Technische Hochschule Ostwestfalen-Lippe

Fachbereich 2

B.A. Medienproduktion

Bachelorarbeit zum Thema:

Untersuchung der SOLID-Prinzipien: Nutzen und Grenzen in der Spieleentwicklung

Vorgelegt von: Till Sobioch

Matrikelnummer: 15460046

Fachsemester: 7

Abgabedatum: 16.02.2024

Erstgutachter: Prof. Dr. rer. nat. Guido Falkemeier

Zweitgutachter: B. Des. Jennifer Meier

Abstract

Das Thema der vorliegenden Bachelorarbeit ist dem Themenkomplex Softwarearchitektur zuzuordnen. Der Fokus liegt dabei explizit auf der Untersuchung der SOLID-Prinzipien hinsichtlich ihres Nutzens und ihrer Grenzen in der Spieleentwicklung. Um diese Untersuchung vorzunehmen, wurde die Spieleentwicklung als eine komplexe Form der Softwareentwicklung eingeordnet, in der sich Vorgehensmodelle wie Scrum und Anforderungsanalysen bewährt haben. Darüber hinaus wurden für die Untersuchung Anwendungsbeispiele der einzelnen SOLID-Prinzipien herangezogen, die dem eigens entwickelten Spiel entspringen. Die Entwicklung des selbst entwickelten Spiels wurde in der Unity Engine und der objektorientierten Programmiersprache C# durchgeführt. Um die fachliche Grundlage für die Anwendungsbeispiele zu liefern, wurden sowohl Grundprinzipien der objektorientierten Programmierung als auch elementare Funktionsweisen der Spiele-Engine Unity erläutert. Zusätzlich wurden Ziele der Softwarearchitektur dargelegt, um den Nutzen und die Grenzen der SOLID-Prinzipien entsprechend zu analysieren.

Da die Entwicklung von digitalen Spielen eine komplexe Form der Softwareentwicklung ist, braucht es ein geplantes Vorgehen in der Entwicklung und eine angestrebte Softwarearchitektur. Diese Architektur kann durch die Entwurfsmuster der SOLID-Prinzipien erreicht werden. So ergibt sich bei Anwendung des Open-Closed-Prinzips ein Softwaresystem, das für den Fall, dass neue Funktionen oder Inhalte ergänzt werden sollen, erweiterbar bleibt. Dabei bleibt das System durch die Prinzipien Interface-Segregation, Dependency-Inversion und das Liskov'sche-Substitutions-Prinzip ein stabiles Konstrukt, bei dem keine problematischen Abhängigkeiten bestehen. Dadurch ist ein solches System einfach veränderbar, wenn nachträglich Anpassungen vorgenommen werden müssen, und besitzt daher einen geringen Wartungsaufwand. Die Anwendung der SOLID-Prinzipien ist jedoch nicht immer sinnvoll. Bei der Anwendung des Interface-Segragation-Prinzip kann es zu einer hohen Anzahl von Schnittstellen kommen, die zu Unübersichtlichkeit führen kann. Dadurch werden Wartungsaufgaben intensiver, da sich zunächst ein Überblick über das kompliziert gestaltete System gemacht werden muss.

Neben den SOLID-Prinzipien gibt es Entwurfsmuster, wie das Factory-Pattern und das Observer-Pattern, welche in dieser Arbeit nicht behandelt werden. Diese liefern, wie die SOLID-Prinzipien auch, Lösungen für wiederholt auftretende Problemstellungen in der Spieleentwicklung.

Glossar

Begriff	Definition / Erklärung
(Echtzeit- / Spiele-) Engine	Ein Programm, das Werkzeuge zur Entwicklung von Spielen und Anwendungen enthält.
Funktion	Eine innerhalb einer Klasse bestehende Struktur, die einen Wert aus- / zurückgibt.
Klasse	Definiert Eigenschaften, Funktionen, Operationen und Methoden.
Methode	Bildet die konkrete Funktionalität (den Inhalt) von Operationen.
Modul	Eine Einheit eines Programms. Kann eine Funktion, Operation, Methode, Klasse, Objekt, Schnittstelle oder Sammlung von mehreren der genannten sein.
Objekt	Teil eines Programms, das einer Klasse angehört. Wird als Exemplar einer Klasse bezeichnet.
Operation	Stellt spezifische Funktionalität dar. Wird eine Operation aufgerufen, wird diese Funktionalität ausgeführt.
Schnittstelle	Enthält Operationen, stellt aber keine Methoden dazu bereit.

Inhaltsverzeichnis

1 Einleitung

Mit dem gewählten Titel „Untersuchung der SOLID-Prinzipien: Nutzen und Grenzen in der Spieleentwicklung“ ist das Thema dieser Arbeit dem Themenkomplex Softwarearchitektur zuzuordnen. Die SOLID-Prinzipien sind Entwurfsmuster, die in der Softwareentwicklung angewendet werden. Mit ihrer Anwendung wird eine stabile Softwarearchitektur bezweckt, die auf Erweiterbarkeit, einen modularen Aufbau und geringen Wartungsaufwand abzielt. Jedem Buchstaben des Wortes SOLID ist ein Prinzip zuzuordnen: Single-Responsibility-, Open-Closed-, Liskovsches-Substitutions-, Interface-Segregation- und Dependency-Inversion-Prinzip. Diese Prinzipien werden in den frühen 2000er Jahren durch Robert C. Martin unter dem Begriff SOLID gesammelt erläutert, sind jedoch bereits in den 1960er Jahren entstanden. Als Prinzipien der Softwareentwicklung können sie ebenfalls in der Entwicklung von Spielen angewendet werden. Die Spieleentwicklung wird in dieser Arbeit daher als eine Form der Softwareentwicklung behandelt. Es soll geklärt werden, wie hoch der Nutzen der SOLID-Prinzipien in der Spieleentwicklung ist. Das beinhaltet auch, inwieweit die Prinzipien den konkreten Anforderungen der Spieleentwicklung gerecht werden. Darüber hinaus soll untersucht werden, wo die Anwendung der fünf Prinzipien in der Entwicklung von Spielen an mögliche Grenzen stößt.

Diese Arbeit beinhaltet einen theoretischen und einen praktischen Teil. Um in das Themenfeld einzuführen, wird zunächst der Prozess der Softwareentwicklung beschrieben. Dabei wird auf Vorgehensmodelle und verschiedene Anforderungen eingegangen, die an Software gestellt werden. Im Anschluss wird das Paradigma der objektorientierten Programmierung beschrieben. Es werden zusätzlich die drei Grundprinzipien Datenkapselung, Polymorphie und Vererbung erläutert, die Teil der objektorientierten Programmierung sind. Daraufhin wird die Entwicklung von Spielen dargelegt. Dabei liegt der Fokus zunächst auf der *Spiele-Engine* von Unity Technologies und den für diese Arbeit relevanten Aspekten der Unity-Engine. Es folgt eine Auseinandersetzung mit den Elementen von Spielen in der Unity-Engine. Das darauffolgende Kapitel behandelt den Themenkomplex der Softwarearchitektur. Es wird geklärt, was das Ziel einer durchdachten Softwarearchitektur ist und wie es erreicht werden kann. Darüber hinaus wird beschrieben, wie Softwaredesign mit der Softwarearchitektur zusammenhängt und was Entwurfsmuster sind. Im Anschluss daran werden die SOLID-Prinzipien jeweils, sowohl theoretisch eingeordnet als auch anhand von Anwendungsbeispielen veranschaulicht. Daraufhin wird zuerst der Nutzen und anschließend die Grenzen der Prinzipien analysiert. Danach wird der praktische Teil dieser Arbeit weiter

ausgeführt, wobei auf Projektplanung, das Spielkonzept und die Spielmechaniken eingegangen wird. Im Anschluss wird veranschaulicht, wie die Anwendung der SOLID-Prinzipien die Entwicklung der selbst entwickelten Demo-Version des Spiels beeinflusst. Abschließend werden die Ergebnisse dieser Arbeit zusammengefasst und es erfolgt ein Ausblick auf die weitere Untersuchung der SOLID-Prinzipien unter Berücksichtigung der gewonnenen Erkenntnisse dieser Arbeit.

2 Softwareentwicklung

Softwaresysteme sind komplexe Strukturen. Sie instand zu halten kann herausfordernd sein. Besonders, wenn die Software älter ist oder bereits bei ihrer Entwicklung der Überblick verloren geht. So kommt es zu Systemen, die nach ihrer Fertigstellung nicht weiter verändert werden. Grund dafür ist die Sorge, dass die Software nach einer Änderung nicht mehr funktioniert (Lilienthal, 2020). Eine Möglichkeit, diesem Problem entgegenzuwirken besteht darin, die Entwicklung der Software als einen stets gleichbleibenden Prozess zu planen (Brandt-Pook & Kollmeier, 2020).

2.1 Prozess Softwareentwicklung

Der Prozess der Softwareentwicklung kann aktiv gestaltet werden, indem ein Plan entworfen wird, nach dem der Prozess ablaufen soll. Ein solcher Plan stellt ein ideales Vorgehen dar und wird anhand eines Modells beschrieben. Daher bezeichnet man diese Pläne als Vorgehensmodelle. Es gibt für die Softwareentwicklung viele unterschiedliche Vorgehensmodelle. Brandt-Pook und Kollmeier (2020) beschreiben ein einfaches Vorgehensmodell, das sie Basismodell nennen. Nach diesem Basismodell sind alle Prozesse von der Entstehung der Idee bis hin zur in Betrieb genommenen Software ein Teil der Softwareentwicklung. Die Programmierung selbst nimmt dabei als Elemente namens Design[1] und Realisierung nur einen Teil der Entwicklung ein. Das Modell gliedert den Entwicklungsprozess in Phasen und zugehörige Prozesse, welche selbst ebenfalls in Teilprozesse gegliedert sind. Die Ausführung der Prozesse folgt dabei einer vorgegebenen Methodik.[2] So wird die Entwicklung erst detailliert geplant und anschließend ausgeführt. Darüber hinaus sind an der Entwicklung eine Vielzahl von Personen beteiligt, die jeweils bestimmte Rollen einnehmen und spezifische Aufgaben erfüllen. Ein Modell des agilen Projektmanagements, welches große Bekanntheit besitzt und sich stark von klassischen Modellen wie dem Basismodell unterscheidet, ist das Scrum-Modell (Brandt-Pook & Kollmeier, 2020). Das Scrum-Modell liefert nur wenige methodische Vorgaben und stellt dafür einen Rahmen dar, in dem die Entwicklung einer Software abläuft. Außerdem werden, anders als bei dem Basismodell nur wenige Verantwortlichkeiten festgelegt (Drähter, Koschek & Sahling, 2023). Die Vorgaben beschränken sich zum einen auf die Erstellung eines Product-Backlogs[3], der die Funktionalitäten der fertigen Software enthält, die entwickelt werden soll.

[1] Design ist in diesem Fall als Softwaredesign zu verstehen und nicht als visuelles Design.
[2] Detaillierte Erläuterungen zur Methodik können nachgelesen werden in *Softwareentwicklung kompakt und verständlich: Wie Softwaresysteme entstehen* von Brandt-Pook und Kollmeier, 2020.
[3] Product-Backlog (engl.): Produkt-Rückstand.

Das Anlegen des Product-Backlogs geschieht dabei aus der Perspektive von Kunden und Kundinnen. Zum anderen wird unter dem Scrum-Modell vorgegeben, dass die Entwicklung der Software in kleinteilige Aufgabenstellungen aufgeteilt wird. Diese Aufgaben werden dann in sogenannten Sprints, kurzen Arbeitsphasen von ein bis vier Wochen Länge, vom Entwicklungsteam abgearbeitet. So wird die Software mit jedem Sprint weiterentwickelt, während das Team durch regelmäßige Treffen im Austausch bleibt. Durch die Nutzung eines Product-Backlogs orientiert sich die Entwicklung an Softwareanforderungen, also an der gewünschten Funktionalität und den Eigenschaften der Software (Brandt-Pook & Kollmeier, 2020). Es bleibt festzuhalten, dass sich die Art des genutzten Modells je nach Unternehmen unterscheiden kann, die Planung der Softwareentwicklung jedoch unerlässlich ist. Teil dieser Planung ist die Analyse der Anforderungen, welche im Folgenden beschrieben wird.

2.2 Anforderungen an Software

Der Grund, aus dem Software entwickelt wird ist, dass es jemanden gibt der sie braucht oder nutzen will. Damit ergeben sich Erwartungen von Seiten der Nutzer und Nutzerinnen an die Software. Diese Erwartungen gehen oft von dem Anwendungsfall aus und sind somit praktischer Natur. Gleichzeitig versucht das Entwicklungsteam diese Erwartungen zu erfüllen und hat dabei eher eine technisch geprägte Vorstellung von der Software. Es muss also eine Auseinandersetzung zwischen den verschiedenen Parteien stattfinden, die von der Entwicklung der Software betroffen sind. Daraus ergeben sich die ersten funktionalen Anforderungen an die zu entwickelnde Software. Es gibt jedoch weitere Formen von Anforderungen, die eine Software erfüllen soll. Darunter qualitative Anforderungen, welche die Leistungsfähigkeit der Software betreffen. Auch die Gestaltung des vorher beschriebenen Entwicklungsprozesses ist eine Art von Anforderung (Schatten et al., 2010). Darüber hinaus werden Anforderungen an Teile innerhalb der Software gestellt. Bei der Entwicklung von Software entstehen *Module.* Diese sollen bestimmte Aufgaben erledigen. Sie erhalten also die Verantwortung, gewisse Anforderungen zu erfüllen (Lahres, Raýman & Strich, 2018). Durch die Auseinandersetzung des Entwicklungsteams mit den Softwareanforderungen wird zu einem funktionalen und qualitativ hochwertigen Softwaresystem beigetragen.

Es lässt sich also sagen, dass bei der Softwareentwicklung eine Vielzahl von Aspekten berücksichtigt wird, bevor es zur Programmierung der Software kommt. Die Verwendung eines Vorgehensmodells und die Analyse von Anforderungen an die Software sind elementare Bestandteile der Softwareentwicklung. Sie sind erforderlich, um die Komplexität in der Softwareentwicklung zu kontrollieren und ein Scheitern dieser zu vermeiden.

3 Objektorientierte Programmierung

Bevor die grundlegenden Prinzipien objektorientierter Programmierung erläutert werden, ist festzuhalten, dass es verschiedene Paradigmen in der Programmierung gibt, unter denen eine Software entwickelt werden kann. Diese Paradigmen wurden in der Entstehungszeit der Programmierung, Anfang der 1960er Jahre, entwickelt. Es gibt drei übergeordnete Paradigmen: Das der strukturierten Programmierung, der funktionalen Programmierung und der objektorientierten Programmierung.[4] Heute werden weiterhin unter diesen Paradigmen Programme entwickelt und je nach Art von Software gibt es Paradigmen, die besser geeignet sind als andere. Die verschiedenen Paradigmen bringen jeweilige Einschränkungen für die Entwickler und Entwicklerinnen mit sich und bieten so Vorgaben für die zu verwendenden Programmstrukturen (Martin, 2018). Der Inhalt dieser Arbeit unterliegt dem Paradigma der objektorientierten Programmierung, welches im Folgenden genauer beschrieben wird. Dazu werden die drei Grundprinzipien der objektorientierten Programmierung erläutert.

3.1 Datenkapselung

Das erste Grundprinzip der objektorientierten Programmierung ist die Kapselung von Daten. Anders als in der strukturierten Programmierung können Daten in der Objektorientierten Programmierung einem *Objekt* zugeordnet werden. Dies hat zur Folge, dass nicht direkt auf die Daten zugegriffen werden kann. Da nur die jeweiligen Objekte selbst ihre Daten lesen oder ändern können, muss eine *Schnittstelle* definiert werden. Über diese können auch andere Objekte auf diese Daten zugreifen. Der Nutzen der Datenkapselung besteht einerseits darin, dass Daten einen eindeutigen Zustand oder Wert haben. Andererseits können Änderungen am Programm einfacher erfolgen, da die Zahl der davon betroffenen Objekte überschaubar bleibt (Lahres et al., 2018).

3.2 Polymorphie

Die Anwendung des Prinzips der Polymorphie hat zur Folge, „dass verschiedene Objekte bei Aufruf derselben Operation unterschiedliches Verhalten an den Tag legen können." (Lahres et al., 2018, S.196). Das funktioniert, indem zunächst dieselbe Schnittstelle in verschiedenen *Klassen* implementiert wird. Über die Schnittstelle wird dann eine *Operation* aufgerufen. Abhängig von der Klasse, die eine Schnittstelle implementiert, werden von der Operation

[4] Für weiterführende Informationen zu den Paradigmen strukturierter- und funktionaler Programmierung kann nachgeschlagen werden in *Clean Architecture. Das Praxis-Handbuch für professionelles Softwaredesign. Regeln und Paradigmen für effiziente Softwarestrukturierung* von Martin, 2018.

verschiedene *Methoden* aufgerufen. So wird, angestoßen von einer Operation, ein unterschiedliches Verhalten von Objekten ausgelöst (Lahres et al., 2018).

3.3 Vererbung

Das Prinzip der Vererbung lässt sich in zwei Kategorien einordnen. Zum einen die Vererbung von Spezifikation und zum anderen die Vererbung von Implementierung. Diese werden im Folgenden getrennt voneinander erläutert.

Vererbung von Spezifikation

Die Vererbung von Spezifikation ist eng mit dem Prinzip der Polymorphie verbunden. Die Spezifikation einer Klasse beschreibt ihre Eigenschaften, wie zum Beispiel welche Operationen sie besitzt. Bei der Vererbung dieser Spezifikation erhält die Klasse, an die vererbt wird, diese Eigenschaften. Zusätzlich kann die erbende Klasse eigene Eigenschaften besitzen, welche die geerbten erweitern. Das Prinzip kommt zum Einsatz, wenn klar ist, dass verschiedene Objekte grundsätzlich eine oder mehrere Gemeinsamkeiten haben. Dadurch wird definiert, welche Norm Objekte erfüllen. Gleichzeitig können sie sich aufgrund ihrer zusätzlichen Spezifikation unterschiedlich verhalten. Es ist außerdem festzuhalten, dass es bei dieser Form der Vererbung weniger um das Erben von Funktionalität geht, sondern das Erben von Verpflichtungen (Lahres et al., 2018). Das bedeutet, dass die geerbten Eigenschaften auch von der erbenden Klasse genutzt werden müssen.

Vererbung von Implementierung

Die andere Form der Vererbung ist die Vererbung von Implementierung. Dieses Prinzip bezieht sich auf das Erben von Funktionalität. Daraus ergibt sich, dass erbende Klassen alle sichtbaren Methoden, sowie Daten übernehmen. Darüber hinaus können erbende Klassen diese Funktionalität auch überschreiben. Dieses Prinzip wird angewendet, um Wiederholungen zu vermeiden, wenn mehrere Klassen die gleiche Operation oder *Funktion* implementieren würden. (Lahres et al., 2018).

Die Erläuterung dieser Prinzipien stellt die Rahmenbedingungen dar, unter denen in der objektorientierten Programmierung entwickelt wird und soll forthin als Grundlage weiterer Ausführungen zum Thema Programmierung in dieser Arbeit dienen.

4 Spieleentwicklung

Die Entwicklung von digitalen Spielen ist eine Form der Softwareentwicklung und muss unter Befolgung von Vorgehensmodellen, sowie ausgerichtet auf die entsprechenden Anforderungen, ablaufen (siehe Kap. 2.1 f.). Dabei weist die Entwicklung von Spielen eine besonders hohe Vielseitigkeit auf. Von der Grundidee, über die Ausarbeitung der Spiel-Mechaniken, der Regeln und der Balance, der visuellen Ausarbeitung, der Story, der Wahl der Zielplattform, des Marketings und die Auseinandersetzung mit Wünschen der Spieler und Spielerinnen, bis hin zu der Veröffentlichung des Spiels (Schell, 2020). Diese Vielzahl an Aspekten spiegelt sich auch in der Programmierung von Spielen wider. So sind die Programmierer und Programmiererinnen von Spielen mit einer hohen Komplexität im Softwaresystem konfrontiert. Um dieser Komplexität Lösungen entgegenzusetzen, hat sich die objektorientierte Programmierung mit ihren Prinzipien bewährt (Lahres et al., 2018). Neben den Prinzipien, die bei der Programmierung von Spielen verfolgt werden, besteht zudem die Wahl der Spiele-Engine, in der das Spiel entwickelt werden soll. Die Unity Engine ist eine solche Spiele-Engine.

4.1 Unity Engine

Die Echtzeit-Engine von Unity Technologies ist eine der führenden Spiele-Engines. Mit ihr können sowohl Spiele als auch sonstige interaktive Software erstellt werden. Dabei können verschiedene Plattformen bedient werden. Außerdem besteht kostenloser Zugriff auf offizielle Lerninhalte, sowie Zugang zum Forum, in dem auftretende Probleme beim Entwickeln diskutiert werden können (Unity Technologies, 2023). Wie andere Spiele-Engines auch, vereint die Unity Engine eine Vielzahl an Werkzeugen, die zur Spieleentwicklung genutzt werden (Fahme & Khan, 2021). Manche dieser Werkzeuge tragen zur visuellen Ausarbeitung in der Spieleentwicklung bei, wie zum Beispiel das Partikelsystem. Andere sind für die visuelle Präsentation des Spiels verantwortlich, so wie die Grafik-Engine, welche die anzuzeigenden Pixel berechnet. Die Physik-Engine berechnet unter anderem Kollisionsdaten von Spielobjekten, spiegelt also physikalische Gesetze innerhalb des Spiels wider. Außerdem enthält die Unity Engine ein auf künstlicher Intelligenz basiertes Werkzeug zur Berechnung von Wegen. Dieses kommt zum Einsatz, wenn sich ein Spielobjekt selbstständig von einem Punkt zu einem anderen bewegen soll (Fahme & Khan, 2021). Darüber hinaus unterstützen verschiedene Spiele-Engines auch unterschiedliche Programmiersprachen. So wird in Unity mit C# programmiert, während in der Unreal Engine, eine Spiele-Engine eines anderen Unternehmens, mit C++ programmiert wird (ebd.). Die Programmiersprache C# ist eine „von

Microsoft entwickelte Sprache aus der .NET-Familie [und] fasst eine ganze Reihe von Konzepten der Objektorientierung gut zusammen.“ (Lahres et al., 2018, S.24). Die Programmierung in C# ermöglicht folglich, die Prinzipien der objektorientierten Programmierung anzuwenden (siehe Kap. 3).

4.2 Elemente eines Spiels

Wie eingangs erwähnt, umfasst die Entwicklung eines Spiels viele unterschiedliche Aspekte (siehe Kap. 4). Ein Unity-Projekt besteht aus verschiedenen Szenen. Diese Szenen können dabei unterschiedliche Inhalte des Spiels repräsentieren. Darunter das Hauptmenü, verschiedene Level, oder allgemeiner ausgedrückt all die Orte innerhalb eines Spiels, zu denen die Spieler und Spielerinnen manövrieren können. Innerhalb einer Szene findet die Komposition von Spielobjekten statt. Die Spielobjekte bilden dabei den visuellen Inhalt einer Szene. Das Verhalten der Spielobjekte lässt sich über ihre Komponenten steuern. Spielobjekte besitzen dabei verschiedene Komponenten. Die Komponenten sind eindeutig einem Spielobjekt zugeordnet. Unterschiedliche Spielobjekte können dabei zwar die gleichen Komponenten besitzen, teilen diese jedoch nicht. Komponenten besitzen Eigenschaften und liefern Informationen, die teilweise von anderen Komponenten genutzt werden (Borromeo, 2021). Die wichtigsten dieser Komponenten werden im Folgenden beschrieben.

Transform

Die Transform-Komponente enthält Informationen zu Position, Rotation und Größe eines Spielobjekts. Ein Spielobjekt kann nicht ohne eine Transform-Komponente erstellt werden und sie kann nicht von einem Spielobjekt entfernt werden (Unity Technologies, 2024a). Wenn ein Spielobjekt außer der Transform-Komponente keine andere Komponente besitzt, repräsentiert es lediglich einen Punkt in der Szene (Borromeo, 2021).

Mesh Filter und Mesh Renderer

Die Mesh Filter-Komponente referenziert ein Mesh[5]. Das im Mesh Filter referenzierte Mesh wird von der Mesh Renderer-Komponente über die Grafik-Engine (siehe Kap. 4.1) berechnet und so grafisch angezeigt (Unity Technologies, 2024b). Der Mesh Renderer benötigt dazu außerdem die Daten der Transform-Komponente, um das berechnete 3D-Modell an der vom Transform beschriebenen Position anzuzeigen (Borromeo, 2021).

[5] Ein Mesh besteht aus Daten, die eine Form beschreiben (Zusätzliche Informationen unter: https://docs.unity3d.com/Manual/mesh-introduction.html).

Collider

Die Collider-Komponente beschreibt die Form eines Spielobjekts, die zur Berechnung physikalischer Eigenschaften genutzt wird. Auch hier wird auf die Daten der Transform-Komponente zugegriffen, um Kollisionen von Objekten zu berechnen (Borromeo, 2021). Die Form der Collider-Komponente muss dabei nicht der vom Mesh Filter referenzierten Form entsprechen (Unity Technologies, 2024c).

Rigidbody

Die Rigidbody-Komponente ermöglicht die Kontrolle über die Bewegung und Position eines Spielobjekts. Dabei werden diese von der Physik-Engine (siehe Kap. 4.1) berechnet. Dazu können physikalische Kräfte simuliert werden, die auf die Rigidbody-Komponente wirken. Für diese Berechnung werden die Eigenschaften der Komponente herangezogen. Darunter die Masse, Luftwiderstand und Beschleunigung (Unity Technologies, 2024d).

Character Controller

Der Character Controller bietet, wie die Rigidbody-Komponente, die Möglichkeit Bewegung und Position eines Spielobjekts zu kontrollieren. Dabei beruht diese jedoch nicht auf physikalischer Korrektheit, sondern auf dem programmierten Verhalten in einem Skript. So können ebenfalls keine physikalischen Kräfte auf den Character Controller wirken (Unity Technologies, 2024e).

Skript

Ein Skript ist eine Komponente, die ebenfalls zu einem Spielobjekt gehören kann. Jedoch liefert es von sich aus keine Informationen, die das Verhalten des Objekts beeinflussen. Skripte sind Komponenten deren Verhalten programmiert wird. Sie ermöglichen die Implementierung eigener Spielfunktionen. So können Skripte die Eigenschaften anderer Komponenten verändern. Mit dem Erstellen eines Skripts wird gleichzeitig eine neue Klasse erstellt (Unity Technologies, 2024f).

In einer einzigen Szene eines Unity-Projekts können bereits viele Spielobjekte existieren, deren Verhalten von Skripten kontrolliert wird. Da diese Spielobjekte oft in Verbindung stehen oder miteinander interagieren, kann die Beziehung zwischen den Objekten und Klassen zu einem komplexen Softwaresystem führen (siehe Kap. 2). Diese Beziehungen können durch Konzepte der Softwarearchitektur kontrolliert werden.

5 Softwarearchitektur

Die beschriebenen Grundprinzipien der objektorientierten Programmierung (siehe Kap. 3.1 f.) tragen zu einer stabilen Softwarearchitektur bei. Martin beschreibt in seinem Werk die Beweggründe, die für eine Auseinandersetzung mit der Architektur eines Systems sprechen, wie folgt: „Die dahinterliegende Strategie verfolgt das Ziel, so lange wie möglich so viele Optionen wie möglich offenzuhalten.“ (2018, S.154). Eine hohe Priorität hat dabei die Funktionalität eines Systems. Allerdings bezweckt die Softwarearchitektur ebenfalls ein leicht verständliches System, das einfach entwickelt und nachhaltig mit wenig Aufwand instandgehalten werden kann (Martin, 2018).

5.1 Ziele einer guten Softwarearchitektur

Die Qualität von Softwarearchitektur lässt sich nicht generell bemessen. Eine Softwarearchitektur kann nur unter Berücksichtigung konkreter Anforderungen (siehe Kap. 2.2) an das jeweilige Softwaresystem vollständig bewertet werden (Gharbi, Koschel, Rausch, & Starke, 2023). Grundlegend lässt sich jedoch sagen, dass eine gute Softwarearchitektur Änderungen an der Software erleichtert (Nystrom, 2015). Davon lassen sich folgende Ziele einer funktionierenden Softwarearchitektur ableiten.

5.1.1 Modularität

Die Unterteilung eines Systems in Einheiten geschieht bei der Programmierung automatisch. Funktionen und Operationen stellen einzelne Einheiten dar. Klassen können mehrere dieser Einheiten beinhalten und mit anderen Klassen interagieren, um so noch größere Module zu bilden. Entscheidend ist, dass die Zusammenhänge zwischen den Einheiten sinnvoll sind und die Einheiten selbst „in ihrem Inneren ein zusammenhängendes, kohärentes Ganzes bilden […].“ (Lilienthal, 2020, S.75). Die Modularität eines Systems spiegelt sich also auf mehreren Ebenen wider und sie auszuarbeiten, trägt zu einem leichter verständlichen System bei.

5.1.2 Wiederverwendbarkeit

Es gibt Module in Softwaresystemen, die projektunabhängig gestaltet werden können. So kann ein System Module enthalten, die für zukünftige Projekte wiederverwendet werden können. Diese Module zu erkennen ist eine Aufgabe in der Softwarearchitektur. Sie können anschließend so gestaltet werden, dass sie die grundlegende Funktionalität erfüllen und eine Anpassung an den jeweils gegebenen Kontext möglich ist (Dunkel & Holitschke, 2003).

5.1.3 Erweiterbarkeit

Wenn der Aspekt der Erweiterbarkeit eines Systems bei der Entwicklung beachtet wird, können Zusatzfunktionen integriert werden, ohne dass im restlichen System große Änderungen vorgenommen werden müssen (Dunkel & Holitschke, 2003). Die Datenkapselung (siehe Kap. 3.1) trägt zu einer einfachen Erweiterbarkeit bei.

5.1.4 Wartungsaufwand

Da sich Anforderungen an Software ändern können (siehe Kap. 2.2), kann es zu großen, nachträglichen Änderungen an einem System kommen. Die Wartung eines Systems beinhaltet ebenfalls das Entfernen von Fehlern. Ein System muss gut strukturiert sein, damit Änderungen durchgeführt werden können, während das restliche System möglichst unbeeinflusst von diesen Änderungen bleibt (Dunkel & Holitschke, 2003). Jedoch trägt auch die aktive Gestaltung von Modularität zu einem geringen Wartungsaufwand bei, da damit eine Übersichtlichkeit des Systems einhergeht.

Die Softwarearchitektur fungiert als ein übergeordnetes Paradigma, unter dem der Aufbau einer Software strukturiert wird. Softwarearchitektur und Softwaredesign greifen dabei ineinander. So kann eine durchdachte Softwarearchitektur durch ein mangelhaftes Softwaredesign obsolet werden. Umgekehrt kann ein System mit hochwertigem Softwaredesign, aber fehlender Softwarearchitektur, durch diesen Umstand ebenfalls Mängel aufweisen (Martin, 2018). Die Anwendung der SOLID-Prinzipien folgt also dem Ansatz, eine durchdachte Softwarearchitektur herzustellen. Jedes der Prinzipien dient als Lösungsansatz für bestimmte Probleme und stellt einen Baustein für gutes Softwaredesign dar.

5.2 Softwaredesign

Eine konkrete Abgrenzung des Softwaredesigns von der Softwarearchitektur ist nicht möglich. Softwaredesign beschreibt die Details eines Systems. Diese Details werden einerseits durch das Planen der gewünschten Softwarearchitektur bedingt und andererseits bedingen sie selbst die Softwarearchitektur. So lässt sich der Entwurfsprozess wie folgt beschreiben: Zunächst wird festgelegt, aus welchen Klassen und Operationen das System grundsätzlich bestehen soll. Anschließend wird überprüft, wo es beispielsweise Schnittstellen geben muss, um eine Kommunikation zwischen Klassen zu ermöglichen. So wird die gewünschte Softwarearchitektur durch das Softwaredesign gestützt, indem das System auf verschiedenen Ebenen betrachtet wird (Martin, 2018).

5.3 Entwurfsmuster

Die Programmierung objektorientierter Software ist ein Prozess, an dessen Anfang festgelegt wird, welche Objekte benötigt werden und wie sie in abstrakter Form als Klassen entworfen werden. Darüber hinaus wird bestimmt, welche Schnittstellen benötigt werden und wie die Hierarchie der Klassen in Form von Vererbung gestaltet wird (Gamma et al., 2015). Um den damit einhergehenden Ansprüchen an Modularität, Wiederverwendbarkeit, Erweiterbarkeit und geringem Wartungsaufwand gerecht zu werden, kommt es oft zu der Anwendung von Entwurfsmustern. Bei der Entwicklung objektorientierter Software treten häufig dieselben Problemstellungen auf. Entwurfsmuster ermöglichen es, diese Probleme auf eine standardisierte Weise zu lösen. So kann sichergestellt werden, dass Designentscheidungen fundiert getroffen werden und einem übergeordneten Plan folgen (Gamma et al., 2015).

6 SOLID-Prinzipien

Die SOLID-Prinzipien sind Entwurfsmuster, die in der objektorientierten Programmierung maßgeblich die Anordnung von Operationen, Funktionen und den in Klassen vorhandenen Datenstrukturen vorgeben. Außerdem bestimmen sie die Art der Verbindung zwischen Klassen. Jeder Buchstabe von SOLID steht für ein Prinzip (Single-Responsibility-Prinzip, Open-Closed-Prinzip, Liskov'sches Substitutionsprinzip, Interface-Segregation-Prinzip, Dependency-Inversion-Prinzip). Die Entstehung dieser Prinzipien geht ursprünglich auf die 1960er Jahre zurück. Im Jahr 2000 fasst Robert C. Martin die Prinzipien unter dem Begriff SOLID zusammen (Martin, 2018). Im Folgenden werden die fünf SOLID-Prinzipien jeweils erst fachlich eingeordnet und anschließend anhand von Anwendungsbeispielen erläutert.

6.1 Single-Responsibility-Prinzip

Die Anforderungen an Softwaresysteme können sich ändern. Das geschieht, wenn zum Beispiel eine neue Funktionalität ergänzt werden soll. Erfolgt bei der Entwicklung einer Software die Anwendung des Single-Responsibility-Prinzips (SRP), so trägt jedes Modul zwar die Verantwortung für die Erfüllung einer oder mehrerer Aufgaben. Gleichzeitig ist jedoch eine Anforderung nur einem Modul zuzuordnen (siehe Kap. 2.2). Es lässt sich unter Einhaltung dieses Prinzips einfach feststellen, welche Module von einer Änderung betroffen sind (Lahres et al., 2018). So ergibt sich die folgende Beschreibung für das SRP: „Ein Modul sollte nur einen einzigen, klar definierten Grund haben, aus dem es geändert werden muss." (ebd., S.43). Der Grund, aus dem ein Modul angepasst wird, verweist also auf die jeweilige Anforderung, welche sich ändert oder hinzukommt. Eine Möglichkeit, das SRP einzuhalten besteht in der Separierung von Modulen, wenn diese von unterschiedlichen Nutzern und Nutzerinnen verwendet werden (Martin, 2018). Das folgende Anwendungsbeispiel veranschaulicht die Einhaltung des Prinzips innerhalb einer Klasse. Die Anwendung des Single-Responsibility-Prinzips innerhalb der Klasse *PlayerMovement* erfolgt, indem verschiedene Anliegen in jeweilige Operationen aufgeteilt werden (siehe Abb. 1). So erfolgt die Verarbeitung von Eingaben der Spieler und Spielerinnen in der Operation *HandleInput()*.

```
public class PlayerMovement
{
     void HandleInput()
     {
          . . .
     }

     void HandleMovement()
     {
          . . .
     }

     void HandleRotation()
     {
          . . .
     }
}
```

Abb. 1: Anwendungsbeispiel für das Single-Responsibility-Prinzip. Die Klasse PlayerMovement

Das eigentliche Bewegen des Spiel-Charakters erfolgt in der Operation *HandleMovement()*. Die Rotation des Spiel-Charakters erfolgt hingegen in der Operation *HandleRotation()*. So sind die beschriebenen Anforderungen eindeutig den einzelnen Operationen zuzuordnen und das Single-Responsibility-Prinzip wird auf Klassenebene eingehalten. Da Skripte in Unity als Komponenten ebenfalls ein Modul innerhalb des Spiels darstellen (siehe Kap. 4.2), kann auch auf dieser Ebene das Single-Responsibility-Prinzip nachvollzogen werden.

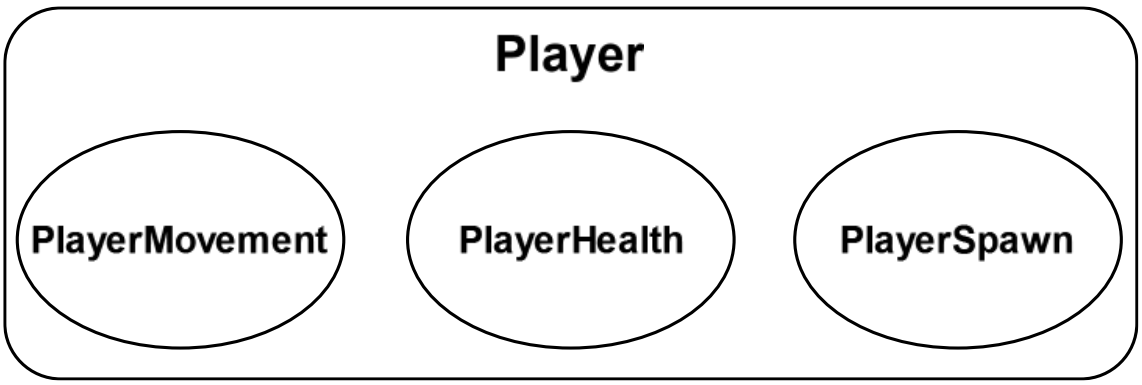

Abb. 2: Das Single-Responsibility-Prinzip unter dem Spielobjekt Player

Das Spielobjekt *Player* besitzt neben dem beschriebenen Modul *PlayerMovement* die Module *PlayerHealth* und *PlayerSpawn* (siehe Abb. 2). Während *PlayerMovement* für die beschriebene Funktionalität verantwortlich ist, erfüllt *PlayerHealth* Anforderungen, die Lebenspunkte betreffen. Das Modul *PlayerSpawn* ist verantwortlich für das Anzeigen des Spielcharakters an der korrekten Position beim Betreten eines Levels. Daraus ergibt sich eine eindeutige Zuordnung der verschiedenen Anforderungen an das Modul *Player* und seine Komponenten.

6.2 Open-Closed-Prinzip

Da es meistens eine oder mehrere Personen gibt, die eine Software nutzen (siehe Kap. 2.2), soll eine gewünschte Erweiterung dieser Software schnell und ohne großen Aufwand erfolgen. Nach dem Open-Closed-Prinzip soll eine solche Erweiterung möglich sein, ohne dass der zu erweiternde Teil einer Software verändert werden muss (Martin, 2018). Das bedeutet, dass Module der Software so entworfen werden, dass sie in der Zukunft für Erweiterungen der Software genutzt werden können. Dabei soll der Quellcode der Module idealerweise unverändert bleiben (Lahres et al., 2018). Daraus ergibt sich das Prinzip: „offen für Erweiterung, geschlossen für Änderung“ (ebd. S. 52). Unter Verwendung von Schnittstellen kann diese Flexibilität erreicht werden. Damit ist keine direkte Modifizierung des Moduls erforderlich. Stattdessen erhält das Modul einen Erweiterungspunkt. Über diesen wird die Operation des Moduls indirekt abgerufen, indem zunächst die Schnittstelle angesprochen wird (Lahres et al., 2018). Das Open-Closed-Prinzip kann darüber hinaus auch auf einer übergeordneten Ebene angewendet werden. Dabei liegt der Fokus auf Komponenten, die aus den einzelnen Klassen und Modulen bestehen. Diese Komponenten der Softwarearchitektur können, gemäß ihrer Abhängigkeiten, hierarchisch angeordnet werden. Eine Anordnung dieser Art veranschaulicht, wo Komponenten auf andere zugreifen und welche von ihnen vor Änderungen geschützt sind (Martin, 2018).

```
public class Golem : InteractionObject, ITalkable
{
     public override void Interact()
     {
          Talk();
     }

     public void Talk()
     {
          //Implementierung Klassenspezifischer Logik
     }
}
```

Abb. 3: Anwendungsbeispiel für das Open-Closed-Prinzip innerhalb einer Klasse

Das Anwendungsbeispiel für das Open-Closed-Prinzip (siehe Abb. 3) bezieht sich auf die Klasse *Golem*. Diese ruft zunächst die Operation *Interact()* auf, durch die anschließend *Talk()* ausgelöst wird. Innerhalb von *Talk()* befindet sich dann der Code, durch den die gewünschte Logik ausgeführt wird. Aufgrund dieses Aufbaus ist die Klasse offen für Erweiterung, da nach Bedarf neue Operationen geschrieben und zu *Interact()* hinzugefügt werden können. Durch diesen Umstand ist die Interact-Operation gleichzeitig geschlossen für Veränderung. Die Einhaltung des Prinzips lässt sich auch auf einer übergeordneten Ebene feststellen (siehe Abb. 4).

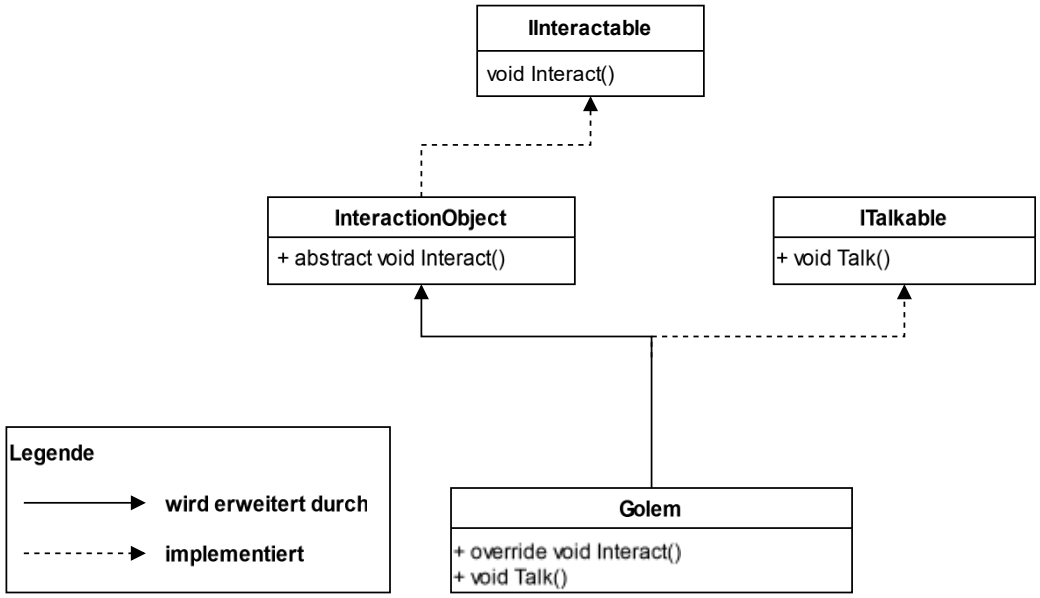

Abb. 4: Das Open-Closed-Prinzip auf einer übergeordneten Ebene

So ist die Klasse *Golem* von Änderungen an der Klasse *InteractionObject* betroffen, nicht aber umgekehrt. Die Klasse *InteractionObject* befindet sich in der Hierarchie der Abhängigkeiten auf einer höheren Ebene und wird vor Änderungen an der Klasse *Golem* geschützt.

6.3 Liskov'sches Substitutionsprinzip

Das Liskov'sche Substitutionsprinzip dient ursprünglich der Steuerung von Vererbung zwischen Klassen. Das bezieht auch Schnittstellen und Implementierungen mit ein. Das Prinzip ist anzuwenden, wenn es Klassen in einem Programm gibt, „[…] die von klar definierten Schnittstellen und der Substituierbarkeit der Implementierungen dieser Interfaces abhängig sind." (Martin, 2018, S.99). Das Prinzip lässt sich in der Vorstellung zweier Klassen erklären, die beispielsweise jeweils einen Wert berechnen und unterschiedliche Eigenschaften haben und somit unterschiedliche Funktionen zur Berechnung des Wertes nutzen. Wenn beide Klassen die Implementierung derselben Schnittstelle erben (siehe Kap. 3.3) und es zum Beispiel eine Anwendung gibt, welche die Funktion der Schnittstelle aufruft, wird das Prinzip eingehalten. Da die Anwendung nicht vom Verhalten der erbenden Klassen abhängig ist und die Klassen ersetzbar, also substituierbar sind, kommt es zu keinem Konflikt (Martin, 2018).

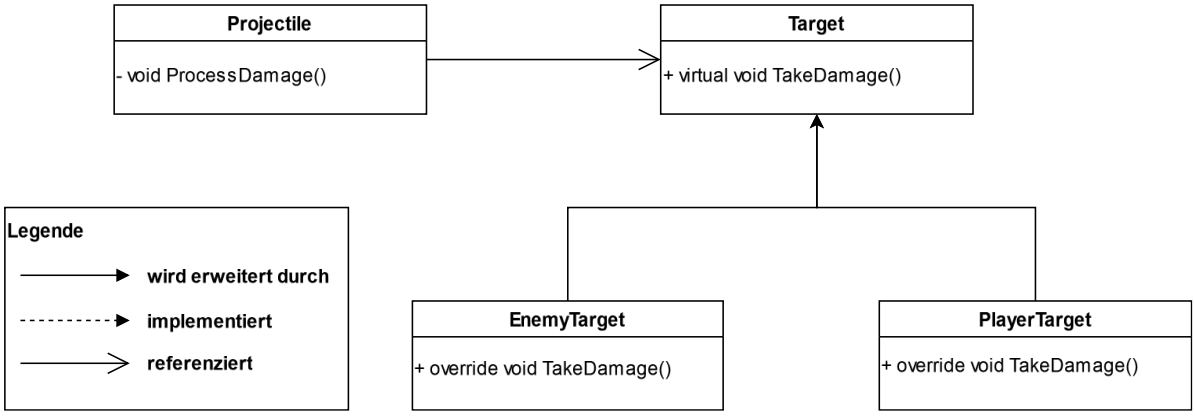

Abb. 5: Anwendungsbeispiel für das Liskov'sche-Substitutions-Prinzip

Das Beispiel für die Anwendung des Liskov'schen-Substitutions-Prinzips (siehe Abb. 5) zeigt die Klasse *Target*, welche die Methode *TakeDamage()* enthält. Die Klasse *Projectile* nutzt diese Operation in der eigenen Operation *ProcessDamage()*. Die Klassen *EnemyTarget* und *PlayerTarget* erben von der Klasse *Target* und implementieren die *TakeDamage()* Operation auf unterschiedliche Weise. Dabei bleibt *Projectile* unabhängig von den erbenden Klassen *EnemyTarget* und *PlayerTarget*. Die erbenden Klassen könnten ausgetauscht werden oder es könnten neue Klassen hinzukommen. Durch den Zugriff von *Projectile* auf die Operation der Klasse *Target*, würde ein solcher Umstand nichts an dem Verhalten von *Projectile* ändern und somit wird das Liskov'sche-Substitutions-Prinzip nicht verletzt.

6.4 Interface-Segregation-Prinzip

Bei der Verwendung von Schnittstellen kann es dazu kommen, dass eine Schnittstelle von mehreren Klassen genutzt wird. Um den verschiedenen Klassen die benötigte Funktionalität bereitzustellen, wird die Schnittstelle um diese erweitert. Dadurch kann es zu dem Problem kommen, dass die verschiedenen Klassen auch Methoden der Schnittstelle implementieren müssen, die sie nicht benötigen, da sie nur für eine der anderen Klassen erforderlich ist. Dann ist die Anwendung des Interface-Segregation-Prinzips nötig. Eine Möglichkeit besteht darin, die Schnittstelle in mehrere Schnittstellen aufzuteilen. So kann jede Klasse eine Schnittstelle ansprechen, die keine ungenutzten Operationen beinhaltet. Eine weitere Möglichkeit ist die Erstellung einer neuen Schnittstelle, die speziell die Anforderungen der einen, sich von den anderen unterscheidenden Klasse, erfüllt (Dirschnabel, 2018). Die Einhaltung dieses Prinzips verhindert, dass Module mehr beinhalten als erforderlich. So wird sichergestellt, dass keine vermeidbaren Abhängigkeiten zwischen Modulen entstehen (Martin, 2018).

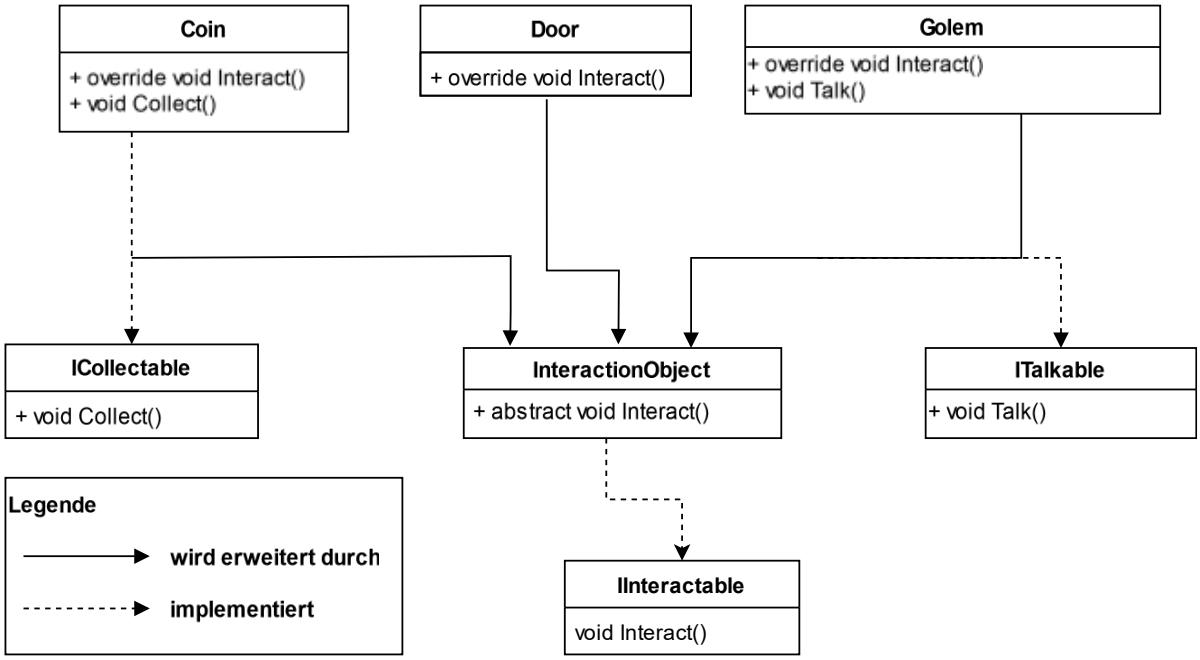

Abb. 6: Anwendungsbeispiel des Interface-Segregation-Prinzips

In dem Anwendungsbeispiel für das Interface-Segregation-Prinzip (siehe Abb. 6) sind die drei Klassen *Coin*, *Door* und *Golem* vorhanden. Außerdem gibt es die Schnittstellen *ICollectable*, *ITalkable* und *IInteractable*. Letztere stellt ihre Operation *Interact()* der Klasse *InteractionObject* zur Verfügung, welche diese als abstrakte Operation implementiert. Die Klassen *Coin*, *Door* und *Golem* erben alle von der Klasse *InteractionObject* und implementieren die Interact-Methode jeweils auf unterschiedliche Weise, indem sie diese überschreiben. Darüber hinaus benötigen die Klassen *Coin* und *Golem* Zusatzfunktionen,

welche nicht in *InteractionObject* enthalten sind, um das Interface-Segregation-Prinzip einzuhalten. So wurden die jeweiligen Schnittstellen *ICollectable* und *ITalkable* mit den benötigten Operationen ausgestattet, welche von *Coin* beziehungsweise *Golem* implementiert werden. Anhand des Negativbeispiels lässt sich die Einhaltung des Interface-Segregation-Prinzips ebenfalls verdeutlichen. Wenn die Operationen *Collect()* und *Talk()* ebenfalls in der Klasse *InteractionObject* enthalten wären, würde es bei der Anpassung einer der Operationen jede Klasse betreffen, die von *InteractionObject* erbt. Dadurch müsste die Anpassung in jeder dieser Klassen vorgenommen werden, auch wenn nicht jede der Klassen die geänderte Funktionalität nutzt.

6.5 Dependency-Inversion-Prinzip

Die Verwendung des Dependency-Inversion-Prinzips unterstützt die Programmierung eines flexiblen Systems. Das wird erreicht, indem innerhalb des Quellcodes keine Abhängigkeiten von konkreten und flüchtigen Modulen hergestellt werden. Dafür kommen abstrakte Schnittstellen zum Einsatz. Daraus folgt, dass Erweiterungen der Funktionalität nur über die Implementierung vorgenommen werden, die abstrakten Module jedoch unverändert bleiben. Dabei stellen die abstrakten Module stabile Konstrukte dar, von denen die flüchtigen und konkretisierten Module abhängig sein können (Martin, 2018). So sind beide Module von der Abstraktion abhängig, welche das benötigte Minimum an Funktionalität enthält. Dadurch wird eine direkte Abhängigkeit zwischen Klassen vermieden (Lahres et. al, 2018).

Sowohl das Anwendungsbeispiel für das Liskov'sche Substitutions-Prinzip (siehe Abb. 5) als auch das für das Interface-Segregation-Prinzip (siehe Abb. 6) beinhalten bereits Beispiele für die Einhaltung des Dependency-Inversion-Prinzips. So bildet die abstrakte Klasse *Target* (siehe Abb. 5) ein stabiles Konstrukt, welches die abstrakte Methode *TakeDamage()* enthält. Die konkrete Funktionalität wird in den jeweiligen Klassen *EnemyTarget* und *PlayerTarget* implementiert, während keine direkten Abhängigkeit zu diesen Implementierungen bestehen, da die Klasse *Projectile* nur auf die abstrakte Operation *TakeDamage()* der Klasse *Target* zugreift. Genauso bestehen mehrere Abhängigkeiten von der abstrakten Klasse *InteractionObject* (siehe Abb. 6). Die konkreten Implementierungen der abstrakten Operation *Interact()* finden sich nur in den erbenden Klassen *Coin*, *Door*, und *Golem* wieder. Von diesen Klassen bestehen keine weiteren Abhängigkeiten, woraus sich die Einhaltung des Dependency-Inversion-Prinzips ergibt.

7 Nutzen und Grenzen der SOLID-Prinzipien

Die SOLID-Prinzipien werden als Entwicklungsmuster angewendet, um die Qualität der Softwarearchitektur eines Systems zu fördern. Daraus ergibt sich ein Nutzen, der in dem folgenden Kapitel dargelegt wird. Darüber hinaus können sich bei der Anwendung der SOLID-Prinzipien Grenzen auftun, an denen die SOLID-Prinzipien die Qualität einer Softwarearchitektur nicht weiter fördern. Diese Grenzen werden im Anschluss an die Auseinandersetzung mit dem Nutzen der SOLID-Prinzipien ebenfalls analysiert. Neben Nutzen und Grenzen besteht zudem Kritik an den Prinzipien. Diese Kritik wird im weiteren Verlauf, nach der Auseinandersetzung mit den Grenzen, dargelegt.

7.1 Nutzen

In diesem Kapitel wird untersucht, wie die SOLID-Prinzipien zu einer guten Softwarearchitektur beitragen und welcher Nutzen sich daraus für die Spieleentwicklung ergibt. Dabei werden zunächst die Ziele der Softwarearchitektur (siehe Kap. 5.1) zur Untersuchung herangezogen und den Auswirkungen der einzelnen SOLID-Prinzipien (siehe Kap. 6) gegenübergestellt. Anschließend werden praktische Beispiele der Auswirkungen in der Spieleentwicklung angerissen.

7.1.1 Modularität

Die Modularität eines Systems wird durch mehrere der SOLID-Prinzipien hergestellt. Die Anwendung des Single-Responsibility-Prinzip (siehe Kap. 6.1) sorgt aus sich heraus für einen modularen Aufbau eines Systems und für die innere Kohärenz der einzelnen Module (siehe Kap. 5.1.1). Der modulare Aufbau wird durch die Anwendung des Open-Closed-Prinzips gestützt, da die einzelnen Module auf mehreren Ebenen abgekapselt werden (siehe Kap. 6.2) und erst so Modularität entsteht. Das Aufgliedern von Schnittstellen nach dem Interface-Segregation-Prinzip (siehe Kap. 6.4) fördert die sinnvolle Verbindung zwischen Einheiten. Darüber hinaus wird dadurch auch das Single-Responsibility-Prinzip und somit die Modularität eines Systems unterstützt, da die Schnittstellen Module darstellen, die eindeutige Verantwortungen haben. Das Liskov'sche-Substitutions-Prinzip und das Dependency-Inversion-Prinzip tragen nicht maßgeblich zur Modularität eines Systems bei.

In der Unity Engine bilden die Spielobjekte mit ihren Komponenten Module. Durch hohe Modularität entsteht eine Übersichtlichkeit über die Vielzahl an Spielobjekten, die in einer Szene vorhanden sein können. Die sinnvolle Verbindung der Spielobjekte unterstützt dies und trägt zu einem leicht verständlichen Aufbau bei. Daraus ergibt sich eine Synergie zwischen der Modularität und dem Vorgehen in der Unity Engine. So können Spielobjekte als

geschlossene Einheiten des Systems in einer Komposition angeordnet werden, die sowohl visueller, als auch funktionaler Natur ist (siehe Kap. 4.2).

7.1.2 Wiederverwendbarkeit

Zu einer Wiederverwendbarkeit von Modulen tragen das Single-Responsibility-Prinzip und das Open-Closed-Prinzip, nur in Verbindung miteinander, bei. Auch wenn die Prinzipien nicht zur Entwicklung vollständig projektunabhängiger Module (siehe Kap. 5.1.2) beitragen, so entstehen bei der Einhaltung der Prinzipien dennoch geschlossene Module, die in andere Systeme integriert werden können. Voraussetzung ist, dass die Anforderungen der Systeme ähnlich sind. Die Wiederverwendbarkeit kann vor Allem durch die Anwendung des Dependency-Inversion-Prinzips (siehe Kap. 6.5) erreicht werden. Aufgrund der Entwicklung stabiler, abstrakter Konstrukte, die Funktionalität bereitstellen aber nicht enthalten, lassen sich Teile der Software in anderen Systemen integrieren und an den gegebenen Kontext anpassen (siehe Kap. 5.1.2). Das Liskov'sche-Substitutions-Prinzip und das Interface-Segregation-Prinzip führen zwar zur Entwicklung von Schnittstellen, bilden aber keine Module, die projektunabhängig wiederverwendet werden können.

Spiele enthalten oft ähnliche Elemente. Wenn bereits ein Interaktions-System (siehe Abb. 6) in einem Spiel existiert, kann es bei Bedarf in ein anderes Spiel integriert und angepasst werden. Ebenso verhält es sich bei einem Teil-System das Angriffsschaden verarbeitet (siehe Abb. 5). Die grundlegende Funktionalität ist bereits vorhanden und die konkret auszuführenden Methoden können in einem anderen Projekt ergänzt werden. Einen hohen Nutzen stellt in der Praxis auch die Wiederverwendbarkeit von Modulen innerhalb des selben Projektes dar. Wenn eine Funktion von verschiedenen Spielobjekten genutzt wird, sollte sie nach dem Liskov'schen-Substitutions-Prinzip (siehe kap. 6.3) nur an einer Stelle programmiert, und für andere Module bereitgestellt werden.

7.1.3 Erweiterbarkeit

Das Prinzip, dessen Anwendung den größten Einfluss auf die Erweiterbarkeit eines Systems hat, ist das Open-Closed-Prinzip. Durch definierte Erweiterungspunkte an Modulen ist das Hinzufügen von zusätzlichen Funktionalitäten möglich. Gleichzeitig bleibt das restliche System unabhängig von der Erweiterung (siehe Kap. 6.2). Das Single-Responsibilty-Prinzip trägt ebenfalls zu einer einfachen Erweiterbarkeit eines Systems bei, da Module, die von einer Zusatzfunktion betroffen wären, leicht erkennbar sind (siehe Kap. 6.1). Das Liskov'sche-Substitutions-Prinzip trägt indirekt zur Erweiterbarkeit bei. Klassen, welche die Implementierung derselben Schnittstelle erben, müssen nach dem Prinzip austauschbar (siehe

Kap. 6.3), und somit auch erweiterbar sein. Eine Aufgliederung von Schnittstellen nach dem Interface-Segregation-Prinzip (siehe Kap. 6.4) hat einen ähnlichen Effekt. Umso mehr Schnittstellen vorhanden sind die nur wenige Operationen bieten, desto leichter ist es, neue Funktionalitäten einer dieser Schnittstellen zuzuordnen. Das Dependency-Inversion-Prinzip kontrolliert Abhängigkeiten (siehe Kap. 6.5), was nicht direkt zu einer leichteren Erweiterbarkeit führt, damit jedoch verhindert, dass Elemente der Software bei Erweiterungen verändert werden müssen, obwohl sie nicht von der Erweiterung betroffen sind.

Spiele werden zumeist nicht nur mit neuen Spielfunktionen erweitert, sondern erhalten oft komplett neue Spielinhalte. Wenn diese entwickelt werden, bilden sie selbst ein komplexes System (siehe Kap. 4). Dieses in das bestehende Spiel zu integrieren ist eine Herausforderung. Wenn ein Spiel leicht zu erweitern ist und keine großen Veränderungen am bestehenden System vorgenommen werden müssen, kann diese Herausforderung bewältigt werden. Hier sind hohe Modularität (siehe Kap. 7.1.1) und Wiederverwendbarkeit (siehe Kap. 7.1.2) hilfreich, da Spielobjekte als Module verwendet und an die neuen Inhalte angepasst werden können. So müssen Funktionalitäten nicht erneut programmiert werden. Gleichzeitig können Module, die von der Anwendung der SOLID-Prinzipien geprägt sind, selbst einfach mit Zusatzfunktionen ausgestattet werden. So können Varianten der bestehenden Module entwickelt werden.

7.1.4 Wartungsaufwand

Durch die klare Zuordnung von Anforderungen an Module nach dem Single-Responsibility-Prinzip (siehe Kap. 6.1) kann bei Änderungen am System direkt festgestellt werden, wo sich die betroffenen Module befinden. Es fördert dabei die Übersichtlichkeit eines Systems (siehe Kap. 5.1.4). Weiter können nach dem Open-Closed-Prinzip Änderungen an einem System vollzogen werden, ohne dass unbeteiligte Module von den Änderungen beeinflusst werden (siehe Kap. 6.2). Das Liskov'sche-Substitutions-Prinzip schafft Übersichtlichkeit über die Beziehung von Modulen und trägt zu einer geringeren Fehleranfälligkeit durch Abhängigkeiten bei (siehe Kap. 6.3). So auch das Dependency-Inversion-Prinzip, durch das Abhängigkeiten zu Teilen der Software, die am meisten von Änderungen betroffen sein könnten, vermieden werden (siehe Kap. 6.5). Durch die Anwendung des Interface-Segregation-Prinzips wird sichergestellt, dass nur die von einer Änderung direkt betroffenen Klassen auch berücksichtigt werden müssen (siehe Kap. 6.4).

Bevor ein Spiel veröffentlicht wird, wird es wie andere Software auch, im Idealfall auf Fehler überprüft. Dennoch können den Nutzern und Nutzerinnen nach der Veröffentlichung Fehler

auffallen. Diese sollten vom Entwicklungsteam möglichst schnell behoben werden. Das wird ermöglicht, indem die Fehlerquelle ohne großen Aufwand gefunden werden kann (s.o.). Durch die Strukturierung des Spiels mit Hilfe der SOLID-Prinzipien kann der Fehler identifiziert und behoben werden. Dabei entsteht kein erneuter Arbeitsaufwand, aufgrund von Abhängigkeiten zu anderen Modulen als dem betroffenen Modul. So können auch generelle Änderungen, beispielsweise um die Schwierigkeit im Spiel anzupassen, ohne Probleme durchgeführt werden. Der Wartungsaufwand wird so geringgehalten.

7.2 Grenzen

Obgleich ein hoher Nutzen durch die Anwendung der SOLID-Prinzipien für die Spieleentwicklung entsteht, so gibt es auch Anwendungsfälle, an denen die Prinzipien an Grenzen stoßen. An welchen Stellen diese Grenzen liegen und wie bedeutend diese sind, wird im Folgenden untersucht.

7.2.1 Disziplin und Ordnung

Die Anwendung von Entwurfsmustern wie den SOLID-Prinzipien erfordert Disziplin, da ständig reflektiert werden muss, wie Änderungen oder Erweiterungen in das Programm integriert werden können. Dabei muss die geschaffene Ordnung des Codes beibehalten werden (Nystrom, 2015). Dies geht mit einem erhöhten Zeitaufwand in der Entwicklung einher. Darüber hinaus kann eine Nicht-Beachtung der vorliegenden Struktur später zu hohem Arbeitsaufwand führen, wenn eine Anforderung hinzukommt, dessen Erfüllung mit großen Änderungen an der Software verbunden ist (Lilienthal, 2020).

7.2.2 Over Engineering

Bei der Anwendung der SOLID-Prinzipien werden Entscheidungen getroffen. Beispielsweise darüber, wann eine Abstrahierung vorgenommen wird, um in der Zukunft bestimmte Erweiterungen der Software zu ermöglichen. Dies geht allerdings mit steigender Komplexität in der Spiele-Software einher. Zu viele solcher Erweiterungspunkte wie Schnittstellen, „[…] Plug-in-Systeme und abstrakte Basisklassen, virtuelle Methoden […]“ (Nystrom, 2015, S.31) und ähnliche Module führen zu einem unübersichtlichen System und Over Engineering[6]. Eine zu häufige Aufgliederung von Schnittstellen nach dem Interface-Segregation-Prinzip (siehe Kap. 6.4) kann einen solchen Effekt haben. Ebenso der Einsatz von abstrakten Schnittstellen oder Klassen nach dem Dependency-Inversion-Prinzip (siehe Kap. 6.5). Wodurch die Suche nach der eigentlichen Funktionalität bei Änderungen erschwert werden kann (Nystrom, 2015).

[6] Over Engineering bedeutet: „[…] das Produkt umfasst weit mehr Funktionen als der Nutzer […] wirklich benötigt.“ (Broy, Geisberger, Kazmeier, Rudorfer & Beetz, 2007, S. 129).

So schreibt Lilienthal, dass ein leicht verständliches System wünschenswert ist und stets die weniger komplexe Vorgehensweise beim Entwerfen der Softwarearchitektur verfolgt werden sollte. Es ist jedoch ein Abwägungsprozess, dessen Ergebnis mit Erfahrung in der Entwicklung zusammenhängt (2020).

Ein guter Überblick ist für die Spieleentwicklung aufgrund der Komplexität von hoher Bedeutung (siehe Kap. 4). Da Spiele lange auf dem Markt bleiben können, muss ihr Quellcode auch nach langer Zeit für Entwickler und Entwicklerinnen verständlich und wartbar bleiben. Außerdem werden Spiele in der Industrie von mehreren Personen entwickelt, was eine Verständlichkeit des Systems ebenfalls bedeutend macht.

7.3 Kritik an den SOLID-Prinzipien

Die SOLID-Prinzipien sehen sich angesichts ihrer langen Geschichte (siehe Kap. 6) mit Kritik konfrontiert. Ein großer Kritiker der Prinzipien ist Daniel Terhorst-North[7]. Im Interview mit Eberhard Wolff von Heise Online schildert North, wie er sowohl sehr guten Code gesehen hat, der jedes der fünf SOLID-Prinzipien verletzt hat als auch sehr schlechten Code, der alle fünf Prinzipien eingehalten hat (Heise Online, 2022). Es folgt eine Zusammenfassung der im Interview geäußerten Kritik zu den einzelnen Prinzipien.

Die Anwendung des Single-Responsibility-Prinzips kann eine zu frühe Unterteilung von Modulen erwirken, wenn diese Module bei Änderungen alle betroffen sind. Eine Unterteilung von Modulen gemäß den Anforderungen sollte nur vorgenommen werden, wenn es nötig ist, jedoch nicht aus Prinzip. North führt darüber hinaus an, dass es eine Vielzahl nicht vorhersehbarer Gründe geben kann, aus denen ein Modul in der Zukunft geändert wird. Das Open-Closed-Prinzip (siehe Kap. 6.2) verfolgt laut North einen falschen Ansatz, wenn es um Änderungen im Code geht. Anstatt Funktionsweisen durch Erweiterungen zu ändern, sollte der Code selbst geändert werden, die Module also nicht für Modifikation geschlossen sein. Durch die Verwendung von Versionierungs-Software könne dies weitgehend ohne Risiko passieren. Am Liskov'schen-Substitutions-Prinzip (siehe Kap. 6.3) kritisiert North, dass die Verbindungen zwischen Klassen und Schnittstellen heute weitaus komplexer sind. Dabei habe die Objektkomposition[8] die Vererbung in großen Teilen der objektorientierten

[7] North ist dabei nicht der einzige Kritiker der SOLID-Prinzipien. In dem Interview verweist er auf einen Vortrag von Kevlin Henney bei der NDC Conference im Jahr 2016 mit dem Titel *SOLID D*econstruction. Verfügbar unter (letzter Zugriff 05.02.2024): https://vimeo.com/157708450

[8] Weiterführende Informationen zur Objektkomposition in: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software.* Von Gamma, E., Helms, R., Johnson, R. & Vlissides, J. (2004). Boston: Addison Wesley Verlag.

Programmierung abgelöst. Das Interface-Segregation-Prinzip (siehe Kap. 6.4) ist laut North eine Strategie, eine Situation, die beim Entwickeln eines Systems auftreten kann, zu bereinigen. Es ist jedoch kein Entwurfsmuster, da es die Entstehung dieser Situation nicht verhindert. An dem Dependency-Inversion-Prinzip (siehe Kap. 6.5) kritisiert North vor Allem die Vorstellung von höheren und niedrigeren Ebenen von Code und dessen Anordnung in einer Hierarchie. Darüber hinaus geht er darauf ein, dass ein gutes Prinzip nicht zu viel Anwendung finden kann, wie es bei manchen der SOLID-Prinzipien der Fall sein kann. Generell kritisiert er, dass die Prinzipien auf einer veralteten Mentalität, basierend auf der objektorientierten Programmierung der 1990er Jahre, beruhen. So bezeichnet er die Prinzipien als ein historisch wertvolles Dokument. Die Prinzipien weisen laut North aufgrund von Entwicklungen in der Softwareprogrammierung einen Mangel an Aktualität auf und sollen Probleme lösen, die heute durch anderes Vorgehen in der Entwicklung nicht mehr zustande kommen.

8 Praktische Umsetzung: Ein (Bei-)Spiel

Im Rahmen dieser Bachelorarbeit ist ein Spiel zu Demonstrationszwecken entstanden, bei welchem angestrebt wurde, die SOLID-Prinzipien einzuhalten. Aus diesem Spiel entspringen die Anwendungsbeispiele zu den einzelnen SOLID-Prinzipien (siehe Kap. 6). Der Entwicklungsprozess folgte dabei Methoden des agilen Projektmanagements (siehe Kap. 2.1). Das Spiel wurde unter Verwendung der Programmiersprache C# in der Unity-Engine entwickelt (siehe Kap. 4).

8.1 Projektplanung

Um den Entwicklungsprozess des Spiels zu koordinieren, wurde auf Methoden des agilen Projektmanagements zurückgegriffen (siehe Kap. 2.1). Hierbei ist zu beachten, dass die Entwicklung von einer Einzelperson durchgeführt wurde und daher Elemente des agilen Projektmanagements, die einen Austausch im Team erfordern, nicht realisiert werden konnten. Um Projektplan, Konzept, Product-Backlog, Sprints und den Fortschritt in der Entwicklung des Spiels zu organisieren, wurde das Projektmanagement-Tool *Trello* genutzt.

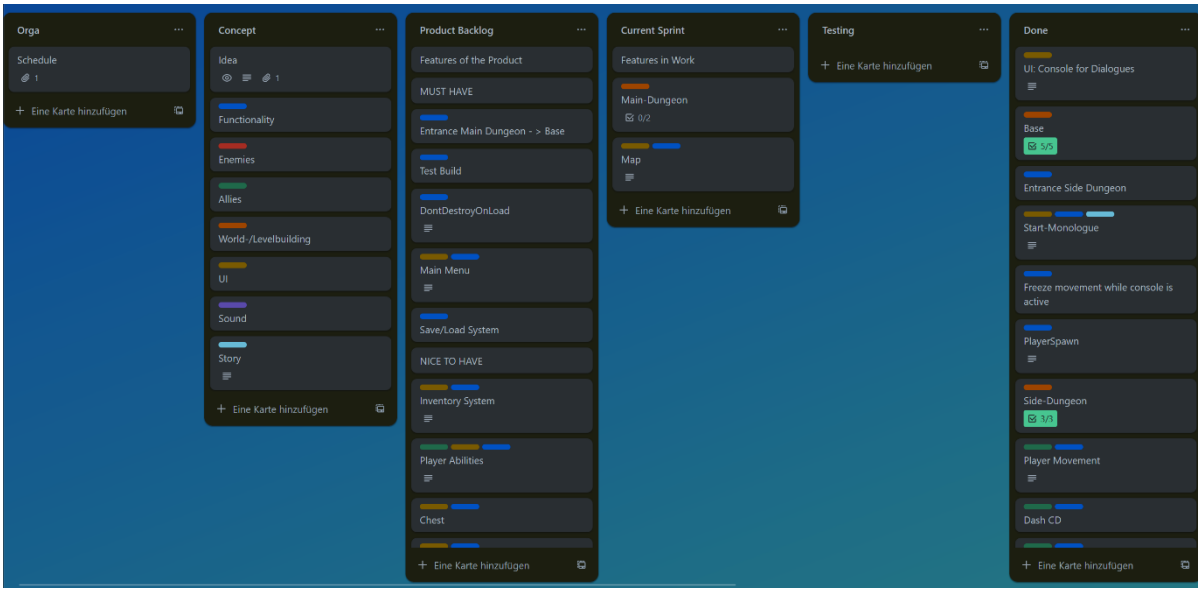

Abb. 7: Ein sog. Trello-Board mit Listen, die verschiedene Karten enthalten.

Das *Trello-Board* wurde dabei mit Listen zu Projektorganisation, Konzept, dem Product-Backlog, dem aktuellen Sprint, zu Spielelementen, die getestet werden und zu erledigten Aufgaben, gefüllt (siehe Abb. 7). Den Listen wurden dabei Karten zugewiesen. Mit Hilfe eines beliebigen Farbschemas, das in der Liste zum Konzept festgelegt wurde, konnten sich die Karten den verschiedenen Spielinhalten zuordnen lassen. So wurden beispielsweise alle Karten, die mit der Farbe Blau markiert sind, der Implementierung von Funktionalität zugeordnet. Die Karten konnten dabei von einer Liste in eine andere bewegt werden.

Außerdem wurden sie mit Beschreibungen und Anhängen versehen, wenn nötig (siehe Abb. 8).

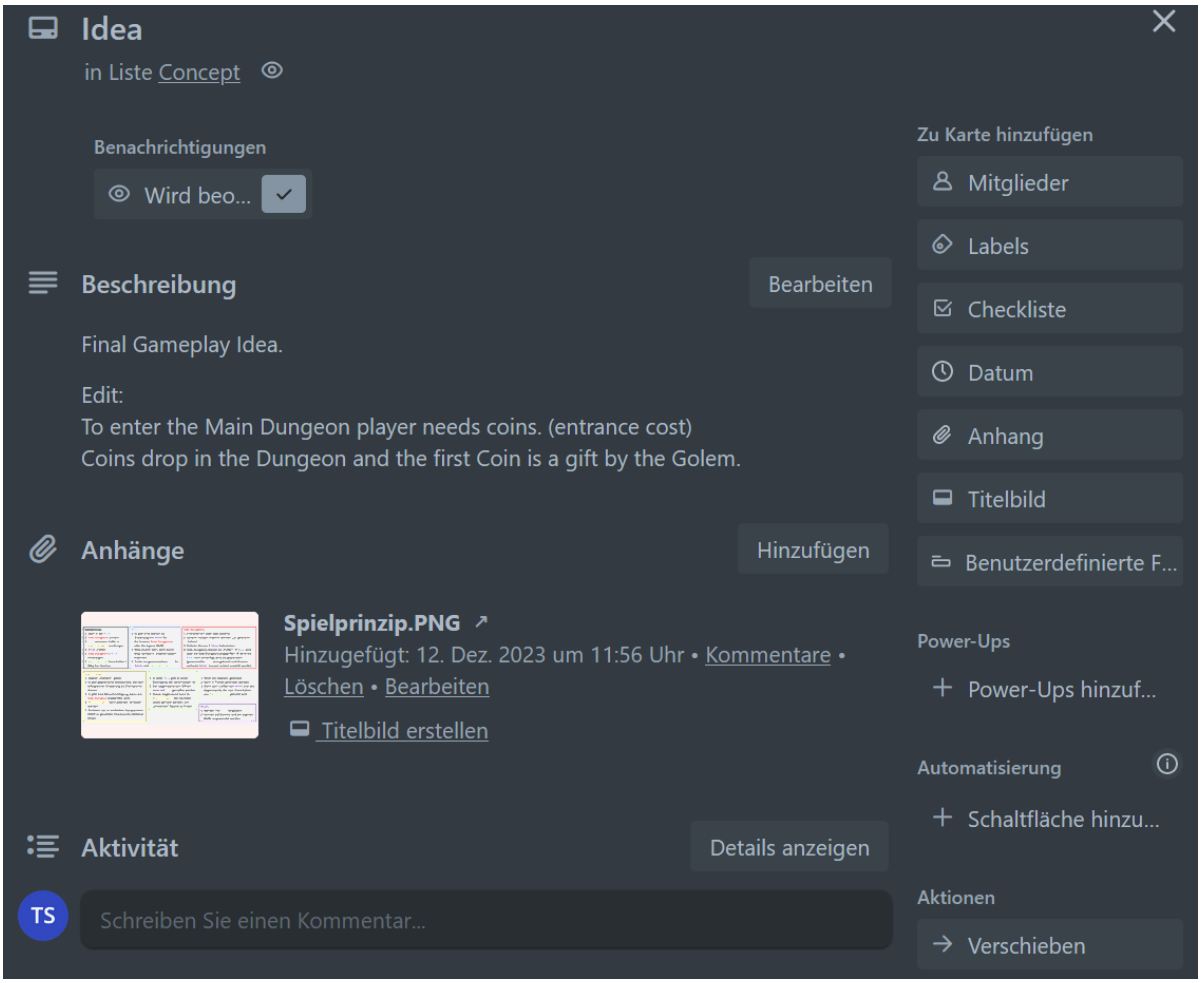

Abb. 8: Aufgeklappte Karte zur Spielidee mit einer Beschreibung und einem angehangenen Bild

So wurde ein Überblick über die unterschiedlichen Aufgaben im Product-Backlog und im aktuellen Sprint geschaffen und bei Bearbeitung einer bestimmten Aufgabe auf zusätzliche Informationen zurückgegriffen.

8.1.1 Zeitplanung

Zu der Projektplanung in der Spieleentwicklung gehört eine realistische Zeitplanung. Die Bachelorarbeit und das Praxisprojekt wurden im zeitlichen Rahmen von zehn Wochen bearbeitet. Unter Berücksichtigung dieser Vorgabe wurde zu Beginn des Projekts ein Zeitplan angelegt (siehe Abb. 9). Der Zeitplan sah vor, in den ersten drei Wochen die Konzeptionierung abzuschließen. Dies beinhaltete Brainstorming, die Ausarbeitung der Spiel-Idee sowie die Entwicklung von Prototypen.

Dezember	Januar	Februar
8. Brainstorming, Entwicklung der Spiel-Idee	5. Sprint 2: Programmierung grundlegender Mechaniken	2. Testspielen
15. Ausarbeitung der Spiel-Idee & Vorbereitung des Product-Backlog	12. -	9. Fehlerbehebung, Hot-Fixes
22. Prototyping	19. Sprint 3: Programmierung zusätzlicher Funktionen	16. Projektabschluss
29. Sprint 1: Worldbuilding	26. -	

KONZEPTIONIERUNG UMSETZUNG

Abb. 9: Zeitplan für die Entwicklung des Spiels im Rahmen der Bachelorarbeit

Darauf folgten Sprints, in denen die Spielumgebung in der Unity-Engine gebaut wurde und die Spielfunktionen programmiert wurden. In den letzten zwei Wochen waren Spiele-Tests und Fehlerbehebungen vorgesehen.

8.1.2 Ideenfindung

Bevor die tatsächliche Entwicklung eines Spiels losgeht, muss zunächst eine Idee ausgearbeitet werden. Zur Ideenfindung wurde ein Brainstorming durchgeführt (siehe Abb. 10).

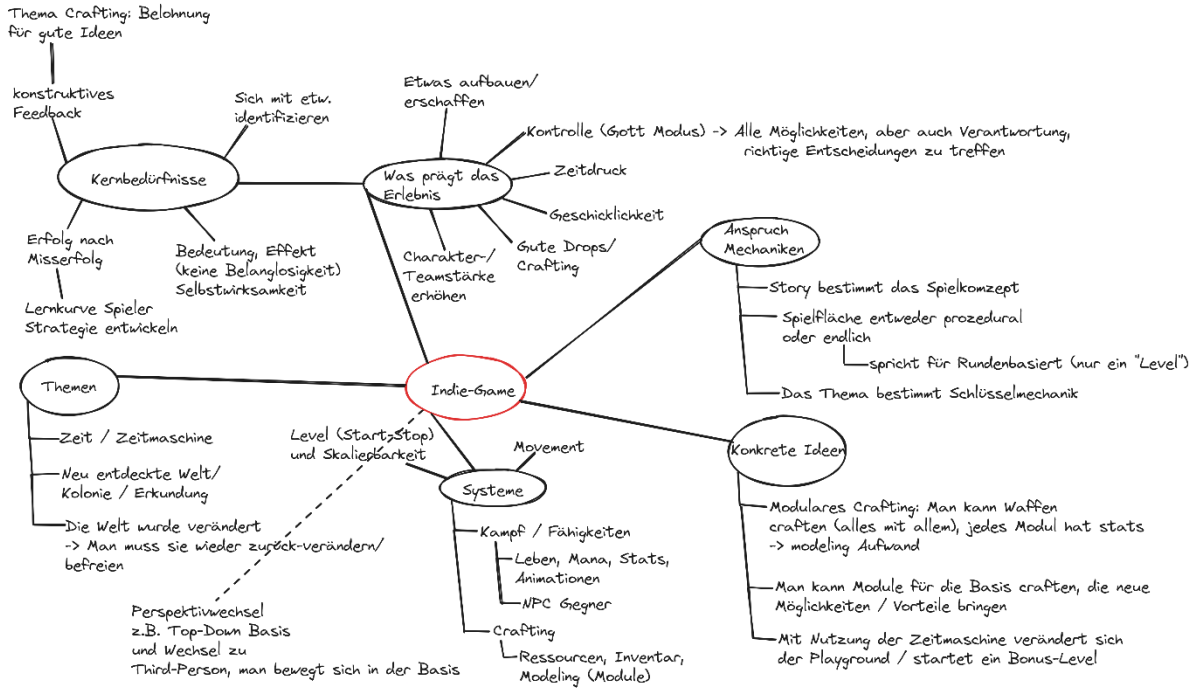

Abb. 10: Ergebnis des Brainstormings zur Entwicklung einer Spielidee

Das Ergebnis des Brainstormings schuf die Grundlage für die Entwicklung konkreter Ideen zum möglichen Inhalt der Story, zum Spielprinzip und zu grundlegenden Spielmechaniken (siehe Abb. 11). Anschließend wurden die konkreten Ideen weiterentwickelt, wobei einige der Ideen verworfen, manche verändert und andere neu hinzugefügt wurden.

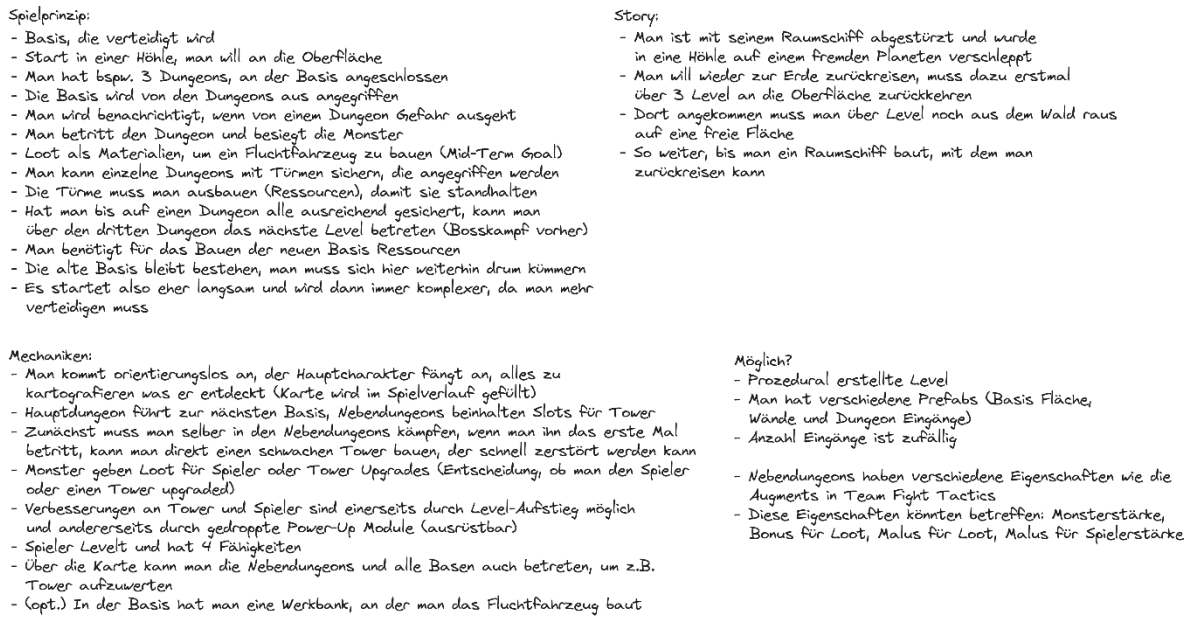

Abb. 11: Liste mit konkreten Ideen für das Spielprinzip, die Mechaniken und die Story

Durch die Überarbeitung dieser Ideen entstand das vollständig ausgearbeitete Spielkonzept.

8.1.3 Entwicklung von Prototypen

Zur Vorbereitung der Entwicklung des Spiels wurden Prototypen entwickelt. Prototypen sind hilfreich, um geplante Spielmechaniken in Hinblick auf die gewünschte Funktionsweise zu testen. So konnte zu Beginn des Projekts festgestellt werden, an welchen Punkten Fehleranfälligkeiten bestehen. Darüber hinaus wurden dabei die ersten Teile des Spiels entwickelt. Es wurde ein Prototyp zur Steuerung des Spielcharakters erstellt. Dieser Prototyp beinhaltete außerdem das Erscheinen eines Gegners, der dem Spielcharakter folgt und eine Angriffs-Animation abspielt, wenn er in Reichweite ist (siehe Abb. 12).

Abb. 12: Prototyp für die Steuerung des Spielcharakters

Neben diesem Prototyp wurde ein weiterer entwickelt, der Funktionsweisen des ersten Prototyps nutzte, diese aber auf ein anderes Szenario anwendete. So erschienen mehrere Gegner am Rand der Spielfläche und bewegten sich zu dem Turm in der Mitte des Spielfelds (siehe Abb. 13).

Abb. 13: Prototyp für den automatischen Angriff von Einheiten durch einen Turm

Bei Erreichen des Turms blieben die Gegner stehen und es wurde eine Angriffs-Animation abgespielt. Während der gesamten Zeit bewegte sich, in kleinen zeitlichen Abständen, ein Projektil von der Turmspitze aus in die Richtung eines Gegners und verschwand, wenn es ihn erreichte. Dieser Prototyp half dabei, bestehende Spielmechaniken auf ihre Skalierbarkeit und Anpassungsfähigkeit zu überprüfen.

8.2 Spielkonzept

Da der Fokus dieser Arbeit auf der Anwendung der SOLID-Prinzipien liegt, wurde ein Spiel im *RPG* (Role-Playing-Game, engl.: Rollenspiel) Genre entwickelt. Aufgrund der programmiertechnischen Vielseitigkeit in diesem Genre konnten alle fünf SOLID-Prinzipien bei der Entwicklung angewendet werden. Damit ging einher, dass das Spiel nur zu Demonstrationszwecken entwickelt wurde und nicht alle Elemente des Spielkonzepts umgesetzt wurden.

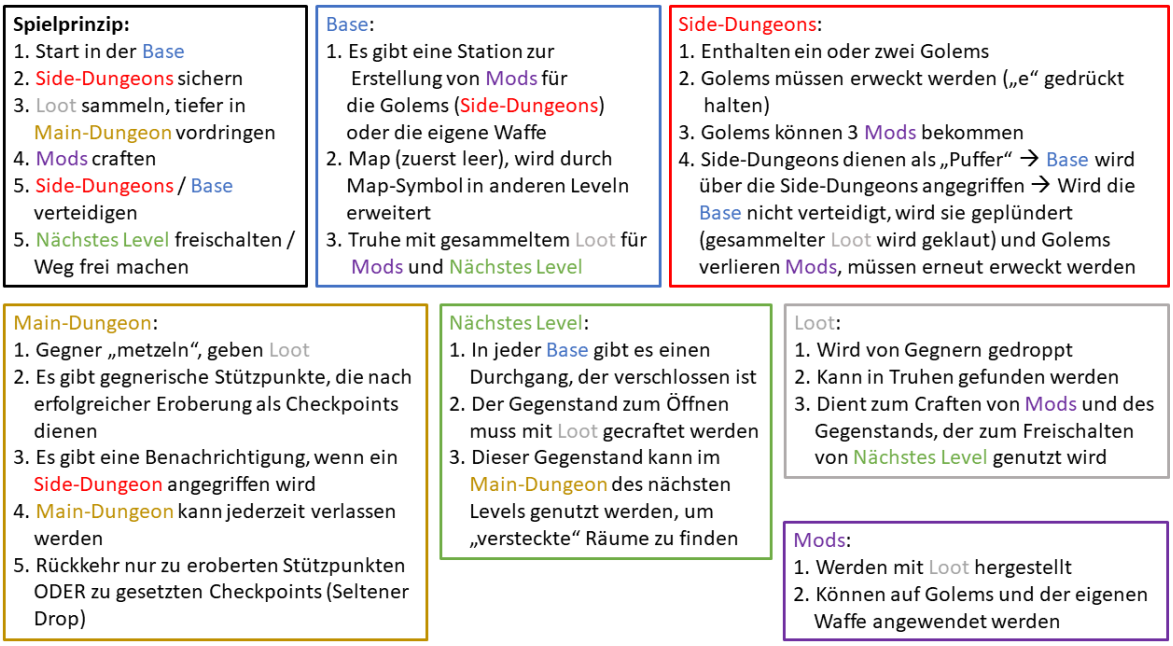

Abb. 14: Vollständige Auflistung der verschiedenen Aspekte des Spiels.

Die Überarbeitung der ersten Spielideen führte zu einem ausgearbeiteten Spielkonzept (siehe Abb. 14). Die Übersicht beinhaltete die übergeordneten Elemente des Spiels und führte sie aus. Anhand dieser Ausführungen konnten alle für das fertige Spiel benötigten Level und Teilelemente abgeleitet werden. So sollte das Spiel in der *Base* starten, von der die Spieler und Spielerinnen zunächst nur Zugang zum *Side-Dungeon* haben. Wenn dieser durch die Aktivierung eines Golems gesichert wurde, sollten sie von der Base aus auch den *Main-Dungeon* betreten können. Dort sollte das Besiegen von Gegnern im Vordergrund stehen, um *Loot* (fallengelassene Gegenstände) einzusammeln und gegnerische Stützpunkte einzunehmen, um den Fortschritt im Main-Dungeon zu sichern. Mit dem Loot sollten einerseits *Mods* zur Verstärkung der eigenen Waffe und des Golems hergestellt werden können. Andererseits sollte der Zugang von der Base zum *nächsten Level* versperrt sein und nur mit einem bestimmten Gegenstand geöffnet werden können. Dieser Gegenstand sollte ebenfalls mit Loot hergestellt werden können. Darüber hinaus sollte der Gegenstand nach

Erreichen des nächsten Levels erhalten bleiben und zur Einführung einer neuen Spielmechanik im nächsten Level genutzt werden.

8.3 Spielmechaniken

Aus dem Spielprinzip gingen verschiedene Spielmechaniken hervor, die implementiert werden mussten. So wurde ein Bewegungssystem für den Spielcharakter entwickelt, wie auch ein Kampfsystem, welches das Auslösen von Angriffen und Verursachen von Schaden ermöglichen sollte (siehe Abb. 15).

Abb. 15: Spielcharakter schießt einen Feuerball auf Gegner

Darüber hinaus wurde ein System benötigt, das Interaktionen mit Spielobjekten in der Umgebung ermöglicht. Dieses sollte sowohl zum Betreten der Dungeons als auch für die Aktivierung des Golems oder zum Einsammeln von Loot genutzt werden können (siehe Abb. 16).

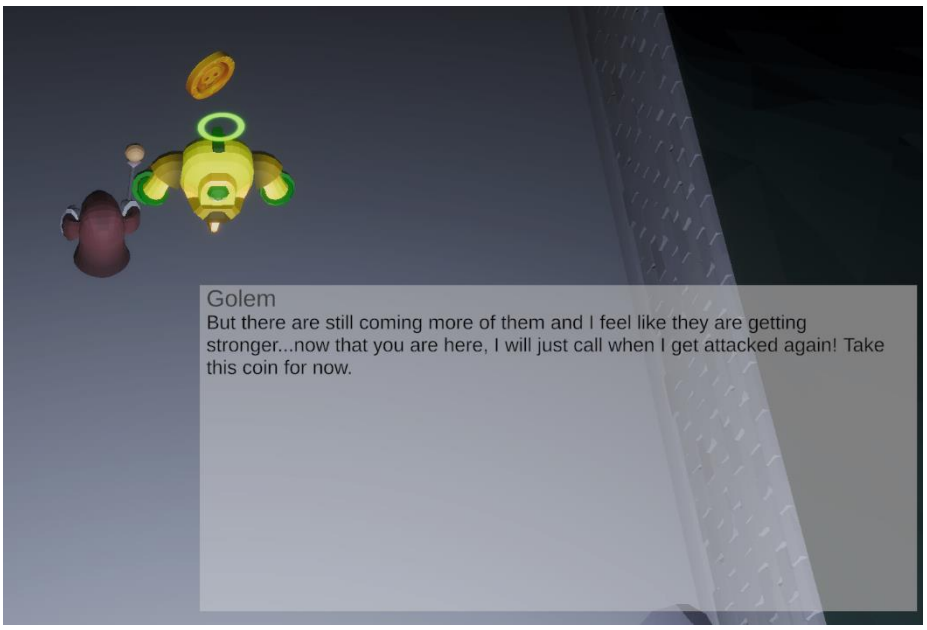

Abb. 16: Interaktion des Spielcharakters mit einem Golem

Es ergaben sich aus dem Spielprinzip weitere Mechaniken, die im Rahmen des Projektes nicht, oder nur teilweise, implementiert wurden. Darunter die Mechanik zur Herstellung von Mods und anderen Gegenständen. Außerdem die Implementierung eines Inventars, in dem gesammelte Gegenstände abgelegt werden können, um sie zu einem späteren Zeitpunkt im Spiel zu nutzen. Bei der Implementierung der vorhandenen Mechaniken wurde auf die Einhaltung der SOLID-Prinzipien Wert gelegt. Im Folgenden wird veranschaulicht, wie sich dies auf das Spiel auswirkte und das Vorgehen in der weiteren Entwicklung beeinflussen kann.

8.4 Weitere Entwicklung unter den SOLID-Prinzipien

Die Anwendung der SOLID-Prinzipien erfolgte wie in den Beispielen beschrieben (siehe Kap. 6). Daraus ergaben sich Möglichkeiten, die die weitere Entwicklung des Spiels maßgeblich beeinflussen (siehe Kap. 7.1). Diese lassen sich konkret anhand von zwei Beispielen veranschaulichen.

Beispiel Nr. 1: Hinzufügen und modifizieren von Gegnern

Der durch die Anwendung der SOLID-Prinzipien geschaffene modulare Aufbau des Spiels ermöglicht das schnelle Designen neuer Gegner. Ein fertiger Gegner besitzt auf der Ebene der Spiel-Logik, neben der Transform-Komponente, eine Collider- und Rigidbody-Komponente (siehe Kap. 4.2). Außerdem besitzt das Spielobjekt eine NavMeshAgent-Komponente, deren Funktionsweise auf künstlicher Intelligenz basiert und zur Berechnung von Wegen genutzt wird (siehe Kap. 4.1).

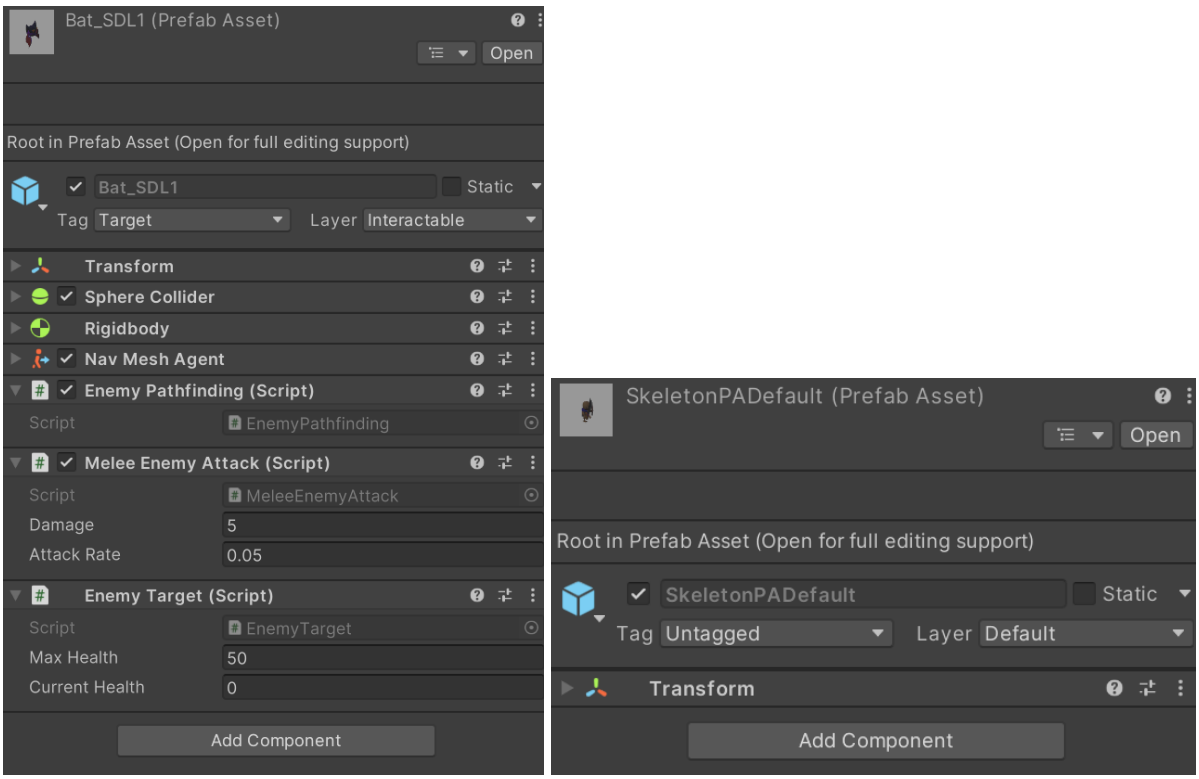

Abb. 17: Links: Vorhandenes Spielobjekt mit benötigten Komponenten. Rechts: Neues Spielobjekt ohne zusätzliche Komponenten

Neben diesen Komponenten sind drei Skripte vorhanden (siehe Abb. 17, Links). Das *EnemyPathfinding* Skript sorgt dafür, dass sich das gegnerische Spielobjekt zum Spielcharakter hinbewegt. Das Skript *MeleeEnemyAttack* löst das Verursachen des festgelegten Schadens am Spielcharakter aus. Das *EnemyTarget* Skript berechnet die Menge an Lebenspunkten, die das Spielobjekt besitzt und berechnet diese neu, wenn das Spielobjekt selbst Schaden nimmt. Ein neu hinzugefügter Gegner besitzt auf der Logikebene nur eine Transform-Komponente (siehe Abb. 17, Rechts). Um einen funktionierenden Gegner zu designen, müssen nur die benötigten Komponenten, sowie die drei Skripte, hinzugefügt werden. Die Skripte können dann noch hinsichtlich der Höhe des Angriffs-Schadens, der Angriffs-Geschwindigkeit und der Lebenspunkte angepasst werden (siehe Abb. 18).

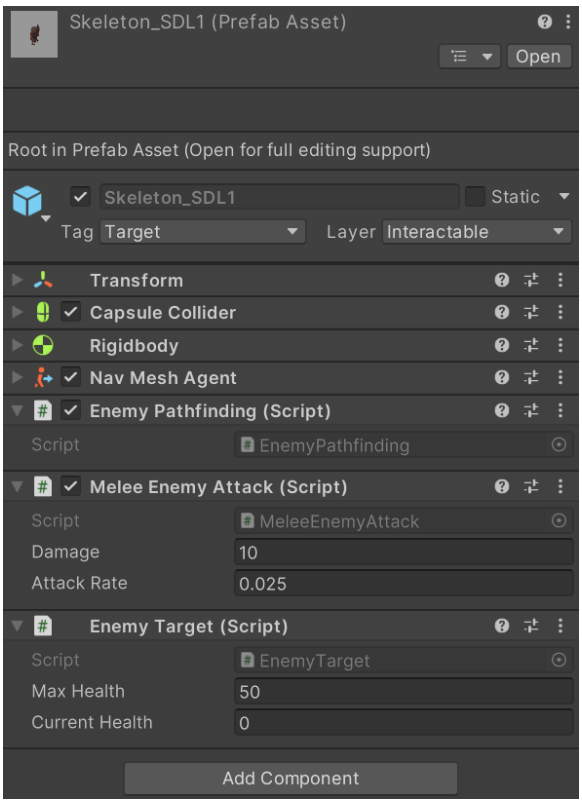

Abb. 18: Neu gestalteter Gegner mit allen benötigten Komponenten und Skripten

Diese Flexibilität ergibt sich aus der Anwendung des Single-Responsibility- und Open-Closed-Prinzips. Die verschiedenen Anliegen sind voneinander getrennt und die benötigten Skripte können als Module betrachtet werden. Es wäre auch eine Einheit möglich, die aufgrund des EnemyPathfinding Skripts zwar dem Spielcharakter folgt, ihn aber nicht angreift und selbst keinen Schaden erhält. In diesem Fall würde sie die Skripte MeleeEnemyAttack und EnemyTarget nicht besitzen. Stattdessen könnte der Einheit ein Skript zum Aufsammeln von Loot hinzugefügt werden, was eine neue Art von Einheit hervorbringen würde. Hinzu kommt, dass aufgrund der Einhaltung des Liskov'schen-Substitutions-Prinzips andere Arten von Gegnern hinzugefügt werden können, deren Schadensabwicklung modifiziert ist. Es könnte eine Art von Gegnern geben, die 2-fachen Schaden nimmt. Oder ein komplett anderes Verhalten ausführt, wenn Schaden genommen wird. Das ist möglich, da dieselbe Klasse Projektil, sowohl vom Spielcharakter als auch von Gegnern genutzt wird und diese nur über die Klasse Target das Verhalten der Methode TakeDamage() am getroffenen Spielobjekt auslöst (siehe Abb. 5).

Beispiel Nr. 2: Hinzufügen interaktiver Elemente

Aufgrund der Anwendung der SOLID-Prinzipien ist das entstandene Spiel mit wenig Aufwand erweiterbar (siehe Kap. 7.1.3). Die Schnittstellen *ICollectable* und *ITalkable* schaffen in Verbindung mit der abstrakten Klasse *InteractionObject* eine Struktur, auf die bei

Erweiterungen zurückgegriffen werden kann. Klassen von Spielobjekten die aufgesammelt werden können, wie beispielsweise Loot, erben von *InteractionObject* die abstrakte Operation *Interact()* und überschreiben diese. Implementiert wird die Schnittstelle *ICollectable* und ihre Operation *Collect(),* welche in *Interact()* aufgerufen wird. Was beim Ausführen der Methode in *Collect()* passiert, wird in den Klassen der Spielobjekte festgelegt (siehe Abb. 19).

```
public class CollectItem : InteractionObject, ICollectable
{
     private InventoryUI inventoryUI;
     private void Awake()
     {
          inventoryUI = GameObject.GetComponent<InventoryUI>();
     }
     private override void Interact()
     {
          Collect();
     }
     public void Collect()
     {
          inventoryUI.AddItem(inventoryUI.coin);
     }
}
```

Abb. 19: Die Klasse CollectItem mit der geerbten Operation Interact() und der implementierten Schnittstelle ICollectable und ihrer Operation Collect()

Da alle Gegenstände, die von Gegnern fallengelassen werden, zunächst im Inventar gesammelt werden, können die jeweiligen Spielobjekte das Skript mit der Klasse *CollectItem* als Komponente erhalten. Die Entwicklung mehrerer Schnittstellen nach dem Interface-Segregation-Prinzip und die Etablierung festgelegter Strukturen nach dem Dependency-Inversion-Prinzip bieten die Möglichkeit, das Spiel mit neuen, interaktiven Elementen zu füllen.

9 Fazit

In dieser Bachelorarbeit sollten der Nutzen und die Grenzen für die Anwendung der SOLID-Prinzipien in der Spieleentwicklung untersucht werden. Dazu wurde zunächst die thematische Grundlage geschaffen, indem Vorgehensweisen in der Softwareentwicklung beschrieben wurden (siehe Kap. 2). Anschließend wurde das Paradigma der objektorientierten Programmierung (siehe Kap. 3) beleuchtet, um eine Orientierung zu schaffen und der Arbeit einen Rahmen zu geben. Weiter eingegrenzt wurde das Thema unter Berücksichtigung der Spieleentwicklung (siehe Kap. 4) und ihrer für die späteren Anwendungsbeispiele relevanten Aspekte. Um einen Maßstab für die Untersuchung der SOLID-Prinzipien zu erhalten, wurden Ziele der Softwarearchitektur (siehe Kap. 5.1) erläutert. Der Themenkomplex wurde durch die Aspekte Softwaredesign (siehe Kap. 5.2) und Entwurfsmuster (siehe Kap. 5.3) weiter ausgeführt. Daraufhin folgte die Auseinandersetzung mit dem Untersuchungsgegenstand (siehe Kap. 6). Die einzelnen SOLID-Prinzipien wurden jeweils erst theoretisch erläutert und anschließend anhand von Anwendungsbeispielen aus der Spieleprogrammierung veranschaulicht. Unter Berücksichtigung der Ziele der Softwarearchitektur wurde untersucht, inwieweit die SOLID-Prinzipien zu einer guten Softwarearchitektur beitragen. Daraus wurde der Nutzen für die Spieleentwicklung abgeleitet (siehe Kap. 7.1). Anschließend wurde unter Hinzunahme von Literatur erörtert, wo die SOLID-Prinzipien an Grenzen stoßen. Dabei wurde beschrieben, wie bedeutend diese Grenzen für die Anwendung der SOLID-Prinzipien in der Spieleentwicklung sind (siehe Kap. 7.2). Zusätzlich wurde bestehende Kritik an den Prinzipien dargelegt (siehe Kap. 7.3). Darauf folgte die Erläuterung zum praktischen Teil dieser Arbeit (siehe Kap. 8). Es wurde der Prozess der Projektplanung dargelegt, sowie die Entwicklung des Spielkonzepts und der Spielmechaniken. Anschließend wurden unter Hinzunahme von Beispielen die konkreten Auswirkungen für die weitere Entwicklung des Spiels veranschaulicht, welche sich aus der Anwendung der SOLID-Prinzipien ergeben haben.

Aus der Untersuchung geht hervor, dass die SOLID-Prinzipien einen hohen Nutzen für die Spieleentwicklung haben können. So ergeben sich durch das Zusammenspiel der verschiedenen Prinzipien verschiedene Möglichkeiten in der Entwicklung. Spielobjekte können einfacher angeordnet werden, was das Erstellen von Szenen erleichtert. Ganze Teilsysteme eines Spiels können für spätere Projekte genutzt werden, wenn sich die Anforderungen an die Systeme ähneln. Daraus ergibt sich auch, dass neue Spielfunktionen oder auch ganze Spielerweiterungen ohne große Änderungen am bestehenden System in dieses eingepflegt werden können. Auch nach Veröffentlichung bleibt der Wartungsaufwand

gering, da das Spiel als Softwaresystem einerseits keine hohe Fehleranfälligkeit aufweist und andererseits strukturiert aufgebaut ist, sodass Fehler schnell behoben und Änderungen ohne das Risiko, das Restsystem zu beeinflussen, durchgeführt werden können. Diese nützlichen Aspekte unterliegen jedoch gewissen Einschränkungen. So ist die Anwendung der SOLID-Prinzipien zum einen mit einem höheren Planungsaufwand in der Spieleentwicklung verbunden. Zum anderen wird das Softwaredesign eines Spiels durch die zu häufige Anwendung der SOLID-Prinzipien schnell unübersichtlich. Vor allem bei komplexen Softwaresystemen wie Spielen ist zu kompliziertes Softwaredesign ein bedeutendes Problem. Die Prinzipien in einem guten Maß und bedacht anzuwenden ist dabei abhängig von der Erfahrung der jeweiligen Entwickler und Entwicklerinnen. Außer diesen Grenzen besteht zudem Kritik an den SOLID-Prinzipien, die sich darauf stützt, dass heute Möglichkeiten in der Softwareentwicklung bestehen, die zur Zeit der Entstehung der Prinzipien nicht vorhanden waren.

Zusammenfassend lässt sich also feststellen, dass die SOLID-Prinzipien trotz Kritik nützlich für die Entwicklung von Spielen sein können, wenn sie mit Bedacht und nicht nur aus Prinzipientreue angewendet werden. Für das selbst entwickelte Spiel bieten sich zukünftig die beschriebenen Möglichkeiten. Dennoch ist die Kritik an den Prinzipien nachvollziehbar und stellt einen relevanten Gegenstand für weitere Untersuchungen dar. So könnte überprüft werden, unter welchen Bedingungen die Kritik zutreffend ist und ob eine Modifizierung der Prinzipien möglich und nötig ist, um sie an heutige Anforderungen in der Software- und Spieleentwicklung anzupassen. Für weiterführende Untersuchungen kann die Anforderungsanalyse eines zu entwickelnden Spiels in Betracht gezogen und eine daraus resultierende Softwarearchitektur festgelegt werden. Auf die Untersuchung könnte eine Zuordnung sinnvoll einsetzbarer Entwurfsmuster folgen.

10 Literaturverzeichnis

Borromeo, N. A. (2021). *Hands-On Unity 2021 Game Development. Create, customize, and optimize your own professional games from scratch with Unity 2021.* (2. Auflage). Birmingham: Packt Publishing Ltd.

Brandt-Pook, H. & Kollmeier, R. (2020). *Softwareentwicklung kompakt und verständlich: Wie Softwaresysteme entstehen* (3. Auflage). Wiesbaden: Springer Fachmedien Wiesbaden GmbH.

Broy, M., Geisberger, E., Kazmeier, J., Rudorfer, A. & Beetz, K. (2007). *Informatik-Spektrum. Ein Requirements-Engineering-Referenzmodell*. Berlin: Springer-Verlag. doi: 10.1007/s00287-007-0149-5

Dirschnabel, S. (2018). *SOLID [4] – Das Interface Segregation Principle.* Zugriff am 25.01.2024. Verfügbar unter: https://www.informatik-aktuell.de/entwicklung/methoden/solid-4-das-interface-segregation-principle.html

Drähter, R., Koschek, H. & Sahling, C. (2023). *Scrum. kurz & gut.* (3. Auflage). Heidelberg: dpunkt.verlag GmbH.

Dunkel, J. & Holitschke, A. (2003). *Softwarearchitektur für die Praxis.* Heidelberg: Springer-Verlag.

Fahme, M.-U.-S. & Khan, T. H. (2021). *How to Make a Game: Go From Idea to Publication Avoiding the Common Pitfalls Along the Way.* Berkeley, CA: Apress.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (2015). *Design Patterns. Entwurfsmuster als Elemente wiederverwendbarer objektorientierter Software* (1. Auflage). Frechen: mitp Verlags GmbH & Co. KG.

Gharbi, M., Koschel, A., Rausch, A. & Starke, G. (2023). *Basiswissen für Softwarearchitekten. Aus- und Weiterbildung nach iSAQB-Standard zum Certified Professional for Software Architecture – Foundation Level (5. Auflage).* Heidelberg: dpunkt.verlag GmbH.

Heise Online. (2022). *Software-architektur.tv: Design Principles – SOLID vs. CUPID.* [Videocast von Eberhard Wolff, Interview mit Daniel Terhorst-North]. Hannover: Heise Medien GmbH & Co. KG. Verfügbar unter: https://www.heise.de/news/software-architektur-tv-Design-Principles-SOLID-vs-CUPID-6337369.html

Lahres, B., Raýman, G. & Strich, S. (2018). *Objektorientierte Programmierung. Das umfassende Handbuch* (4., aktualisierte Auflage). Bonn: Rheinwerk Verlag.

Lilienthal, C. (2020). *Langlebige Softwarearchitekturen. Technische Schulden analysieren, begrenzen und abbauen.* (3. Auflage). Heidelberg: dpunkt.verlag GmbH.

Martin, R. C. (2018). *Clean Architecture. Das Praxis-Handbuch für professionelles Softwaredesign. Regeln und Paradigmen für effiziente Softwarestrukturierung.* (1. Auflage). Frechen: mitp Verlags GmbH & Co. KG.

Nystrom, R. (2015). *Design Patterns für die Spieleprogrammierung.* Frechen: mitp Verlags GmbH & Co. KG.

Schatten, A., Demolsky, M., Winkler, D., Biffl, S., Gostischa-Franta, E. & Östereicher, T. (2010). *Best Practice Software-Engineering. Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen.* Heidelberg: Spektrum Akademischer Verlag.

Schell, J. (2020). *Die Kunst des Game Designs. Bessere Games konzipieren und entwickeln.* (3. Auflage). Frechen: mitp Verlags GmbH & Co. KG.

Unity Technologies (2023). *Unity Engine.* Verfügbar unter: https://unity.com/de/products/unity-engine

Unity Technologies (2024a). *Unity User Manual 2022.3 (LTS). Transforms.* Verfügbar unter: https://docs.unity3d.com/Manual/class-Transform.html

Unity Technologies (2024b). *Unity User Manual 2022.3 (LTS). Mesh Filter component.* Verfügbar unter: https://docs.unity3d.com/Manual/class-MeshFilter.html

Unity Technologies (2024c). *Unity User Manual 2022.3 (LTS). Collider 2D.* Verfügbar unter: https://docs.unity3d.com/Manual/Collider2D.html

Unity Technologies (2024d). *Unity User Manual 2022.3 (LTS). Rigidbody component reference.* Verfügbar unter: https://docs.unity3d.com/Manual/class-Rigidbody.html

Unity Technologies (2024e). *Unity User Manual 2022.3 (LTS). Character Controller component reference.* Verfügbar unter: https://docs.unity3d.com/Manual/class-CharacterController.html

Unity Technologies (2024f). *Unity User Manual 2022.3 (LTS). Creating and Using Scripts.* Verfügbar unter: https://docs.unity3d.com/2022.3/Documentation/Manual/CreatingAndUsingScripts.html

Verband der deutschen Games-Branche e.V. (2023). *Jahresreport der deutschen Games-Branche 2023.*

11 Abbildungsverzeichnis

12 Eigenständigkeitserklärung

Technische Hochschule Ostwestfalen-Lippe

Fachbereich Medienproduktion

FB MP Vers. 04/21Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit mit dem Titel:

Untersuchung der SOLID-Prinzipien: Nutzen und Grenzen in der Spieleentwicklung

Selbstständig, ohne unerlaubte fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Die Stellen der Arbeit, die dem Wortlaut oder dem Sinn nach anderen Werken (dazu zählen auch Internetquellen) entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht.

Bielefeld, 16.02.2024, T. Sobiech

Ort, Datum, Unterschrift