Projektdokumentation – Projekt Software Engineering

Erstellt von: Jan Wunderlich
Matrikelnummer: 92205002
Studiengang: Master of Science Informatik
Tutor: Prof. Dr. Markus Kleffmann
Datum: 28.12.2023

Inhaltsverzeichnis

1 Vorgehensmodell

Im Rahmen der Entwicklungsphase für das vorliegende Projekt wird die agile Methode Scrum eingesetzt. Scrum ist ein bewährtes und flexibles Vorgehensmodell, das sich besonders gut für Softwareprojekte eignet, bei denen sich Anforderungen ändern können. Die Wahl von Scrum als Vorgehensmodell für die Erstellung des Endless-Runner-Spiels erfolgt aufgrund spezifischer Überlegungen, die die besonderen Anforderungen dieses Vorhabens berücksichtigen.

Scrum zeichnet sich durch eine hohe Flexibilität aus, die besonders der Natur von Endless-Runner-Games entgegenkommt. Diese Spiele unterliegen häufig sich ändernden Anforderungen, sei es zur Verbesserung des Gameplays, zur Integration neuer Features oder zur Anpassung aufgrund von Testergebnissen. Scrum ermöglicht dementsprechend eine agile Reaktion auf diese Veränderungen, indem sie in den Product-Backlog integriert und in den kurzen Sprints priorisiert werden.

Die iterative und inkrementelle Struktur von Scrum stellt sicher, dass die Implementierung des Endless-Runner-Spiels in handhabbare Sprints unterteilt werden kann. Dies ermöglicht nicht nur eine kontinuierliche Entwicklung, sondern auch die schnelle Sichtbarkeit von Fortschritten. In kurzen Entwicklungszyklen können so regelmäßig spielbare Teile des Spiels erstellt, getestet und verbessert werden. Die Sprintdauer für dieses Projekt wird auf 5 Tage festgelegt, um eine ausgewogene Balance zwischen Effizienz, Flexibilität und regelmäßiger Leistungsbewertung sicherzustellen. Ein weiterer Vorteil von Scrum liegt im klaren Fokus auf die Benutzererfahrung und die kontinuierliche Anpassung des Produkts. Diese Aspekte sind besonders relevant für die Entwicklung eines Endless-Runner-Games, bei dem die Spielerfahrung im Mittelpunkt steht. Die Möglichkeit, Gameplay, Grafiken und Benutzeroberfläche iterativ zu verbessern, sorgt dafür, dass das Spiel am Ende den Erwartungen der Spieler entspricht.

Obwohl Scrum ursprünglich für die Teamarbeit konzipiert wurde, bietet es auch für Einzelentwickler klare Vorteile. Eine transparente Kommunikation sowie regelmäßige Selbstreflexion ermöglichen es, den eigenen Fortschritt zu bewerten, Schwierigkeiten zu identifizieren und gezielte Anpassungen vorzunehmen. Die Verwendung eines Kanban-Boards und des Product-Backlogs innerhalb des Scrum-Frameworks unterstützt zudem die effiziente Verwaltung von Aufgaben und Prioritäten. Das Kanban-Board verbessert die Übersichtlichkeit und zeigt den Status aller Aufgaben des aktuellen Sprints an. Das Product-Backlog dient als zentrale Sammelstelle für Ideen und Aufgaben, die in den kommenden Sprints organisiert werden.

Im Projekt wird die agile Methode Scrum als Einzelperson angewandt, wobei alle drei Scrum-Rollen, also Scrum Master, Product Owner und Developer, in Personalunion ausgeführt werden. In der Rolle des Scrum Masters liegt der Fokus auf klarer Kommunikation, der Lösung von Hindernissen und der Gewährleistung eines reibungslosen Ablaufs. Als Product Owner werden Backlog-Priorisierung, Anforderungsformulierung und die Betonung des Endproduktwerts übernommen. Als Developer wird technisches Know-how für die letztendliche Umsetzung eingebracht. Eine transparente

Kommunikation, regelmäßige Selbstreflexion und effiziente Tools ermöglichen die Dynamik der Scrum-Rollen auch als Einzelperson optimal zu gestalten.

Insgesamt schafft Scrum eine ausgewogene und flexible Umgebung, maßgeschneidert für die Herausforderungen der Einzelentwicklung eines Endless-Runner-Spiels. Es ermöglicht nicht nur eine effiziente und iterative Entwicklung, sondern auch die stetige Anpassung des Spiels, um eine optimale Spielerfahrung zu gewährleisten. Die kurzen, regelmäßigen Iterationen fördern dabei eine dynamische Anpassung an sich verändernde Anforderungen und eine kontinuierliche Verbesserung der Spielerzufriedenheit im Verlauf des Entwicklungsprozesses.

2 Technologien und Tools

Um eine effiziente und erfolgreiche Umsetzung des Projekts zu gewährleisten, werden in der Entwicklungsphase verschiedene Technologien, Tools und Frameworks eingesetzt. Als Programmiersprache wird Python gewählt, da persönliche Erfahrungen mit dieser Sprache vorliegen. Python bietet eine klare und leserliche Syntax, was die Entwicklung und Wartung erleichtert. Zudem verfügt Python über umfangreiche Bibliotheken, die speziell für die Spieleprogrammierung genutzt werden können. Pygame (siehe https://www.pygame.org/wiki/about) wird als Bibliothek für die Spieleentwicklung integriert, da es sich als äußerst geeignet für die Umsetzung eines Endless-Runner-Games erwiesen hat. Pygame bietet Funktionen und Werkzeuge, die besonders auf die Anforderungen von 2D-Spielen zugeschnitten sind. Die Bibliothek vereinfacht die Handhabung von Grafiken, Animationen, Kollisionserkennung und andere relevante Aspekte der Spieleentwicklung. Die einfache Integration von Bildern, Sounds und weiteren Ressourcen macht Pygame zu einer optimalen Wahl für die Umsetzung eines Endless-Runner-Spiels.

Für die IDE bzw. Entwicklungsumgebung wird PyCharm verwendet. Diese Wahl beruht auf persönlichen Erfahrungen und Vorlieben im Umgang mit PyCharm. PyCharm bietet verschiedenen Funktionen wie Code-Intelligenz, Debugging-Tools oder eine Codeformatierung unter Beachtung des integrierten Standards PEP-8 für Python, die bei der Entwicklung und Fehlersuche helfen. Als zentrales Tool für die Umsetzung des Scrum-Vorgehensmodells wird JIRA eingesetzt. JIRA vereint verschiedene Funktionen, die für die agile Softwareentwicklung entscheidend sind. Es ermöglicht die Verwaltung des Product-Backlogs, die Planung von Sprints und die Organisation bzw. Nachverfolgung von Aufgaben durch ein Kanban-Board. Durch die Integration von JIRA wird eine transparente und strukturierte Umsetzung des Scrum-Frameworks gewährleistet, was die effektive Planung und Durchführung des Entwicklungsprozesses ermöglicht. Für das vorliegende Projekt respektive das zu implementierende Endless-Runner-Game wurde ein neues eigenständiges Projekt in JIRA angelegt, welches unter folgendem Link zu erreichen ist: https://pse-endlessrunnergame.atlassian.net/jira/software/projects/PSE/boards/1/reports/burnup.

Als Versionierungstool wird GitHub bzw. Git eingesetzt. GitHub ist eine Plattform, die auf Git basiert und eine zentrale sowie kollaborative Umgebung für die Codeverwaltung bietet. Die Integration von Git ermöglicht eine effiziente Verwaltung von Code, das Nachvollziehen von Änderungen sowie das Wechseln von verschiedenen Entwicklungsständen, sollte ein entsprechender Bedarf bestehen. Für das Endless-Runner-Game wurde bereits ein neues GitHub Repository erstellt, das unter https://github.com/jan-wun/PSE_Endless-Runner-Game betrachtet werden kann.

Die bewusste Auswahl von Technologien und Tools, darunter Python als Programmiersprache, Pygame für die Spieleentwicklung, PyCharm als IDE, JIRA für das Scrum-Vorgehensmodell und GitHub als Codeversionierungstool, schafft eine harmonische Tool-Landschaft. Diese Tools unterstützen nicht nur die spezifischen Anforderungen des Projekts, sondern gewährleisten auch einen reibungslosen Entwicklungsprozess durch effektive Codeverwaltung und agile Planung.

3 UML-Diagramme

3.1 Klassendiagramm

Zur Visualisierung der Struktur des geplanten Endless-Runner-Spiels wird ein UML-Klassendiagramm verwendet. Dieses Diagramm bietet einen klaren und verständlichen Überblick über die Klassen, ihre Attribute, Methoden und die Beziehungen zwischen ihnen. Durch die Nutzung eines UML-Klassendiagramms kann folglich ein schneller Einblick in die Architektur des Softwareprojekts gewonnen werden. Aus Übersichtlichkeitsgründen enthält das Diagramm für das vorliegende Software Engineering Projekt, das auf der nächsten Seite im Querformat dargestellt ist, nicht sämtliche Attribute, Methoden und Beziehungen zwischen allen Klassen, sondern beschränkt sich auf das Wichtigste.

Die Struktur des Endless-Runner-Spiels wurde im UML-Klassendiagramm sorgfältig entworfen, um die verschiedenen Komponenten und ihre Interaktionen übersichtlich darzustellen. Die Hauptklassen für die Realisierung des Gameplays sind *Game*, *Player*, *Enemy*, *PowerUp*, *Obstacle*, *Weapon* und *Projectile*. Das *Game*-Objekt koordiniert alle Abläufe im Spiel, während die anderen Klassen spezifische Spielentitäten repräsentieren und von der Klasse *Entity* gemeinsame Attribute und Methoden vererbt bekommen. Die Klasse *Player* modelliert die Spielfigur, die Klassen *Enemy* und *Obstacle* stehen für Gegner und Hindernisse, *Weapon* modelliert die verschiedenen Waffen im Spiel, *Projectile* die Schüsse und die Klasse *PowerUp* die verschiedenen Items, welche das Spiel bietet.

Zusätzlich dazu zeigt das Diagramm verschiedene Menüs, die allesamt von der Klasse *Menu* erben. Die einzelnen Menüs stellen verschiedene Seiten respektive Zustände im Spiel dar. Auch wenn alle Menüs von *Menu* erben, sind lediglich die fünf Klassen *MainMenu*, *SettingsMenu*, *ShopMenu*, *ControlsMenu* und *StatsMenu* nach dem gleichen Layout aufgebaut. Das Game-Over-Menü und das Pause-Menü unterscheiden sich absichtlich, um diese Menüs thematisch bzw. logisch besser vom Rest abzugrenzen zu können.

Neben den Klassen für die Implementierung der Spiellogik und den Menüs beinhaltet das Diagramm noch zwei weitere wichtige Klassen, nämlich die Klasse *SaveLoadManager* und die Klasse *Assets*. Während Erstere zum Speichern und Laden von Spieldaten, die vor einer Abänderung abgesichert sind, verwendet wird, dient Letztere zum Einladen sämtlicher Spielressourcen wie Bildern, Audio-Dateien oder Schriftarten. Außerdem werden in dieser Klasse Konfigurationsparameter eingelesen, über welche, verschiedenste Spielvariablen, die direkte Auswirkungen auf das Spielerlebnis haben, festgelegt werden können.

Darüber hinaus beinhaltet das Diagramm verschiedene Enums. Die drei Enums *GameState*, *PlayerState* und *EnemyState* bilden den aktuellen Zustand des Spiels, Spielers oder der jeweiligen Gegner ab. Diese Zustände werden innerhalb der entsprechenden Klasse verwaltet und beeinflussen das Spielgeschehen entscheidend. Diese klaren Hierarchien und Verbindungen ermöglichen eine bessere Strukturierung und Wartbarkeit des Codes während der Entwicklung.

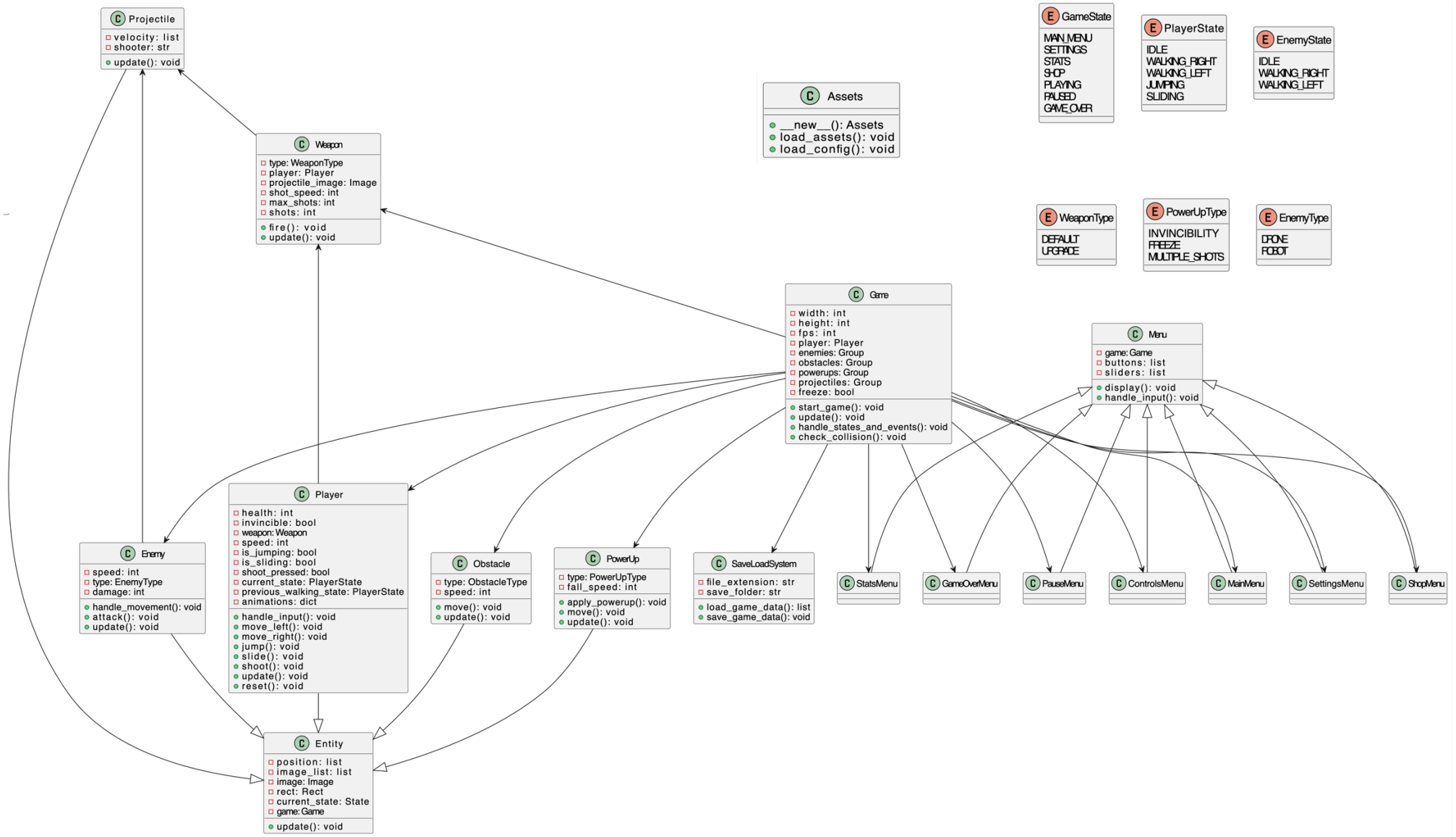

Abbildung 1: UML-Klassendiagramm

3.2 Aktivitätsdiagramm

Um den zentralen Ablauf der Hauptspiel-Schleife in der Game-Klasse zu visualisieren, wird ein UML-Aktivitätsdiagramm eingesetzt, welches in Abbildung 2 dargestellt ist. Die Hauptspiel-Schleife stellt das Kernstück des Spiels dar und orchestriert die wesentlichen Prozesse, die für das reibungslose Funktionieren des Spiels erforderlich sind. Das Aktivitätsdiagramm dient dazu, den iterativen Charakter der Hauptspiel-Schleife und seine wesentlichen Elemente zu verdeutlichen. Es beginnt mit der Initialisierung des Spiels, einschließlich der Konfiguration von Spielobjekten, Zuständen und Variablen. Anschließend erfolgt eine dauerhafte Schleife, die so lange läuft, wie der Spieler sich in der Anwendung bzw. im Spiel befindet und dieses nicht aktiv verlassen hat. Innerhalb dieser Schleife werden Zustände und Ereignisse behandelt, um den aktuellen Spielzustand zu überwachen.

Wenn sich das Spiel im Zustand „PLAYING“ befindet, werden die Spielobjekte aktualisiert und gerendert, um eine nahtlose Spielerfahrung zu gewährleisten. Falls das Spiel in den Zustand „GAME_OVER“ wechselt, wird der Game Over-Bildschirm angezeigt. An dieser Stelle hat der Benutzer die Möglichkeit, das Spiel zu beenden oder es neu zu starten. Für den Fall, dass der Spieler das Spiel verlassen möchte, wird das Spiel geschlossen und die Schleife wird nicht erneut ausgeführt. Bei einem Neustart werden hingegen alle erforderlichen Schritte durchgeführt, um das Spiel in einen spielbaren Zustand zurückzuversetzen. Das Diagramm endet mit einer Schleifenrückkehr, die sicherstellt, dass der Ablauf in der Hauptspielschleife fortgesetzt wird, solange das Spiel läuft.

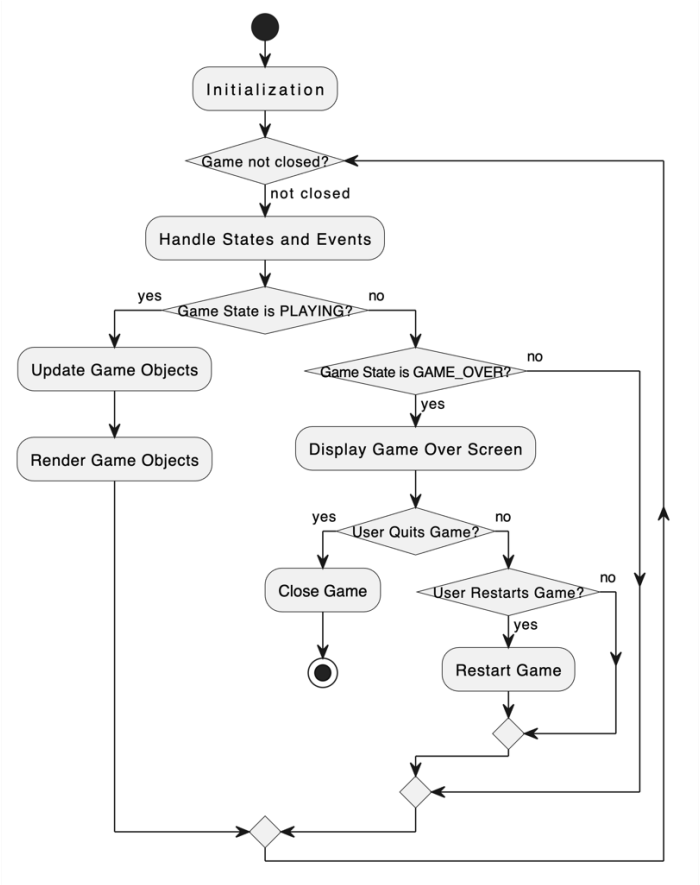

Abbildung 2: UML-Aktivitätsdiagramm

3.3 Sequenzdiagramm

Das in Abbildung 3 auf der nachfolgenden Seite illustrierte UML-Sequenzdiagramm zeigt die Interaktion zwischen dem Benutzer, der durch den *User*-Akteur repräsentiert wird, der *Player*-Klasse und der *Game*-Klasse. Das Diagramm dient dazu, den Fluss der Benutzerinteraktionen mit der Spielfigur im Spiel zu visualisieren. Aus Übersichtlichkeitsgründen ist das Sequenzdiagramm bewusst vereinfacht dargestellt und sämtliche Bedingungen, die erfüllt sein müssen, damit eine Tastatureingabe des Benutzers auch eine entsprechende Aktion der Figur nach sich zieht, wurden vernachlässigt. Diese können dem Quellcode sowie den dazugehörigen erklärenden Kommentaren in ebendiesem entnommen werden.

Das Sequenzdiagramm beginnt damit, das der Benutzer das Spiel startet, was durch den Aufruf der Methode *start_game()* geschieht. Daraufhin aktiviert sich die Klasse *Game* und führt die Methode *handle_states_and_events()* für jedes auftretende bzw. identifizierte Event aus. In dieser Methode werden allgemeine Spielzustände und Ereignisse behandelt. Im Anschluss daran wird überprüft, ob der aktuelle Zustand das Spielen ist. Bei positivem Ergebnis erfolgt die Ausführung der Funktion *update()*, welche die Spielobjekte aktualisiert. Zudem wird die Klasse *Player* aktiviert und über die klasseneigene Methode *update()* die Funktion *handle_input()* aufgerufen, die auf Benutzereingaben lauscht.

Das Diagramm zeigt dann die Verarbeitung von Benutzereingaben für verschiedene Tasten, darunter die vier Pfeiltasten (LEFT_ARROW, RIGHT_ARROW, UP_ARROW, DOWN_ARROW) und die Leertaste (SPACE). Jede dieser Eingaben führt zu entsprechenden Aktionen in der Klasse *Player*, also zu einer Bewegung des Charakters nach links oder rechts, zu einem Sprung, zum Sliden respektive Rutschen oder zum Abfeuern eines Schusses. Das Diagramm illustriert die User-Interaktionen dementsprechend beispielhaft für ein konkretes Szenario, in dem der Spieler die vorgenannten Aktionen in exakt dieser Reihenfolge ausführen muss, um zu überleben. Denkbar wäre ein solches Szenario im Spielkontext folgendermaßen: Der Spieler bewegt sich zunächst nach links, sieht einen Schuss der Drohne von oben kommen und bewegt sich dann nach rechts, um diesem Schuss auszuweichen. Anschließend muss er erst über einen Meteoriten springen und dann sliden, um nicht ein entgegenkommendes Auto zu berühren. Zu guter Letzt, wird ein aufgetauchter Roboter abgeschossen, damit dieser keine weiteren Probleme verursacht.

Nach jeder Benutzerinteraktion wird die Methode *update_animation()* aufgerufen, um Animationen zu aktualisieren. Animation werden dadurch realisiert, dass verschiedene Bilder der Spielfigur in schneller Geschwindigkeit hintereinander angezeigt werden. Neben den reinen Spielaktionen werden also auch visuelle Aspekte berücksichtigt, um die verschiedenen Interaktionsmöglichkeiten mit dem Charakter besser voneinander unterscheiden zu können und dem Benutzer allgemein eine bessere Spielerfahrung zu bieten.

Nach diesem Punkt würde sich der Zyklus, anders als im Diagramm einfachheitshalber dargestellt, ab der Methode handle_states_and_events() wiederholen. Diese Schleife setzt sich fort, solange

das Spiel läuft. Durch die zyklische Natur wird der fortlaufende Prozess von Benutzerinteraktionen und Spielaktualisierungen während des Spiels repräsentiert.

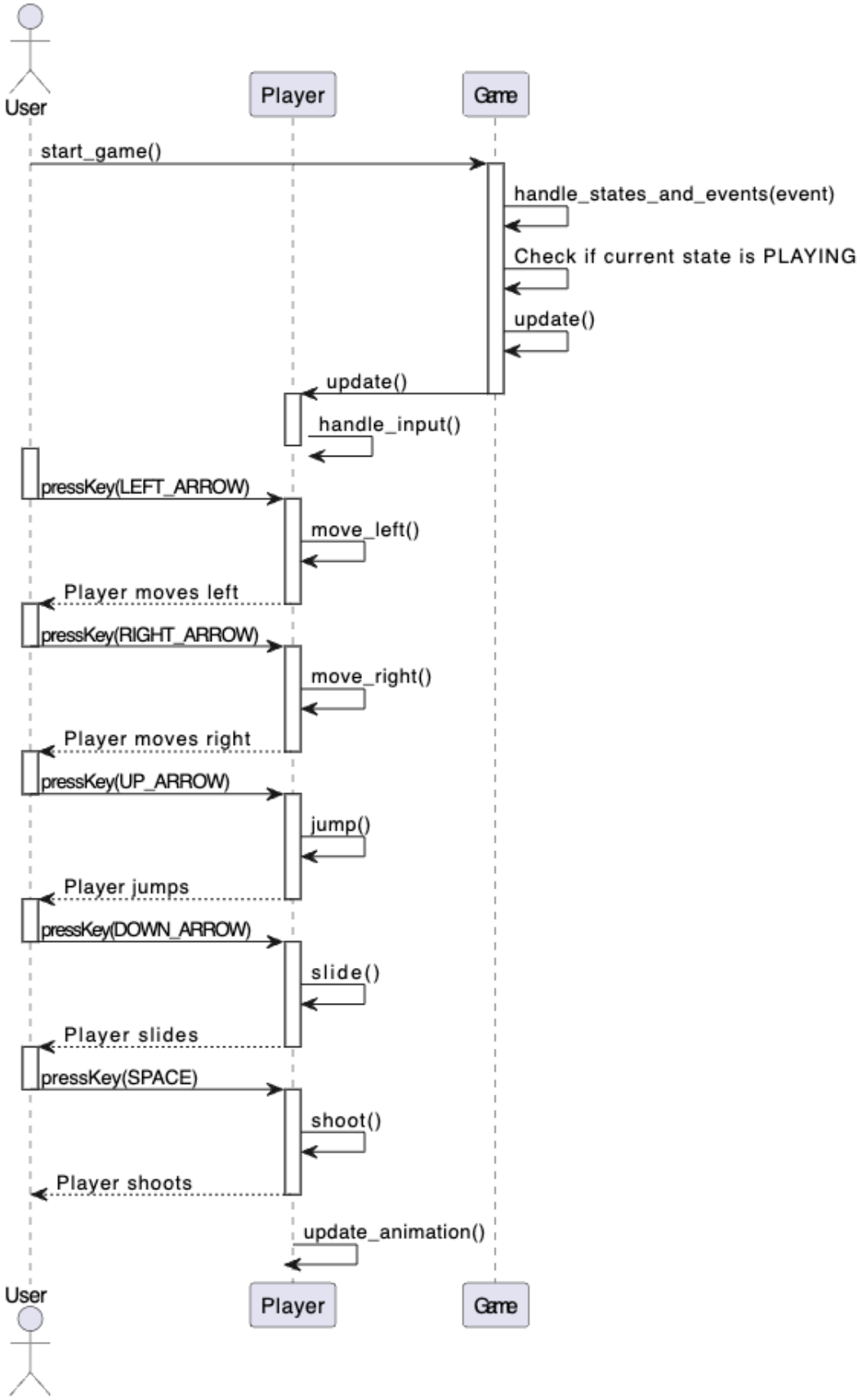

Abbildung 3: UML-Sequenzdiagramm

4 Design- und Implementierungsentscheidungen

4.1 Singleton Entwurfsmuster

Bei der Implementierung des Endless-Runner-Spiels wird auf das Singleton-Entwurfsmuster zurückgegriffen, um eine effiziente und konsistente Verwaltung sämtlicher Spielressourcen zu gewährleisten. Das Singleton-Muster gehört zur Kategorie der Erzeugungsmuster und stellt sicher, dass lediglich eine Instanz einer Klasse existiert, was wiederum globalen Zugriff auf diese zentralen Ressourcen ermöglicht. Im vorliegenden Projekt ist die Klasse *Assets* als Singleton implementiert.

Der Mechanismus des Singleton-Musters wird durch die geschickte Anwendung der __new__ -Methode realisiert. Diese Methode sorgt dafür, dass zu jedem Zeitpunkt nur eine einzige Instanz der klasse existiert, was die Konsistenz und Koordination der Spielressourcen optimiert. Diese Ressourcen umfassen nicht nur Bilder für Spieler, Hindernisse, Gegner, Menüs und Power-Ups, sondern schließen auch Audiodateien für Musik und Soundeffekte mit ein. Darüber hinaus werden Konfigurationsparameter, die in der „*config.json*“-Datei hinterlegt sind, sorgfältig eingelesen.

Ein zentraler Fokus liegt auf der übersichtlichen Organisation der Ressourcen, wobei die *Assets*-Klasse verschiedene Klassenattribute nutzt, um nicht nur auf die Singleton-Instanz, sondern auch auf die Konfigurationsparameter zugreifen zu können. Diese Strukturierung erleichtert nicht nur den Zugriff, sondern ermöglicht auch eine klare Zuordnung und Verwendung der geladenen Ressourcen. Ein anderer nicht unwesentlicher Aspekt dieser Asset-Verwaltung liegt in der Einbindung von Schriftarten (Fonts), die eine visuell ansprechende Gestaltung des Spiels ermöglichen. Durch die Verwendung des Singleton-Entwurfsmusters wird nicht nur die Skalierbarkeit des Spiels verbessert, sondern auch die Pflege und Erweiterung der Ressourcen vereinfacht. Das Singleton-Muster eröffnet somit die Möglichkeit eines globalen, konsistenten Zugriffs auf alle Spielressourcen von verschiedenen Teilen bzw. Komponenten des Spiels aus. Insgesamt bietet die Wahl des Singleton-Entwurfsmusters eine robuste Lösung für die zentrale Verwaltung der für das Spiel benötigten Ressourcen, fördert die Wartbarkeit des Codes und optimiert die Performance des Pygame-Spiels.

4.2 Sonstige

Neben der Verwendung des Singleton-Entwurfsmusters gibt es eine Reihe weiterer bedeutsamer Design- und Implementierungsentscheidungen, die auf dem Weg zum fertigen Endless-Runner-Spiel getroffen wurden. Die Entscheidung für eine objektorientierte Design-Struktur ermöglichte eine klare und modulare Organisation des Codes. Durch die Nutzung von Klassen und Vererbung konnte eine hierarchische Struktur geschaffen werden, die die verschiedenen Spielelemente, wie Spieler, Gegner, Hindernisse und Power-Ups, effizient repräsentiert. Die Vorteile der Objektorientierung, wie Wiederverwendbarkeit, Kapselung und leichtere Erweiterbarkeit, waren entscheidend für die Bewältigung der Komplexität des Endless-Runner-Spiels.

Die Klasse Entity spielt eine zentrale Rolle im Design, da sie die gemeinsamen Eigenschaften und Verhaltensweisen aller Spielelemente definiert. Die Implementierung der *update()*-Funktion sowie von Attributen wie der Position und des aktuellen Status oder angezeigten Bilds in dieser Basisklasse ermöglicht unter anderem eine einheitliche Aktualisierung der Positionen der Spielobjekte und reduziert somit redundanten Code. Die Verwendung von Enums zur Repräsentation von Spielzuständen, Spielerzuständen, Gegnertypen und anderen kategorialen Daten trägt erheblich zur Lesbarkeit des Codes bei. Diese Entscheidung hilft nicht nur, den Code verständlicher zu machen, sondern minimiert auch potenzielle Fehlerquellen durch die Vermeidung von magischen Werten.

Ein weiterer entscheidender Punkt ist die Anpassbarkeit des Spiels. Durch die Integration einer *config.json*-Datei können sämtliche Spielparameter, von der Geschwindigkeit der Spielerbewegung bis zu den Zeitintervallen, in denen Gegner, Hindernisse oder Power Ups spawnen, einfach und schnell angepasst werden. Diese externe Konfigurationsdatei bietet nicht nur Flexibilität während der Entwicklung, sondern ermöglicht es auch, das Spiel nachträglich, ohne Änderungen am Quellcode vornehmen zu müssen, zu adjustieren. Dieser Punkt war unter anderem zur schnellen, nahtlosen Einarbeitung von Feedback von Nutzern entscheidend und trägt maßgeblich dazu bei, das Gleichgewicht des Spiels zu wahren und die Spielerfahrung zu optimieren, ohne in den Code eingreifen zu müssen.

Die Implementierung des Spielzyklus und des Aktualisierungsmechanismus spielt eine bedeutende Rolle für die reibungslose Funktionsweise des Endless-Runners. Die *update()*-Methode wird kontinuierlich aufgerufen und ermöglicht die Aktualisierung aller Spielobjekte, was wiederum die Animationen, Bewegungen und Interaktionen im Spiel sicherstellt. Durch die kluge Nutzung dieses Mechanismus konnte eine effiziente und ressourcenschonende Spiellogik entwickelt werden.

Die Integration eines Systems zur Speicherung und zum Laden von Spielständen (*SaveLoadSystem*) eröffnet dem Spieler die Möglichkeit, das Spiel zu beenden respektive zu schließen und später mit gleichem Spielstand fortzufahren. Das bedeutet, dass Dinge wie die gesammelten Coins, erreichte Statistiken oder gewählte Lautstärkeeinstellungen auch über die aktuelle Spielsession hinaus gespeichert werden. Die Wahl einer einheitlichen Dateiendung und eines festen Ordnerpfads für die

gespeicherten Daten gewährleistet eine konsistente und benutzerfreundliche Implementierung dieses Features.

Die *Menu*-Klasse, die verschiedene Menüs innerhalb des Spiels repräsentiert, wurde so entworfen, dass sie nicht nur die Anzeige, sondern auch die Handhabung von Benutzerinteraktionen ermöglicht. Hierzu zählt neben des Drückens von Schaltflächen auch die Verwendung von sogenannten Slidern, mit denen sich im Einstellungsmenü die Lautstärkeoptionen festlegen lassen. Diese Entscheidung erleichtert nicht nur die Erweiterung um neue Menüs, sondern sorgt auch für eine klare Trennung zwischen Spiellogik und Benutzeroberfläche.

Darüber hinaus wurde die Logik der Kollisionserkennung zwischen den verschiedenen Spielobjekten von einer anfänglichen rechteckigen Kollisionserkennung zu einer pixelgenauen Kollisionserkennung optimiert, was zu einer erheblichen Verbesserung des Spielflusses sowie der Nutzerzufriedenheit beiträgt. Statt der Überprüfung, ob die rechteckigen Bounding-Boxes, welche die Spielentitäten umgeben, überlappen, wird jeder überlappende Pixel dahingehend untersucht, ob er zum jeweiligen Spielobjekt gehört, oder aber transparent ist. Folglich ist das Resultat des Kollisionschecks nur dann positiv, wenn sich zwei Spielentitäten auch wirklich berühren und nicht nur deren Rechtecke. Es wird also sichergestellt, dass es in den überlappenden Bereichen beider Objekte auch wirklich mindestens einen intransparenten Pixel gibt.

Zusammenfassend lassen sich diese Design- und Implementierungsentscheidungen als zentrale Säulen des Endless-Runner-Spiels betrachten. Von der klaren Hierarchie der Klassen über die effektive Nutzung von Enums bis hin zur externen Konfigurationsdatei tragen diese Entscheidungen maßgeblich zur Lesbarkeit, Wartbarkeit und Anpassbarkeit des Codes bei. Sie bilden das Gerüst, auf dem das Spiel aufbaut, und ermöglichen eine flexible und zugleich effiziente Entwicklung.

5 Benutzeranleitung

Zur Ausführung der Anwendung respektive des Endless-Runner-Spiels müssen die folgenden Voraussetzungen erfüllt und die aufgeführten Schritte durchgeführt werden.

Voraussetzungen:

- Python ist in der Version 3.11 oder neuer auf dem System installiert.
- Git ist auf dem Rechner installiert.

Ausführung der Anwendung:

- Klonen des GitHub-Repository (https://github.com/jan-wun/PSE_Endless-Runner-Game).
- Navigation in das geklonte Verzeichnis.
- Starten der Anwendung mithilfe der Batch-Datei „start.bat“ (https://github.com/jan-wun/PSE_Endless-Runner-Game/blob/main/start.bat).