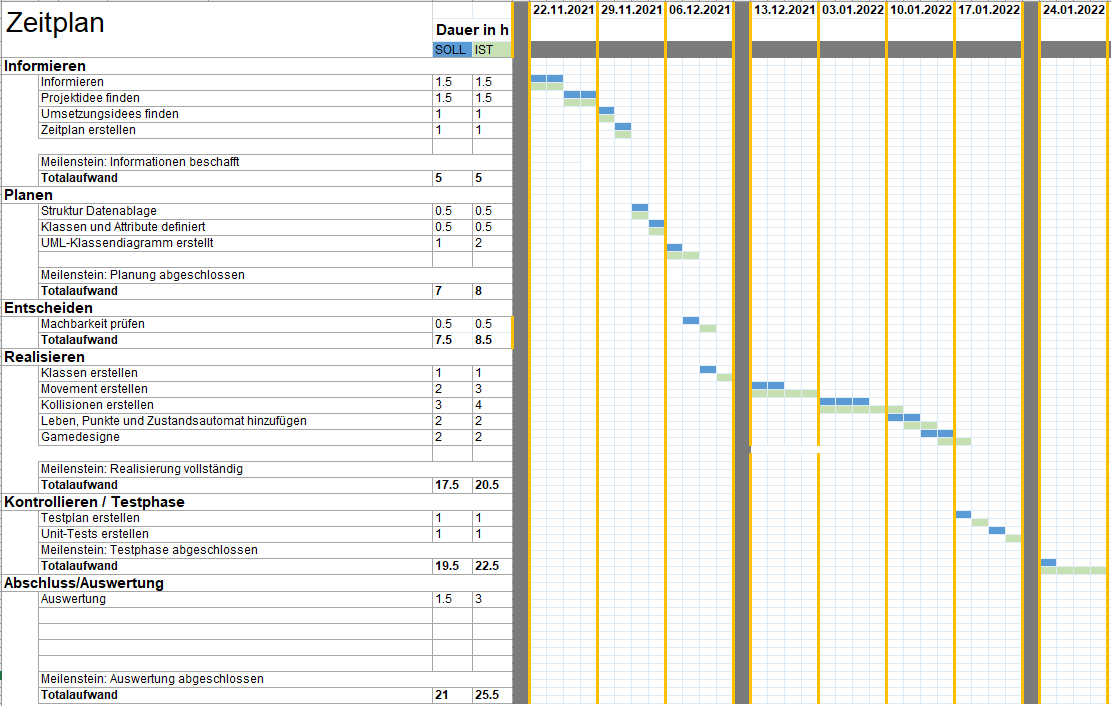
Bild Klassendiagram

# Planen

Als nächster Schritt kommt die Planung meines Projekts.

Ich musste mir überlegen, wie ich mir meine Zeit für mein Projekt einteile, um in dem vorgegebenen Zeitrahmen fertig zu werden.

Aus diesem Grund habe ich einen Zeitplan erstellt, bei dem für mich ersichtlich ist wie gut ich vorankomme und welche Aufgaben auf mich zukommen werden.



# Entscheiden

Nachdem die Planung für das Projekt abgeschlossen ist, geht es um das Entscheiden.

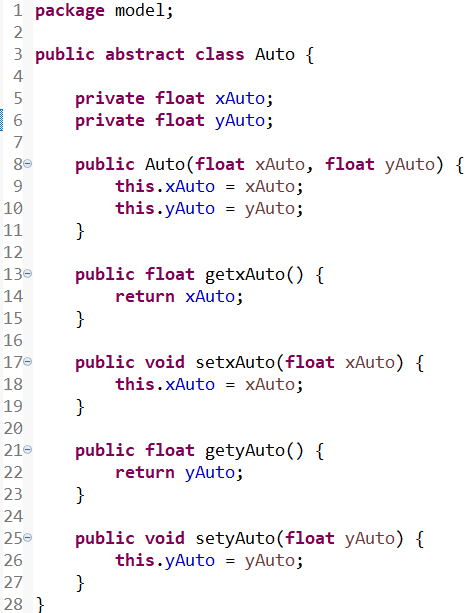
Für das Projekt sind uns viele Sachen wie die Kriterien, Programmiersprache und Framework vorgegeben, was mir viele Entscheidungen abgenommen hat. Mithilfe des UML-Klassendiagramms und dem Zeitplan bin ich überzeugt, dass meine Projektplanung funktionieren wird.

# Realisieren

Nun geht es ans Realisieren. Hierfür habe ich auf Github ein neues Projekt erstellt, mit dazugehörigem Ordner für die Doku, den Zeitplan und meine Tagesjournale.

Danach habe ich meinen Code geschrieben. Obwohl dies der wichtigste Schritt ist, funktioniert er nur, wenn die vorherigen gut abgeschlossen wurden.

## Abstrakte Klasse, Vererbung

 Warum Abstrakt? Auto-Klasse:

Ich habe mich für eine Abstrakte Klasse Auto entschieden, weil ich drei einzelne Klassen für meine Auto-Objekte erstellen wollte und deshalb in dieser Klasse keine Objekte habe.

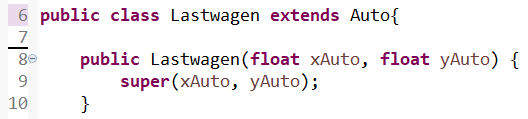
Warum die Vererbungen?

Da ich drei Klassen mit gleichen Funktionen und gleichen Attributen hatte, war es am einfachsten eine Klasse zu erstellen, die ihnen die x und die y-Koordinaten übergibt.

So konnte ich mir etwas Code sparen und hatte die sauberste Lösung dafür.

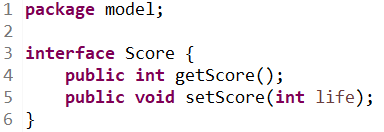
Mithilfe der Getter und Setter, können die Klassen

* Lastwagen
* AutoLinks
* AutoRechts

auf die x und y-Koordinaten zugreifen. Lastwagen Klasse:  
So brauchen sie nicht ihr eigenes Koordinatensystem.

## Interface

Bei meinem Interface, habe ich mich für ein Interface namens Score entschieden.



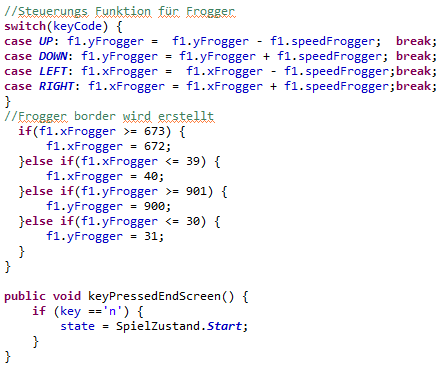
Es wird für zwei meiner Klassen, die fürs zählen verantwortlich sind genutzt.

* Highscore (Verantwortlich für Punktestand)
* Leben (Verantwortlich für Leben)

Ich habe mich hier zwecks Übersicht und Ähnlichkeit der Klassen für ein Interface entschieden.

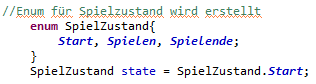


## Steuerung und Border Frogger

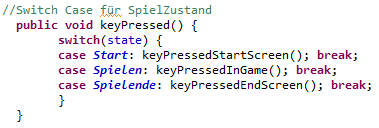
Die Steuerung für den Frogger habe ich mit einem “switch keyCode“ gemacht.   
Mit UP, Down, Left und Right werden direkt die Pfeiltasten angesprochen.

Die Border wird mit einem maximalen und einem minimalen X, sowie Y gesetzt, die der Frogger nicht überschreiten kann, da er sonst auf einen fixen Punkt zurückgesetzt wird.

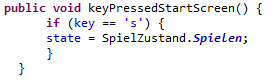
## Zustandsautomat

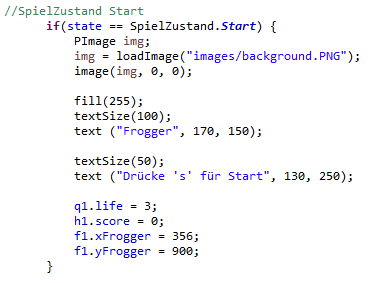


Als ersten Schritt zu meinem Zustandsautomaten, wird ein Enum erstellt. Darin enthalten sind Start, Spielen und Spielende.  
Dann wird der State auf SpielZustand.Start gesetzt.



Danach wird eine Switch Case Funktion erstellt, bei der sich die KeyPressed funktion je nach Spielzustand ändert.



Als Beispiel die KeyPressed Funktion des Startscreens. Hier kann man mit dem drücken auf die “s“ Taste in den Spielzustand Spielen wechseln.

Der letzte Snippet beinhaltet die Draw-Funktion für den Spielzustand Start und die Werte, die wieder auf ihren Ausgangswert gesetzt werden.

# Korrigieren

In diesem Schritt wird mein Programm getestet, es werden viele Szenarien durchgegangen, so dass ein User keine Fehler machen kann.

Testplan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Titel | Beschreibung | Soll-Ergebnis | Ist-Ergebnis | Status |
| Zustandsautomat wechseln, von Start ins Spiel | Man kommt ausschliesslich mit dem drücken der ‘s’ Taste ins Spiel. | Eingabe ‘s’ um ins Spiel zu kommen | Man kommt mit der Taste ‘s’ vom Start ins Spiel | OK |
| Zustandsautomat wechseln, von Ende zum Start | Man kommt ausschliesslich mit dem drücken der ‘n’ Taste zum Start zurück. | Eingabe ‘n’ um ins Spiel zu kommen | Man kommt mit der Taste ‘n’ vom Ende zum Start des Spiels | OK |
| Zustandsautomat wechsel, von Spiel auf Ende | Wenn Frogger alle Leben verliert kommt man zum Ende | Wenn Leben weg sind ist das Spiel fertig | Wenn Leben aufgebraucht sind kommt man zum End-Bildschirm | OK |
| Steuerung Frogger | Frogger lässt sich mit den vier Pfeiltasten steuern | Frogger kann mit den Pfeiltasten in alle vier Richtungen fahren | Frogger kann mit den Pfeiltasten in alle vier Richtungen fahren | OK |
| Frogger Kollision Autos | Frogger muss mit jedem Auto kollidieren | Frogger wird bei jeder Kollision zurück auf Start gestellt und bekommt ein Leben abgezogen | Frogger kommt auf Start nach Kollision und bekommt ein Leben abgezogen | OK |
| Frogger Kollision Baum | Frogger kann auf jedem Baum stehen ohne auf Start zurück zu kommen | Frogger kann auf Baum bleiben | Fosch kann auf Baum stehen | OK |
| Frogger Kollision Seerose | Wenn Frogger eine Seerose berührt, soll er zurück auf Start und einen Punkt bekommen | Bei Berührung von Frogger und einer Seerose kommt er zurück auf Start mit einem Punkt mehr | Wenn Frogger auf eine Seerose kommt, ist er zurück auf Start mit einem Punkt mehr | OK |

Test dokumentation

JUnit-Tests

# Auswerten

Im letzten Schritt werden meine Arbeit und das Endergebnis ausgewertet, konkret heisst das, dass die Ziele nochmals reflektiert werden, um sicherzustellen, dass sie alle erreicht wurden. Zudem wird eine Reflexion über das ganze Projekt geschrieben.

Selbsteinschätzung Note