Konzeptskizze für Zyklus 1

Beteiligte Person:

Name	Matrikelnummer
Marlon Cadel	1276264
Sascha Hahn	1320857
Mohammed Tarek Dawalibi	1284854

Inhaltsverzeichnis:

1 Konzeptskizze für “Speichern und Laden”

Beschreibung der Aufgabe

In dieser Aufgabe sollen Sie Fallen implementieren, die das Voranschreiten im Dungeon schwieriger machen.

Beschreibung der Lösung

Der Dungeon wird gespeichert, sobald ein neues Level geladen wird. Die Entities und das Level werden in einer Datei gespeichert. Beim erneuten Starten des Spiels wird der Spielstand, wenn es einen gibt, geladen.

Methoden und Techniken

Der Code wird entsprechend der Java-Doc Vorgaben Dokumentiert.

Ansatz zur Modellierung

Wir erstellen eine Klasse "Save", welche die statischen Methoden laden() und speichern() beinhaltet. Die laden() Methode wird dann in der steup() Methode der Game Klasse aufgerufen und die speichern() Methode wird in der onLevelLoad() Methode aufgerufen.

Die zu speichernden Klassen erben von der Klasse/dem Interface Serialization. Daraufhin werden neue Dateien/neue Dateien erstellt/geöffnet, um die Zustände zu speichern. Beim schließen des Spiels wird automatisch Gespeichert und beim start des Spiels, wird der Inhalt der Dateien/ der Datei geladen und den passenden Objekten zugewiesen.

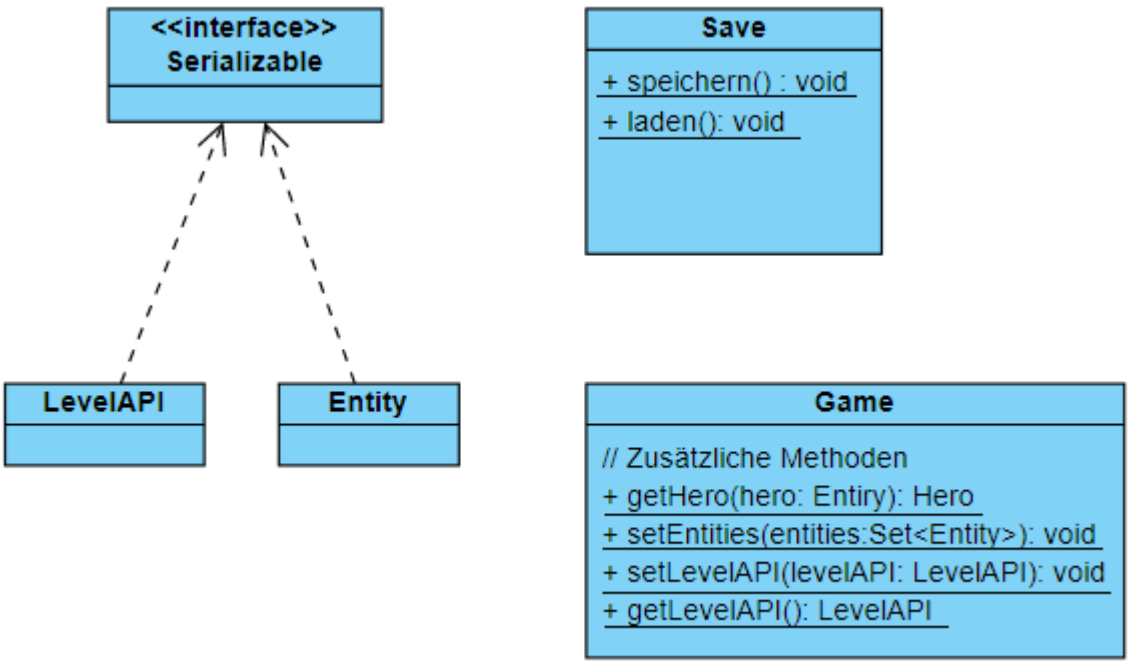
Die Game Klasse muss auch noch mit set() und get() Methoden erweitert werden, um auf die Attribute hero, entities und level API zuzugreifen.

Grundfunktionen:

```java
public class Save
{
    public static void speichern()
    {
        Entity hero = Game.getHero();
        Set<Entity> entities = Game.getEntities();
        LevelAPI api = Game.getAPI();
        // implementation
    }

    public static void laden()
    {
        // implementation
    }
}
```

Daraus egibt sich folgendes UML Diagram:



2 Konzeptskizze für “Monster”

Beschreibung der Aufgabe:

Es müssen 3 Monster erstellt werden, die Monster zeichnen sich dadurch aus, dass jedes der Monster eine unterschiedliche Fähigkeit besitzt (eigenes Verhalten/Grafik/Animationen).

Beschreibung der Lösung:

Die Position, sowie auch die Bewegung der Monster ist zufällig (horizontal und vertikal). Je weiter ein Spieler im Dungeon voranschreitet, desto mehr Monster erscheinen und desto stärker werden die Monster (mehr Schaden). Die Anzahl der Monster ist bei jedem Level Start zufällig verteilt. Jedes Monster hat eine Animation.

Methoden & Techniken:

Zum Einsatz kommt JavaDocs, zur besseren Übersicht im Code.

Ansatz und Modellierung:

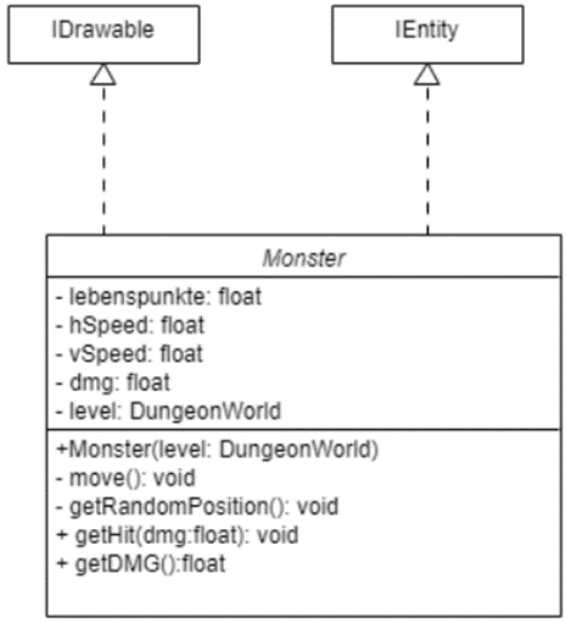
Zunächst wird eine abstrakte Klasse erzeugt, welche von der Klasse Entity erbt und als Unterklassen dann an die einzelnen verschiedenen Monsterklassen vererbt. Somit hat jedes Monster einen Pool von Attribute:

- float lebenspunkte: Leben des Monsters
- float xSpeed: Die Geschwindigkeit, in der sich das Monster horizontal bewegt
- float ySpeed: Die Geschwindigkeit, in der sich das Monster vertikal bewegt
- float dmg: Den Schaden, den das Monster im Kampf macht
- dungeonWorld level: Die Monster müssen die schwierigkeit/index des Levels kennen um progressiv stärker zu werden.

void move(): Bewegt das Monster in eine zufällige Richtung

- void getHit(float dmg): Zieht dem Monster Lebenspunkte ab
- float getDMG(): Gibt Schaden zurück
- getRandomPosition(): Wird im Konstruktor aufgerufen, sucht sich eine zufällige Position im Dungeon als Spawnpunkt

UML:



Beschreibung der Monster:

1. **Zombie:**

Lebenspunkte: 10

Schaden: 1

Bewegung x,y: 0.06, 0.1

2. **Wolf:**

Lebenspunkte: 5

Schaden: 3

Bewegung x,y: 0.5, 0.5

3. **Mumie:**

Lebenspunkte: 3

Schaden: 2

Bewegung x,y: 0.2, 0.2

Die Methode spawnMonster() wird aufgerufen, um die Monster im Dungeon zu verteilen. Dabei wird unter Berücksichtigung des aktuellen Dungeon Levels die Anzahl der Monster zufällig gewählt.

Verlässt der Spieler das Level, werden alle Monster aus dem EntityController gelöscht. Dafür wird die gesamte Liste des EntityController gelöscht und der Held neu hinzugefügt.

3 Konzeptskizze für "Fallen"

Beschreibung der Aufgabe

In dieser Aufgabe sollen Sie Fallen implementieren, die das Voranschreiten im Dungeon schwieriger gestalten.

Beschreibung der Lösung

Die Fallen werden random in dem Level gespawnt. Die Anzahl dieser ist auch random in einer bestimmten Range, welche von der Größe des Levels abhängt. Fallen können von Entitäten aktiviert werden, wobei sie eine aktivierte Animation anzeigen. Nach der Standard Animation hat die Falle auch eine Animation zum Verschwinden, da sie eine bestimmte Anzahl an Aktivierungen funktioniert und anschließend verschwindet/kaputt geht.

Die Fallen können außerdem auch unsichtbar, komplett sichtbar oder teilweise sichtbar sein, wofür es auch verschiedene Animationen gibt.

Eine Falle kann Entitäten mit Lebenspunkten schaden zufügen, Entitäten teleportieren, welche eine Velocity Component haben, oder Entitäten, welche nicht der Hero sind, an einer bestimmten oder random Position des Levels spawnen.

Beim progressiven Durchgehen der Level, sollen die Fallen mehr Schaden verursachen oder in anderer Weise schwieriger werden bei anderen Features in der Zukunft. Zuletzt soll es auch Schalter gebene die, die Fallen deaktivieren können.

Methoden und Techniken

Der Code wird entsprechend der Java-Doc Vorgaben Dokumentiert.

Ansatz zur Modellierung

Eine Abstrakte Oberklasse "Fallen" trägt die Grundeigenschaften(Attribute und Components) für alle Fallen. Components wären PositionComponent und HitBoxComponent. Components die noch implementiert werden müssen wären AnimationComponentTrap und ActivationComponent. Variablen sind: "levelcounter", welcher die Schwierigkeit des Levels speichert und "visibility" welche die Sichtbarkeit angibt.

Um den "levelcounter" passend zu ändern hat die Klasse eine static public Methode increaseLevel(). Diese wird am Ende/Anfang von onLevelLoad() ausgeführt.

Die Hauptfunktionen werden durch die Unterklassen, TrapDamge, TrapTeleport und TrapSpawn umgesetzt. Diese brauchen entsprechende Components um ihre Funktionen umzusetzen.

Zum Ende werden auch weitere Systems gebraucht, welche den Spawn, sowie bereits existierende Systems wie DrawSystem, CollisionSystem, HealthSystem und VelocitySystem.

Daraus ergibt sich folgendes UML Diagram: