Architektur des Kerns einer Game-Engine und Implementierung mit Java

Alexander Dammerer

DIPLOMARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Digitale Medien

in Hagenberg

im Dezember 2010

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 29. November 2010

Alexander Dammerer

Inhaltsverzeichnis

Erklärung **iii**

Kurzfassung **vii**

Abstract **viii**

1 Einleitung **1**
1.1 Motivation 2
1.2 Zielsetzung 3
1.3 Aufbau dieser Arbeit 4

2 Grundlagen und Stand der Technik **5**
2.1 Komponentenbasierte Software-Entwicklung 5
2.1.1 Grundlegende Funktionsweise 5
2.1.2 Elemente einer komponentenbasierten Architektur . . 5
2.1.3 Komposition eines Spielobjektes 6
2.2 Kommunikation 8
2.2.1 Referenzen 8
2.2.2 Dataports 9
2.2.3 Nachrichten und Events 9
2.2.4 Signals und Slots 9
2.3 Sprachfeatures 10
2.3.1 Reflection 10
2.3.2 Annotation 10
2.3.3 Verfügbarkeit in den Sprachen 11
2.4 Möglichkeiten für Identifikatoren 12
2.4.1 Zahlenwerte und Enumerationen 12
2.4.2 String 12
2.4.3 Class-Objekte 13
2.5 Komponentenbasierte Engines/Frameworks 14
2.5.1 Elephant (C#) 14
2.5.2 PushButton (AS3) 14
2.5.3 Unity3D 14
2.5.4 Cogaen/CogaenJ 15

2.6 Java 16
2.6.1 JOGL 16
2.6.2 JOAL 16
2.6.3 Java Native Interface 16
2.6.4 Java Plug-In Framework 17
2.6.5 JMonkey Engine 17

3 Entwurf **18**
3.1 Problemstellung 18
3.1.1 Modularität 19
3.1.2 Lebenszyklen 19
3.1.3 Spielobjekt-Verwaltung 20
3.1.4 Kommunikation 20
3.2 Überblick über die Architektur 21
3.3 Plug-In Mechanismus 22
3.3.1 Anforderungen 22
3.3.2 Bestandteile 24
3.3.3 Abgedeckte Funktionalitäten 26
3.3.4 Lebenszyklus der Plug-Ins 29
3.3.5 Lebenszyklus des Kerns 30
3.3.6 Lebenszyklus der Applikation 31
3.3.7 Erweiterungsmöglichkeiten 32
3.4 Spielobjekt-Verwaltung 35
3.4.1 Anforderungen 35
3.4.2 Probleme 37
3.4.3 Bestandteile 38
3.4.4 Abgedeckte Funktionalitäten 41
3.4.5 Lebenszyklus der Komponenten 43
3.4.6 Lebenszyklus der Spielobjekte 44
3.4.7 Erweiterungsmöglichkeiten 45
3.5 Kommunikation 47
3.5.1 Anforderungen 47
3.5.2 Bestandteile 49
3.5.3 Verknüpfung mit dem Komponenten-Modell 51
3.5.4 Kommunikationskanäle 52
3.5.5 Erweiterungsmöglichkeiten 54

4 Implementierung **55**
4.1 Setup 55
4.2 Plug-In Mechanismus 56
4.2.1 Architektur 56
4.2.2 Identifikation von Modulen und Plug-Ins 60
4.2.3 Verwaltung der Abhängigkeiten 62
4.2.4 Ermittlung der Startreihenfolge 65

4.2.5 Der eingebaute Schutzmechanismus 66
4.2.6 Die Initialisierung einer Applikation 67
4.2.7 Die Verwaltung der Plug-Ins 67
4.3 Spielobjekt-Verwaltung 68
4.3.1 Architektur 68
4.3.2 Die Implementierung eigener Komponenten 71
4.3.3 Die Erzeugung der Komponenten 75
4.3.4 Die Implementierung von Spielobjekten 76
4.3.5 Die Erzeugung von Spielobjekten 79
4.4 Kommunikation 81
4.4.1 Architektur 81
4.4.2 Die Einbettung in das Komponenten-System 84
4.4.3 Die Anwendung von Messages 85
4.4.4 Die Anwendung von Events 85
4.4.5 Die Anwendung von Signals und Slots 85

5 Schlussbemerkungen 87
5.1 Ergebnisse 87
5.1.1 Modularität 87
5.1.2 Spielobjekt-Verwaltung 88
5.1.3 Kommunikation 88
5.1.4 Einfache Anwendbarkeit 89
5.2 Ausblick 89

A UML-Diagramme 91

B Inhalt der CD-ROM 94
B.1 Diplomarbeit 94
B.2 Literatur 94
B.3 Projekt 94

Literaturverzeichnis 95

Kurzfassung

Die Entwicklung von Videospielen hat sich seit den Tagen, an denen ein einzelner Entwickler jedes Spiel von Grund auf alleine entwickelt hat, enorm verändert. Die moderne Entwicklung von Spielen beinhaltet große Teams und Game-Engines, welche eine wiederverwendbare Basis für die Programmierer bieten. Da jedes Spiel einen unterschiedlichen Bedarf an Funktionalität hat, muss eine Game-Engine leicht zu modifizieren sein.

Die vorgestellte Architektur des Kerns einer Game-Engine nutzt einen Plug-In Mechanismus, welcher einen einfachen Wechsel der zur Verfügung gestellten Funktionalität erlaubt. Um dasselbe Maß an Modifizierbarkeit im Bereich der Spielobjekt-Verwaltung zu bieten, verfolgt die Architektur den Ansatz der komponentenbasierten Entwicklung. Dies erlaubt die Komposition von Spielobjekten ohne die unabhängigen Funktionalitäten starr aneinander zu binden. Um dies zu ermöglichen, wird auch ein Kommunikations-Mechanismus benötigt, der den Komponenten erlaubt sich auf flexible und unabhängige Weise zu verständigen. Das Kommunikations-System nutzt Referenzen, *messages*, *events* und einen Kanal auf Basis von *signals and slots* um die Komponenten zu verbinden.

Die entwickelte Game-Engine, genannt *Java Game Components* (JGC), ist der implementierte Machbarkeitsnachweis. Sie macht starken Gebrauch von *Reflection*, *Annotation* und der generischen Programmierung um häufige Aufgaben zu automatisieren und Funktionalitäten wie *Dependency Injection* zur Verfügung zu stellen. Das Ziel der Implementierung war es, die Architektur sowohl einfach anwendbar als auch einfach erweiterbar zu machen. Die Ideen für die vorgestellte Implementierung entstanden während einer Reihe von Projekten für *Cogaen*/*CogaenJ* (Component-Based Game Engine for Java).

Abstract

Since the days when a single developer created every game from scratch the development of games has changed enormously. Modern game development includes large teams and game engines which provide a reusable base for the programmers. Because each game has a different demand for functionality the engine has to be modifiable easily.

The presented architecture of the core of a game engine uses a plug in mechanism which allows an easy change of included functionality. To provide the same level of modifiability at the game object management the architecture follows the component-based approach. This facilitates the composition of game objects without tying the independent functionality together. This also requires a communication mechanism which enables the components to communicate in an independent and flexible way. The communication system uses references, messages, events and a signal and slot based channel to connect the components.

The developed game engine, called *Java Game Components* (JGC), provides the proof of concept implementation. It makes strong use of Reflection, Annotation and generic programming in order to automate common tasks and provide functionality like dependency injection. The goals of the implementation have been to make the architecture easy to use and easy to expand. The ideas for the presented implementation arose during the work on a series of projects for *Cogaen*/*CogaenJ* (Component-Based Game Engine for Java).

Kapitel 1

Einleitung

Die Entwicklung von Videospielen hat seit der Entstehung der ersten Spiele immer größere Ausmaße angenommen. Um heute auf dem Markt konkurrenzfähig zu bleiben werden immer größere Mengen an Assets, immer aufwändigeres Game-Play und stetig wachsende Welten benötigt. Zusätzlich werden funktionsreichere Engines für Grafik, Physik und Audio entwickelt um die Spieler zufrieden stellen zu können und aufs Neue zu beeindrucken. Die Entwicklung der Spiele und der verwendeten Engines ist mit viel Arbeitsaufwand und somit mit hohen Kosten verbunden. Budgets von mehreren Millionen Euro für die Entwicklung eines Vollpreis-Spieles sind seit langem keine Seltenheit mehr. Ein Weg um die Kosten für die Entwicklung zu senken ist die Entwicklung einer Game-Engine die für mehrere Spiele wiederverwendet werden kann und so zumindest auf Seiten der Programmierung Zeit- und Kostenersparnis bringt. Eine zusätzliche Alternative ist der Kauf oder die Lizenzierung einer bereits bestehenden Engine sowie Erweiterungen von Drittanbietern, da dies in vielen Fällen günstiger ist als die benötigten Features selbst zu implementieren. Dadurch wird es den Spieleentwicklern ermöglicht sich auf die Erstellung der für das Spiel relevanten Inhalte zu konzentrieren und den Großteil der Programmierung an andere Unternehmen auszulagern. Die Spezialisierung der Unternehmen auf bestimmte Teilgebiete (Grafik, Physik, Audio, künstliche Intelligenz) ermöglicht zugleich eine größere Anzahl an zur Verfügung stehenden Features sowie eine höhere Qualität. Dies ergibt sich aus dem Umstand, dass Fehler in den einzelnen Teil-Engines durch die Verwendung in mehreren Spielen bereits entdeckt und ausgebessert wurden. Insgesamt ergibt sich dadurch die angestrebte Konkurrenzfähigkeit durch einem wachsenden Funktionsumfang bei reduziertem Entwicklungsaufwand.

Ein zur Entwicklung von AAA-Spielen entgegengesetzter Trend ist *Social Gaming*, welcher aus dem *Browser Game Boom* entstanden ist. Hier werden nicht immer größere Spiele mit noch aufwendigerer Technik entwickelt sondern eine große Menge an *Casual Games* für die Massen, welche über *So-*

cial Networks wie *Facebook* vertrieben werden. Zur Zeit werden noch keine großen Ansprüche an die Technik gelegt, 2D Grafik und Stereosound reichen in den meisten Fällen aus. Zusätzlich erreicht der Entwickler über die Plattformen Millionen von potenziellen Spielern auf einen Schlag. Die Entwicklung von Spielen für diesen Markt ist von kleinen Teams in nur wenigen Monaten schaffbar, wogegen die Entwicklung von Vollpreistiteln Teams mit Größen von bis zu 100 Mitgliedern mehrere Jahre beschäftigt. Gewinn wird auf diesem Sektor hauptsächlich nicht mehr durch den Verkauf der Spiele selbst, sondern durch die Einblendung von Werbung oder über sogenannte Mikrotransaktionen (Kauf von Zusatzfunktionalität im Spiel) erzielt. Selbst erfolgreiche Spieleentwickler von AAA-Titeln wie *Sid Meier* sprechen auf diesen Trend an und entwickeln Ableger ihrer Spiele (*Civilization Network*) für *Facebook* [7]. Zahlreiche *Browser Games* werden mit web-basierten Technologien wie HTML, PHP, JavaScript und Flash/ActionScript entwickelt. Jedoch stoßen Entwickler aufwändigerer Spiele auch mit Flash bereits an ihre Grenzen[2]. Ein zusätzlicher Markt für *Casual Games* entstand durch die Einführung des *iPhones*, was den mobilen Spielesektor aufwertete. Über den *App Store* werden tausende Spiele angeboten und erreichen den Kunden jederzeit ohne von der Couch aufzustehen. Auch hier werden Großteils kleine *Casual Games* angeboten und zu geringen Preisen verkauft (im Schnitt zwischen 2 und 5 Euro). Dem Trend folgen nicht nur die Entwickler der Spiele sondern auch die Entwickler der Engines rüsten nach. Als Paradebeispiele dafür dient *Unity Technologies*, welche für ihre Game-Engine *Unity3D*[1] einen Webplayer anbieten um die Spiele im Browser spielbar zu machen, und *Epic Games*[2], welche gerade die *Unreal Engine 3*[3] für das *iPhone* lauffähig machen. Bedenkt man die großen Schwierigkeiten die bei solchen Portierungen auftreten können (*CryTek* benötigte im Vergleich zu *Epic* lange um ihre Engine auf Konsolen lauffähig zu machen) bleibt abzuwarten ob sich diese Engines für vergleichbar schwache Endgeräte durchsetzen können.

1.1 Motivation

Beiden Trends gemeinsam ist eine dahinterstehende Engine welche für mehrere Spiele verwendet werden kann. Während die Entwicklung der Engines in der Industrie immer weiter vorangetrieben wird gibt es auf diesem Gebiet der Forschung noch großen Aufholbedarf. Der Großteil der erhältlichen Literatur hat einen starken Fokus auf Rendertechniken. Die darin konzipierten Engines sind eng mit der jeweils gewählten Grafik-Technologie (OpenGL, DirectX) verknüpft und schwer austauschbar. Auf die dahinterliegenden Funktionalitäten einer Game-Engine wird nur selten eingegangen. In [1] wird die

[1] http://unity3d.com
[2] www.epicgames.com
[3] www.unrealtechnology.com

Problematik wie folgt beschrieben:

> There appears to be general agreement that game engines are not only useful, but due to the complexity of modern computer games, are actually required for game development. Given this, there exists a surprisingly small body of literature on game engine design. The available research has mainly focussed on game engine subsystems, such as rendering, AI (artificial intelligence) or networking. However, issues regarding the overall architecture of engines, which connects these subsystems, have merely been brushed over.

Obwohl die Forschung im Spielebereich noch zurückliegt, können Ergebnisse aus anderen Gebieten der Informatik verwendet werden, da sich die Entwicklung einer Game-Engine Großteils mit der Entwicklung anderer Software gleicht. Unterschiede ergeben sich durch die Anforderungen an das Endprodukt, welches sich durch eine starke Interaktion mit dem Kunden (Spieler) sowie der Elemente untereinander auszeichnet.

Da der Kern und dessen Architektur das Fundament einer jeden Game-Engine bilden, ist es zu Beginn notwendig, diesen zu erforschen. Kleine Änderungen in diesem Bereich des Konzepts oder der dazugehörigen Implementierung ziehen in der Regel aufwändige Refactoring-Arbeiten nach sich, welche sich auf alle angebundenen Module erstreckt.

In der vorliegenden Arbeit bezieht sich der Begriff Game-Engine auf den gesamten zur Verfügung stehenden Funktionsumfang der Applikation inklusive seiner Subsysteme. Ebenfalls eingeschlossen ist der umgebende Workflow mit Editor und weiteren Entwicklungstools. Diese Auffassung des Begriffes bezieht sich auf [4] wo durch eine Umfrage erhoben wurde, worauf amerikanische Spieleentwickler beim Kauf einer Engine achten.

Die Kernarchitektur ist dabei die verknüpfende Schicht mit der Aufgabe der Verwaltung der Module (oben Subsystem genannt), sowie der Verknüpfung der vorhandenen Funktionalitäten zu logischen Elementen. Auch muss diese Schicht bereits ein für die Engine einheitliches Kommunikationskonzept zur Verfügung stellen auf welches die Module aufbauen können.

1.2 Zielsetzung

Ziel dieser Arbeit ist die Konzeption der Kernarchitektur einer Game-Engine. Betrachtet man den unterschiedlichen Funktionsumfang der einzelnen Spiele muss diese Architektur so aufgebaut sein, dass bei Bedarf Funktionalität hinzugefügt, entfernt und ausgetauscht werden können. Auch soll die Architektur in der Lage sein Abhängigkeiten untereinander aufzulösen.

Da die einzelnen Module voneinander so unabhängig wie möglich entwickelt werden, wird ein Mechanismus benötigt, welcher diese Elemente inner-

halb der Engine wieder zu konkreten Objekten zusammenfasst. Es gilt die unterschiedlichen bereits bestehenden Lösungsansätze zu vergleichen und jene zu wählen, welche sich für das Gesamtkonzept am besten eignen.

Nachdem ein Spiel stark von der Interaktion der Objekte und der Module geprägt ist, wird auch ein einheitliches Kommunikationssystem benötigt. Es wird auch auf die Inter-Objekt-Kommunikation (Kommunikation zwischen den Objekten) sowie die Intra-Objekt-Kommunikation (Kommunikation innerhalb der Objekte) eingegangen.

Es sollen auch potenzielle Problemstellen aufgezeigt werden, welche sich erst im Zusammenspiel mehrerer Module ergeben. Ein großes Problem stellen Module dar, welche ihre Aufgaben intern über mehrere Update-Zyklen verteilt erledigen. Dadurch entsteht eine Unsicherheit ob eine Aufgabe bereits abgeschlossen wurde oder nicht. Verlässt sich ein anderes Modul darauf, dass eine Aufgabe bereits erledigt wurde kann es im schlimmsten Fall zu Abstürzen kommen, sollte dies nicht der Fall sein. Es sollen in dieser Arbeit auch Lösungsansätze für dieses konkrete Problem aufgezeigt werden.

1.3 Aufbau dieser Arbeit

Im ersten Kapitel wird der Fokus der Arbeit definiert und die Zielsetzung erklärt. Auch auf die Relevanz der vorliegenden Arbeit wird eingegangen.

Kapitel 2 gibt einen Überblick über den derzeitigen Stand der Technik auf den für die Arbeit relevanten Gebieten sowie über gebräuchliche Lösungsansätze für die aufgezeigten Problematiken.

Im dritten Kapitel werden die Ansätze zu dem gewünschten Konzept einer Kernarchitektur für eine Game-Engine verknüpft. Dabei wird auf die jeweiligen Vor- und Nachteile der Lösungsansätze eingegangen.

Das Kapitel Implementierung beschreibt die Erfahrungen aus der Erstellung von *Java Game Components* (JGC), dem im Vorfeld erstellten Praxisteil dieser Arbeit. Es wird auch auf die Eigenschaften der Sprache Java eingegangen. Besonders die Vor- und Nachteile für die beigelegte Implementierung, welche sich durch Java ergeben, werden hier vermittelt.

Den Abschluss bildet das fünfte Kapitel mit der Auswertung der Ergebnisse sowie Hinweisen auf noch bestehende Probleme im Modell und der Implementierung.

Kapitel 2

Grundlagen und Stand der Technik

2.1 Komponentenbasierte Software-Entwicklung

2.1.1 Grundlegende Funktionsweise

Die Idee des Component-Based Software Engineering (CBSE) ist es, die Objekte, vergleichbar mit einem Baukasten-System, aus einzelnen kleinen Elementen (Komponenten) zusammenzusetzen. Diese einzelnen Bauteile sind so gestaltet, dass sie über vorhandene Schnittstellen miteinander interagieren. Jede Komponente ist dabei (abgesehen von den gemeinsamen Schnittstellen) möglichst unabhängig und kann daher einfacher erstellt und getestet werden [23].

2.1.2 Elemente einer komponentenbasierten Architektur

Bei der Erstellung einer Software auf Basis von Komponenten reicht es nicht, einzelne Komponenten zu entwickeln. Zusätzlich wird die Definition von Standards benötigt, über welche die Komponenten miteinander interagieren, wie sie erzeugt werden und welche Möglichkeiten sich für die Komposition bieten. Heineman und Councill definieren die folgenden vier Kernelemente[11] (Übersetzung entnommen aus [19]):

> Eine **Softwarekomponente** ist ein Softwareelement, welches einen Komponentenmodell konform ist, unabhängig entwickelt werden kann und ohne Modifikation gemäß einem Kompositionsstandard mit anderen Softwarekomponenten kombiniert werden kann.
>
> Das **Komponentenmodell** definiert spezifische Kompositions- und Interaktionsstandards.

> Eine **Komponentenmodell-Implementierung** enthält Softwareelemente, welche benötigt werden, um die Ausführung der Komponente, welche dem Modell konform ist, zu unterstützen.
>
> Eine **Softwarekomponenten-Infrastruktur** stellt eine Menge von interagierenden Softwarekomponenten dar, die dafür bestimmt ist, dass das Softwaresystem oder Subsystem, welches mit diesen Komponenten konstruiert wurde, den klar definierten Leistungsspezifikationen entspricht.

Ein Bestandteil des Komponentenmodells sind die **Lebenszyklen**. Diese definieren Phasen in denen eine Komponente vorbereitende bzw. abschließende Aufgaben erledigen und ihre eigentliche Funktion erfüllen kann. Die Einhaltung des Lebenszyklus ist ein wichtiges Kriterium für die Konformität mit dem Modell.

2.1.3 Komposition eines Spielobjektes

Klassischer objektorientierter Ansatz

In der objektorientierten Programmierung werden Spieleobjekte über eine Vererbungshierarchie erstellt. Die Basis bildet eine Klasse (z. B. `GameObject`) von der alle Objekte abgeleitet werden. Die Basisklasse enthält dabei nur jene Funktionalität, die für alle Klassen notwendig ist (z. B. für die Serialisierung). Ein Beispiel für eine solche Vererbungshierarchie zeigt Abbildung 2.1(a). Bei aufwändigen Spielen treten dabei eine Reihe von Problemen auf, was in erster Linie daran liegt, dass die Hierarchie sehr groß und unübersichtlich wird. Die häufigsten davon, die *Gregory* in seinem Buch erwähnt, sollen hier nun kurz erklärt werden [8]:

Bubble-Up Effect: Besteht die Notwendigkeit, eine existierende Funktionalität einer Klasse, einem anderen Zweig zur Verfügung zu stellen, wandert diese in der Hierarchie nach oben. Dadurch werden die höheren Klassen der Hierarchie aufgebläht und unübersichtlich. Gregory nennt auch die *Unreal Engine* als ein praktisches Beispiel für dieses Problem. Die Basisklasse *Actor* beinhaltet alle Funktionalität, angefangen von Grafik und Physik bis zu den Methoden für die Netzwerk-Engine.

Deadly Diamond: Ein anderer Ansatz um die selbe Funktionalität in zwei Klassen zur Verfügung zu stellen, ist die Verwendung von multipler Vererbung (z. B. in C++). Diese führt aber zu dem Problem, dass eine Basisklasse doppelt vorhanden ist und Zweideutigkeiten entstehen.

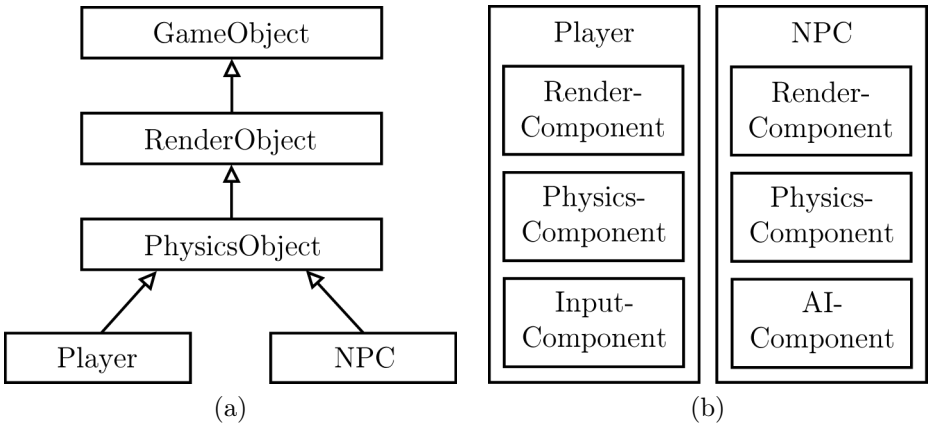


Abbildung 2.1: Darstellung der Komposition von Spielobjekten über eine Vererbungshierarchie (a) und über den komponentenbasierten Ansatz (b).

Komponentenbasierter Ansatz

Der komponentenbasierte Ansatz soll die Probleme, die durch zu komplexe Hierarchien entstehen, lösen. Dazu wird die Hierarchie komplett aufgebrochen und die Funktionalität in einzelne Komponenten verlagert. Dadurch wird es möglich, dass jedes Spielobjekt voneinander unabhängig ist und nur jene Komponenten (somit auch nur jene Funktionalität) beinhaltet, die es auch benötigt. Die Abbildung 2.1 (b) zeigt die Komposition zweier voneinander unabhängigen Spielobjekte. Eine Vererbungshierarchie bei den Komponenten ist durchaus möglich und innerhalb eines einzelnes Teil-Systems (z. B. für Rendering) durchaus üblich. Bedacht werden muss bei einer Vererbung jedoch, dass jede Komponente eindeutig identifizierbar bleibt um die Grundlage für eine automatische Verknüpfung zu erhalten.

Um die Erstellung komplexer Spielobjekte zu erleichtern und die Entwicklung dadurch weiter zu beschleunigen, ist die Unterstützung spezieller Komponenten-Typen vorgesehen. Diese sollen es ermöglichen, bestimmte Teile einer Komposition zu erstellen und in mehreren Spielobjekten wieder zu verwenden. Ein erster Ansatz dafür ist es zu ermöglichen einem Spielobjekt ein anderes Spielobjekt unterzuordnen. Ein konkreter Anwendungsfall dafür, ist die Lebensanzeige eines Gegners. Dessen Komposition besteht aus mehreren Komponenten, die bei der Verwendung bei mehreren Gegner immer wieder neu erstellt werden müsste. Über untergeordnete Spielobjekte, wird die Lebensanzeige einmal erstellt und jedem Gegner hinzugefügt. Die Lebensanzeige verhält sich dabei wie eine normale Komponente, die konkrete Schnittstellen beim übergeordneten Spielobjekt voraussetzt und sich über diese in die Gesamt-Komposition integriert.

2.2 Kommunikation

In interaktiven Spielwelten und Simulationen besteht zwischen den einzelnen Elementen (Entitäten, Spielobjekten) die Notwendigkeit miteinander zu interagieren, Daten auszutauschen und Reaktionen auszulösen. Ohne diese wäre es nicht möglich, dass zwei Spielobjekte miteinander kollidieren, ein Spieler gegen ein Monster kämpft oder die künstliche Intelligenz eine Armee in Formation über das Schlachtfeld führt.

Für jede Art der Kommunikation wird ein Kanal[1] benötigt. Der Kanal beschreibt dabei einen Weg zur Übermittlung von Informationen. Obwohl in der Regel der Austausch von Information die Absicht der Kommunikation ist, muss diese nicht immer Daten beinhalten. Oft ergibt sich die Information aus der Tatsache des Erhalts einer Nachricht. Für solche Fälle ist es von Vorteil datenlosen Kommunikationskanäle zur Verfügung zu stellen.

Für die Kommunikation zwischen Objekten ist es nicht zwingend notwendig, dass sich die Objekte kennen und über ihre Schnittstellen Bescheid wissen. Generell genügt es, wenn sie den Kommunikationskanal und das verwendete „Protokoll“ kennen. Den unterschiedlichen Kommunikationskanälen übergeordnet gibt es dabei zwei unterschiedliche Arten von Kommunikation:

Inter-Objekt-Kommunikation: Beschreibt die Kommunikation von zwei oder mehreren getrennten Spielobjekten. Sonderfälle davon treten im Bereich der Netzwerkspiele auf, wo die Kommunikation zusätzlich zwischen Client und Server erfolgen kann.

Intra-Objekt-Kommunikation: Besonders in der komponentenbasierten Spieleentwicklung ist es notwendig, dass die Bausteine eines einzelnen Spielobjektes miteinander kommunizieren.

2.2.1 Referenzen

Der einfachste Weg zur Kommunikation zwischen Objekten ist deren Referenzierung und der direkte Aufruf der Methoden. Dies ist jedoch nicht immer möglich, da unter Umständen weder die konkrete Implementierung noch ein implementiertes Interface bekannt sind. Ebenso ist die direkte Referenzierung in manchen Fällen nicht von Vorteil, da die referenzierten Daten von einem System verwaltet werden, dass bei Änderungen der Daten keine Möglichkeit hat neue Referenzen zu setzen. Somit ist dies zwar der schnellste Kommunikationskanal aber auch der unflexibelste.

[1] de.wikipedia.org/wiki/Kanal_(Informationstechnologie)

2.2.2 Dataports

Eine Erweiterung der direkten Referenzierung bilden Dataports. Linklater beschreibt Dataports wie folgt [17]:

> Essentially, a Dataport is a data structure which has has a unique global identity at runtime. ... Once the Dataport has been registered, Dataport pointers can request to be connected to a given Dataport via the Dataport Manager. Dataport pointers attach to and detach from Dataports at runtime, meaning that the dataflow between code modules can be dynamically manipulated.

Entwickelt wurden sie für die dynamische Einbindung von nachträglichen Erweiterungen, welche vom Spieler aus dem Internet geladen wurden. Zusätzlich kann dieses System jedoch auch für die normale Kommunikation verwendet werden.

2.2.3 Nachrichten und Events

Eine Alternative zur Bindung von Daten an eine Referenz ist die Bindung der Kommunikationsteilnehmer an einen vordefinierten Kanal. Harmon schlägt diesbezüglich vor, die gesamte Funktionalität einer Engine über Nachrichten zu lösen und verweist dabei auf die Funktionsweise der Win32-API (siehe [10]). Ein Spielobjekt ist in seinem System vollständig unabhängig von Inhalt und Funktion und dazu da um auf Nachrichten zu reagieren. Erhält es keine Nachrichten, wird sie keine Funktion ausführen (auch keine Updates und kein Rendering).

Das Nachrichten-System wurde für das vorliegende Modell übernommen (so bildet diese Arbeit die Grundlage für den `EventManager`). Auf die Umsetzung nur über Nachrichten wurde jedoch verzichtet.

2.2.4 Signals und Slots

Signals und Slots sind eine Erweiterung des Observer-Patterns (siehe [13, Kapitel 33]) und wurden ursprünglich für die Bibliothek *QT* von *Trolltech Inc.* entwickelt. Das System sieht vor, dass Signale von bestimmten Ereignissen ausgelöst werden. Dabei bezeichnen Signals die Ausgänge und Slots die Eingänge. Im Grunde sind sowohl Signals als auch Slots Zeiger auf Funktionen, was zur Folge hat, dass die Signatur der Funktionen für die Verknüpfung identisch sein muss [22].

Über die Deutlichkeit der Namensgebung des Systems (was bezeichnet den Ein- und was den Ausgang) gibt es unterschiedliche Auffassungen (siehe [9, Abschnitt 3.1]). Auch in der vorliegenden Implementierung wurde die Namensgebung vertauscht (ein Objekt erhält eingehende Signale die durch Slots verschickt werden).

2.3 Sprachfeatures

Die in dieser Arbeit vorgestellte Implementierung macht verstärkt Gebrauch von den Möglichkeiten, die sich durch Reflection, Annotation und Generics bieten. Dadurch sollen Routine-Aufgaben bei der Erstellung dem Entwickler abgenommen und zentralisiert implementiert werden. Dieser Abschnitt gibt eine Übersicht, welche Sprachfeatures benötigt wurden und sowie Literatur-Verweise, ob sie in den als relevant empfundenen Sprachen zur Verfügung stehen.

2.3.1 Reflection

Reflection ist die Fähigkeit einer Programmiersprache, die Struktur von Klassen zur Laufzeit zu analysieren und Methoden anhand ihres Namens (als String) aufzurufen. Genauer beschreibt Forman die Fähigkeiten wie folgt [6]:

> Reflection is the ability of a running program to examine itself and its software environment, and to change what it does depending on what it finds. To perform this self-examination, a program needs to have a representation of itself. This information we call metadata. In an object-oriented world, metadata is organized into objects, called metaobjects. The runtime self-examination of the metaobjects is called introspection.

Dadurch wird es möglich Mechanismen zu entwickeln, welche gekennzeichnete Methoden aufrufen ohne dass deren genauer Name bei der Kompilierung bekannt ist. Verwendet wird Reflection bei der vorliegenden Implementierung vor allem für den Plug-In Mechanismus und die *Dependency Injection*.

2.3.2 Annotation

Annotation[2] beschreibt die Möglichkeit, einzelne Elemente (z. B. Variablen) um Meta-Information zu ergänzen. Diese kann im Anschluss zur Laufzeit vom System ausgewertet werden. Ein Anwendungsbeispiel aus der vorliegenden Implementierung ist die Kennzeichnung von Membervariablen, die Abhängigkeiten zu Modulen definieren. Durch die Annotation unterscheiden sich die Membervariablen für die Abhängigkeiten von den „regulären“ Membervariablen der Klasse. Die *Dependency Injection* greift nur auf die gekennzeichneten Felder zu.

[2] de.wikipedia.org/wiki/Annotation_(Programmierung)

2.3.3 Verfügbarkeit in den Sprachen

Um das vorliegende Modell gemeinsam mit allen Features der Implementierung möglichst einfach in eine andere Sprache zu übersetzen, ist es von Vorteil, wenn die Ziel-Sprache ebenfalls über Reflection und Möglichkeiten zur Annotation verfügt. Generell stellte sich heraus, dass zwar Reflection in den meisten Sprachen zur Verfügung steht, jedoch Annotation nicht.

Java: Für Java steht Reflection seit dem JDK 1.1 zur Verfügung. Erste Schritte im Umgang mit Reflection-API aus Java bietet [12]. Um alle gebotenen Möglichkeiten voll ausschöpfen zu können, empfiehlt sich die Lektüre von [6].

ECMA-Script: Zu den Dialekten von ECMA-Script zählen unter anderem JavaScript und ActionScript 2. Reflection steht nicht direkt zur Verfügung, dennoch können Instanzen eines Typs anhand des Namens als String erzeugt werden (siehe [5]).

ActionScript 3: Seit der Version 3 steht für die ActionScript ein Class-Objekt zur Verfügung. Dieses bietet jedoch nicht die gleiche Funktionalität wie das Pendant aus Java. Für die Introspektion, vor allem die Abfrage von Superklassen, wird `flash.utils.describeType()` benötigt[3].

C++: Da Reflection standardmäßig in C++ nicht zur Verfügung steht, muss die benötigte Funktionalität entweder selbst implementiert oder über zusätzliche Bibliotheken ergänzt werden. Bibliothek die Reflection zur Verfügung stellen sind die Boost Library[4] und das .Net-Framework 4[5] .Ansätze zu eigenen Implementierungen zeigt [3].

C#: Als Teil des .NET-Frameworks verfügt C# über eine sehr ausgereifte Reflection API. Ebenso ist es in C# möglich Annotations (genauer Attributes) zu verwenden.

Objective-C: Auch für die Entwicklung von Spielen für Endgeräte von *Apple* steht eine ausreichende Reflection-API zur Verfügung. Nähere Details dazu finden sich in der *Objective-C Runtime Reference*.

[3] http://livedocs.adobe.com/flash/9.0/ActionScriptLangRefV3
[4] http://boost-extension.redshoelace.com
[5] http://msdn.microsoft.com/de-de/library/system.reflection.aspx

2.4 Möglichkeiten für Identifikatoren

Um ein Element innerhalb einer Menge eindeutig zu kennzeichnen und dadurch auch die Möglichkeit zu bieten, dieses Element abzufragen, werden Identifikatoren (engl. ID) benötigt. Dabei gibt es einige Kriterien die bei der Wahl nach einem geeigneten System in Betracht zuziehen sind. Das Hauptkriterium ist die Eindeutigkeit. Dabei spielt auch die Größe der Menge eine Rolle, da das gewählte System in der Lage sein muss, so viele unterschiedliche Identifikatoren zu erzeugen, wie die Menge Elemente beinhaltet. Zudem ist die Komplexität beim Vergleichen von Bedeutung. Werden oft Vergleiche mit hoher Komplexität ausgeführt, so kann dies zu einem Performance-Problem werden (z. B. String-Vergleiche). Das letzte wesentliche Kriterium ist die Verständlichkeit (Lesbarkeit) des Identifikators und ob von ihm auf das identifizierte Element geschlossen werden kann. Dies spielt vor allem während der Entwicklung und beim Debuggen eine Rolle.

2.4.1 Zahlenwerte und Enumerationen

Eine erste Möglichkeit für die Realisierung war die schlichte Nummerierung (später über Enumerationen gelöst). Der Umfang der möglichen Identifikatoren ist abhängig vom gewählten Typ, gleich bleibt jedoch die schnelle Vergleichbarkeit. Ebenso bietet dieser Ansatz eine hohe Kompatibilität mit Strings. Der Nachteil dieses Ansatzes ist, dass von dem Wert kein Rückschluss auf das identifizierte Objekt gegeben ist.

2.4.2 String

Strings bieten eine Alternative, die sich vor allem durch die hohe Flexibilität des Inhalts auszeichnet. Dabei ist die Lesbarkeit des Wertes abhängig von der gewählten Implementierung. Ein Nachteil ist der aufwendige Vergleich von Zeichenketten. Dafür gibt es jedoch in manchen Sprachen Mechanismen die dieses Problem ausgleichen (z. B. verwaltet Java einen internen Pool aller Zeichenketten).

Klassen- und Packetnamen

Die Verwendung von Klassen- und Packetnamen bietet unterstützt durch den *Compiler* eine Eindeutigkeit des Wertes. Zusätzlich steht in Verbindung mit Reflection auch eine Möglichkeit zur Verfügung, eine Klasse (genauer das *Class*-Objekt) anhand ihres Namens anzufordern (siehe [12, Seite 1020]). Geeignet ist dieser Ansatz für die Identifikation von Typen von Elementen, jedoch nicht für deren Instanzen.

UUID

Eine besondere Form eines Identifikators, die auch als String eingesetzt wird, ist die UUID. Ähnlich den regulären Zahlenwerten, bietet diese jedoch keinen Rückschluss auf das Element. Leach et al beschreibt eine UUID wie folgt [15]:

> A UUID is 128 bits long, and can guarantee uniqueness across space and time. UUIDs were originally used in the Apollo Network Computing System and later in the Open Software Foundation's (OSF) Distributed Computing Environment (DCE), and then in Microsoft Windows platforms.

In einigen Sprachen stehen eigene Klassen zur Repräsentation einer UUID zur Verfügung, generell reicht jedoch auch die Speicherung als String. Aufwändig ist auch die Erzeugung, da dafür eigene Tools benötigt werden.

2.4.3 Class-Objekte

Eine weitere Möglichkeit zur Realisierung von Identifikatoren ergibt sich durch die Verwendung von Reflection. Krüger beschreibt die Class-Objekte aus Java wie folgt [12]:

> Zu jeder Klasse, die das Laufzeitsystem verwendet, wird während des Ladevorganges ein Klassenobjekt vom Typ `Class` erzeugt. Die Klasse `Class` stellt Methode zur Abfrage von Eigenschaften der Klasse zur Verfügung und erlaubt es, Klassen dynamisch zu laden und Instanzen dynamisch zu erzeugen. Darüber hinaus ist sie der Schlüssel zur Funktionalität der Reflection-APIs.

Da die *Class*-Objekte nur einmal vom System erzeugt werden bietet dieser Ansatz die gewünschte Eindeutigkeit. Erreichbar sind sie direkt aus der Klasse. Da für jede Klasse nur eine Instanz eines *Class*-Objektes angelegt wird, kann diese auch einfach über die Referenz verglichen werden. Ebenso stehen Methoden zur Abfrage von implementierten Interfaces oder der Klassen aus der Vererbungshierarchie zur Verfügung.

Dieser Ansatz bietet zusätzlich den Vorteil, dass kein zusätzlicher Speicher (z. B. für die Speicherung einer UUID als String) verwendet wird, da die Class-Objekte ohnehin bereits zur Verfügung stehen. Der Name der Klasse kann als String ausgegeben werden und umgekehrt kann über diesen, auch wieder das Class-Objekt angefordert werden.

Gleichzeitig weist die Verwendung des Class-Objektes eine hohe Wartungsfreundlichkeit auf. Werden Identifikatoren dazu verwendet, mit einem Wert mehrere Elemente zu identifizieren, muss bei diesem Ansatz der Identifikator nicht separat geändert werden, wenn sich die Implementierung ändert.

2.5 Komponentenbasierte Engines/Frameworks

2.5.1 Elephant (C#)

Das Elephant-Framework bietet eine Komponenten-Architektur für C# in Verbindung mit XNA an. So gibt eine zwei unterschiedliche Arten von Komponenten von denen abgeleitet werden kann. Eine für die Erstellung von Logik und eine für jene Komponenten die gerendert werden müssen. Verglichen mit der Architektur, die in dieser Arbeit vorgestellt wird, sind die Komponenten eher mit Modulen vergleichbar. Jedoch wird das Framework nicht mehr weiter entwickelt (letztes Update 2007). Zum Download bereit steht sie auf `www.codeplexx.com/elephant`.

2.5.2 PushButton (AS3)

Für die Entwicklung von komponentenbasierten Spielen für Flash kann die PushButton Engine genannt werden. Sie wird von den PushButton Labs entwickelt und unter der MIT Lizenz vertrieben. Zum Download erhältlich ist sie auf `pushbuttonengine.com`. Die PushButton Labs wurden von Jeff Tunnell und Rick Overman gegründet, welche beide auf eine langjährige Erfahrung in der Entwicklung von AAA-Spielen zurück blicken können. Da die Engine frei erhältlich ist, erfolgt die Finanzierung durch die damit entwickelten Spiele. Ebenso soll bald der Verkauf[6] von einzelnen Komponenten starten.

2.5.3 Unity3D

Unity3D ist zurzeit als führende Game-Engine auf Basis von Komponenten zu bezeichnen. Dabei hebt sich die Engine besonders durch ihren visuellen Editor und die Unterstützung von zahlreichen Plattformen heraus. Zusätzlich verfügt es über eine sehre leistungsfähige Render-Engine. Es besteht die Möglichkeit, einmal entwickelte Spiele für Windows, Mac, iPhone/iPod/iPad, Android und mittels eigenem Player für das Web anzubieten. Das SDK steht jedem für nicht kommerzielle Zwecke zur Verfügung. Für die kommerzielle Nutzung stehen für jede Plattform Lizenzen zur Verfügung. Da der Quellcode nicht offen zugänglich ist, kann keine genaue Aussage über das Komponenten-Modell getroffen werden.

Unity3D unterstützt die Erstellung von verschachtelten Spielobjekten. Dies ist einerseits hilfreich, das Spielobjekte vordefiniert (als so genannte Prefabs) und wieder verwendet werden können. Andererseits ist dies auch notwendig, da die Komposition eines Spielobjektes nur eine Komponente von einem jeden Typ haben darf (z. B. nur einen Collider). Um diese Limitierung zu umgehen, werden weitere Komponenten vom selben Typ in eines der verschachtelten Spielobjekte ausgelagert.

[6] http://pushbuttonengine.com/faq

2.5.4 Cogaen/CogaenJ

Cogaen (Component-based Game Engine) wird an der FH Oberösterreich, Campus Hagenberg unter der Leitung von Roman Divotkey entwickelt. Unter Teilnahme der Studiengänge Medientechnik und -Design, Interactive Media (früher Digitale Medien) und Mobile Computing wird seit Jahren im *GLab*[7] an komponentenbasierten Architekturen für Game-Engines geforscht. Cogaen zeichnet sich insbesonders dadurch aus, dass es Übersetzungen (teilweise für ältere Versionen) für zahlreiche Sprachen und Endgeräte gibt. So existieren zurzeit Übersetzungen für C++ (Windows, Mac), C# (PC, *Xbox360*), Objective-C (*iPod*, *iPhone*), ActionScript2 (*Flash* für Web-Spiele) und Java (PC, Web-Spiele). Eine Liste von damit entwickelten Projekten findet sich auf `cogaen.org`. Aktuell befindet sich die Version 3 in Entwicklung. Für Cogaen 2 stehen Anbindung für *Ogre3D*, *PhysX* und *FMOD* zur Verfügung. Eine Besonderheit der Architektur sind Abstraktionsschichten für die einzelnen Bereiche (Visual, Physics, Audio), welche es ermöglichen, Anbindungen für neue Teil-Engines zu entwickeln und dabei die bereits entwickelten Komponenten wieder zu verwenden.

Die Übersetzung für Java (CogaenJ auf Basis von Cogaen 2), sowie das für alle Übersetzungen einheitliche Modell mit dem Plug-In Mechanismus, der komponentenbasierten Spielobjekt-Verwaltung und der Kommunikation mittels Nachrichten und Events bildeten die Grundlage für die vorliegende Arbeit. Da die für diese Arbeit erstellte Implementierung (Java Game Components, JGC) nicht mit Cogaen kompatibel ist, versteht sich JGC als Erweiterung des Modells für die optimale Nutzung von Reflection und Java sowie einem eigenen Komponenten-System. Reflection soll dabei Routine-Aufgaben, wie die Bereitstellung von Factory-Klassen, automatisieren und die Entwicklung erleichtern.

ACES

ACES hat seine Ursprünge bei der Übersetzung von Cogaen für die Sprache C#. Die Engine wurde für die Verwendung auf der *XBox360* in Verbindung mit *XNA* erstellt. Neu in der Architektur von *ACES* ist die Einführung von hierarchischen Komponenten. Diese lösen das Problem bei der automatischen Suche nach einer Komponente von einem bestimmten Typ, wenn mehrere Komponenten dieses Typs enthalten sind. Ähnlich *Unity3D* ist es auf diesem Weg möglich, Zweideutigkeiten, bei der automatischen Verknüpfung der Komponenten, über ihre Position in der Hierarchie zu lösen.

[7] http://games.fh-hagenberg.at

2.6 Java

Mit der Plattform `game.dev.java.net` bietet Java eine erste Anlaufstelle für die Suche nach nützlichen Technologien für die Spieleentwicklung. Dennoch sollten die aufgelisteten Projekte kritisch bewertet werden, da einige von ihnen noch nicht ausgereift oder bereits veraltet sind. Ein Überblick über relevante Projekte soll hier nun genannt werden.

2.6.1 JOGL

Die Anbindung für die *Open Graphics Library* wird als JSR 231 (Java Specification Request)[8] entwickelt und soll für alle Java-Applikationen eine einheitliche Schnittstelle bieten. Der Vorteil von JOGL ist, das die damit erstellten Spiele über volle Hardwarebeschleunigung verfügen, unabhängig davon, ob sie als Desktop-Applikation oder als Applet erstellt wurden. Der Nachteil ergibt sich durch die Notwendigkeit der Einbindung der dazu benötigten nativen Bibliotheken. Je nach Zielplattform müssen die korrekten nativen Bibliotheksdateien eingebunden werden, was zum Teil mit einem gewissen Aufwand verbunden ist.

2.6.2 JOAL

Die Anbindung für die *Open Audio Library*[9] ist das Pendant von JOGL für den Audiobereich. Ebenso wie JOGL handelt es sich dabei um eine Low-Level API, welche nur Basisfunktionalitäten zur Verfügung stellt und die Grundlage für umfangreichere Sound-Engines bildet.

2.6.3 Java Native Interface

Mit dem Java Native Interface (JNI) ist es möglich auf die Funktionalität nativer Bibliotheken zuzugreifen. Die so eingebundenen Bibliotheken müssen dazu für die jeweilige Zielplattform kompiliert werden da eine Plattformunabhängigkeit wie ansonsten von Java geboten nicht gewährleistet ist.

Anwendung in Spielen findet JNI bei Anbindungen an verschiedenste Projekte (z. B. für die Anbindung der FMOD Sound-Engine). Eine vollständige Portierung nach Java entfällt und die Anbindungen verfügen über die gleichen Fähigkeiten wie die nativen Hauptprojekte. Einen genauen Überblick über die technischen Möglichkeiten und eine Anleitung zum Umgang mit JNI findet sich in [16].

[8] http://jcp.org/en/jsr/detail?id=231
[9] https://joal.dev.java.net

2.6.4 Java Plug-In Framework

Das Java Plug-In Framework (kurz JPF[10]) bietet die Möglichkeit Plug-Ins zu verwalten und ist im Status der Entwicklung weiter fortgeschritten als jUtils[11]. Im Funktionsumfang enthalten sind Registrierung und Deregistrierung von Plug-Ins sowie die Verwaltung von Abhängigkeiten. Es sind noch weitere Funktionalitäten enthalten, wie die Aktivierung von Plug-Ins zur Laufzeit, welche sich im Spielebereich jedoch nicht notwendig sind. Somit ist das JPF ein guter Startpunkt für die Entwicklung einer Architektur mit Plug-In Mechanismus sollte jedoch im weiteren Verlauf durch eine auf die Bedürfnisse optimierte Implementierung ohne Funktionsüberschuss ausgetauscht werden.

2.6.5 JMonkey Engine

Die JMonkey Engine[12] (JME) ist eine leistungsfähgie 3D-Render-Engine für Java. Ausgelegt für die Spieleentwicklung verfügt es zusätzlich über Funktionalitäten wie Physik und Netzwerk. Entwickelt wird es als Open Source Projekt von einer Community unter der BSD Lizenz. Ein erster Alpha-Release der Version 3 fand am 17. Mai 2010 statt.

Die JME ist zurzeit die am weitesten entwickelte frei erhältliche Game-Engine auf der Basis von Java. Obgleich die JME weithin als reine Java-Engine beschrieben wird, stimmt dies nur zum Teil. Auch die JME basiert auf JOGL und JOAL und benötigt somit die dazugehörigen nativen Bibliotheken. Die Architektur der Engine erlaubt es alternativ andere Anbindungen für spezielle Funktionen zu verwenden. Ein Beispiel dafür ist die Sound-Engine FMOD.

Trotz der Abhängigkeiten nativen Bibliotheken ist es mit der JME möglich nicht nur Spiele für den PC zu entwickeln sondern auch aufwendige Browser-Games umzusetzen. Mit Poisonville entsteht zurzeit beim deutschen Entwickler BigPoint ein außergewöhnliches MMO-Spiel im Stil von Grand Theft Auto. Durch für den Browserspielsektor außergewöhnliche 3D-Grafik hebt es sich deutlich von bisherigen Konkurrenzprodukten wie Runescape ab.

[10] http://jpf.sourceforge.net
[11] https://jutils.dev.java.net
[12] www.jmonkeyengine.com

Kapitel 3

Entwurf

Dieses Kapitel beschreibt die Anforderungen, welchen die Architektur einer Game-Engine gerecht werden muss, sowie das daraus resultierende Modell. Im Weiteren werden die Bestandteile der Architektur sowie deren Zusammenspiel erörtert. Es werden ebenfalls problematische Stellen im Modell der Architektur aufgezeigt. Die einzelnen Probleme, welche sich durch unterschiedliche Implementierungen des Modells ergeben können, werden im anschließenden Kapitel 4 erörtert.

3.1 Problemstellung

Eine Game-Engine ist von der Grundkonzeption nicht auf die Entwicklung eines einzelnen Videospiels ausgelegt, sondern wird bestenfalls für einen einzelnen Titel optimiert. Um eine Vielzahl von Spielen entwickeln zu können, muss die Engine eine große Bandbreite an Funktionalitäten abdecken, die sich jedoch teilweise von Spiel zu Spiel unterscheiden und im Zuge des technologischen Fortschritts immer umfangreicher werden. Das Problem bei einer fest verknüpften Architektur ist, dass in einem Spiel nicht benötigte Funktionalität enthalten ist. Dies lässt den Quellcode unübersichtlich werden, da die beteiligten Programmierer nicht auf Anhieb unterscheiden können, ob die Funktionalität für das Spiel relevant ist oder nicht und so unter Umständen überflüssiger Wartungsaufwand betrieben wird. Ein Beispiel dafür wäre, dass ein reines 2D-Spiel mit einer 3D-Engine entwickelt wird. Obwohl nicht benötigt, wird in einem solchen Szenario ebenfalls Funktionalität für das Laden und Verwalten von 3D-Modellen zur Verfügung gestellt (sofern es sich um eine High-Level-API handelt).

Im schlimmsten Fall können Änderungen an einer Engine für ein neues Spiel mit gleichem *Game Play* mehrere Jahre in Anspruch nehmen. So geschehen bei der von *John Carmack* für *ID* entwickelten Engine für *Quake 3* (siehe [20, Kap. 1]).

3.1.1 Modularität

Ein modularer Aufbau der Engine ermöglicht es den Entwicklern nur jene Funktionalitäten in die Engine einzubinden, welche auch benötigt werden. Dieser reduziert den Aufwand beim anpassen der Engine an ein Spiel. In einem aktuellen Erfahrungsbericht [14] eines deutschen Spieleentwicklers wird der Vorteil wie folgt beschrieben:

> Gerade für Projekte mit kurzer Laufzeit ist das ein großes Plus. Durch die konsequente Umsetzung spezieller Funktionalität über Plug-ins ist es darüber hinaus für Spiele vom Schlage „Ankh – Die verlorenen Schätze“ nicht nötig, Features oder externe Middleware aus dem Kern „herauszuoperieren“.

Dies ermöglicht es schneller mit der Erstellung von für das Spiel relevanten Funktionalitäten zu beginnen und reduziert die Kosten der Vorproduktion. Ebenso verringert sich die Gefahr von Stehzeiten. Die Entwicklung von neuen Modulen kann auf mehrere Programmierer aufgeteilt werden, welche unabhängig voneinander arbeiten können. Ist ein Modul funktionsfähig, wird es in die Engine eingebunden und steht dem Team zur Verfügung. Die Gefahr von beschädigten Builds bei der Integration neuer Features sinkt. Selbst wenn Probleme dabei auftreten, kann das Modul vorübergehend wieder aus dem Spiel genommen werden.

Dem Vorteil der einfacheren Anwendung steht jedoch eine komplexere Entwicklung gegenüber. Im Fall einer fest verknüpften Architektur können konkrete Aussagen getroffen werden, welche Schnittstellen zur Verfügung stehen und wie sich eine Implementierung genau verhält. Beim modularen Aufbau mit austauschbaren Implementierungen können sich im Detail unvorhersehbare Unterschiede ergeben.

3.1.2 Lebenszyklen

Lebenszyklen entstehen durch die internen Abläufe der Engine. Sie definieren die Reihenfolge in denen Objekte erzeugt, konfiguriert, verwendet und wieder zerstört werden. Klar definierte Lebenszyklen vereinfachen den Umgang mit der Engine. Sie bieten auch gute Ansatzpunkte für Erweiterungen und Verbesserungen, da sie Ansatzpunkte für Automatisierungen und Verbesserungsmöglichkeiten in der Struktur darstellen. Durch gute Planung der Lebenszyklen lassen sich Probleme im Zusammenspiel der Elemente bereits vor der Implementierung erkennen. Wesentliche Fragestellung, die sich bei der Planung der Lebenszyklen ergibt, ist ob alle vorbereitenden Arbeitsschritte bereits erledigt wurden wenn ein neuer Abschnitt im Zyklus erreicht wird.

3.1.3 Spielobjekt-Verwaltung

Die Verwaltung der Spielobjekte beinhaltet nicht nur die Basisfunktionalität des Erzeugens und Zerstörens. Viel mehr von Bedeutung in diesem Bereich ist die Art und Weise wie sich die Spielobjekte zusammensetzen. Das gewünschte Modell soll es ermöglichen, dass die einzelnen Funktionalitäten zusammengeführt werden können ohne die Module starr zu verbinden. Änderungen in der Logik oder der Komposition eines Objekts sind im Idealfall einfach zu realisieren und wirken sich nur begrenzt auf die anderen Objekte aus.

Im Hinblick auf die Verwendung eines Editors treten zusätzliche Anforderungen auf. Am Beispiel von *Unity3D* erklärt, soll es ein Editor auf dem aktuellen Stand der Technik, zusätzlich zur reinen Platzierung in der Spielwelt, auch ermöglichen die Objekte selbst zu definieren. Über vorgefertigte Bausteine ermöglicht es *Unity3D* die Objekte per simplen *drag and drop* zu definieren. Die Funktionalitäten eines Objektes können jederzeit erweitert, ausgetauscht oder verändert werden.

3.1.4 Kommunikation

Eine Game-Engine zeichnet sich im Vergleich zu anderen Softwarepaketen dadurch aus, dass die Spielobjekte verstärkt miteinander interagieren müssen. Beispiele für die Inter-Objekt-Kommunikation sind die Synchronisation zwischen zwei Spielobjekten (Logik des Avatars und dem HUD) oder das reagieren auf Ereignisse (der Avatar durchquert einen Lichtschranken und löst dadurch die Explosion mehrerer Bomben aus). Das Kommunikationsmodell muss dahingehend ausgerichtet sein, dass nicht nur der Datenaustausch zwischen einem Sender und einem Empfänger möglich ist, sondern dass auch multiple Empfänger auf einmal bedient werden können. Normalfall der Kommunikation ist der Austausch von Daten um diese weiter zu verarbeiten.

In manchen Fällen jedoch werden keine Daten benötigt und die Kommunikation dient nur zum Auslösen einer Reaktion. Ein solches Szenario beinhaltet keine Notwendigkeit die reagierenden Objekte zu kennen oder zwischen verschiedenen Auslösern zu differenzieren. Die Speicherung der Identifikatoren von Empfängern führt dabei nur zu eigentlich nicht benötigtem Verbrauch von Arbeitsspeicher. Besonders häufig tritt dieses Szenario in der Intra-Objekt-Kommunikation ein. Eine Logik-Komponente wartet darauf, dass ein bestimmtes Verhalten aktiv geschaltet wird, stellt ein valides Beispiel dar. Während der Entwicklung und dem Debuggen kann diese Aktivierung über einen Tastendruck erfolgen, was das Arbeiten erleichtert. Im fertigen Spiel wird die Aktivierung durch eine zweite Logik-Komponente ausgelöst. Das Kommunikationsmodell benötigt eine Möglichkeit, diese Elemente unabhängig vom jeweiligen Empfänger oder Sender miteinander und so einfach wie möglich zu koppeln.

3.2 Überblick über die Architektur

Bevor in den folgenden Abschnitten auf die einzelnen Bestandteile und deren Zusammenspiel eingegangen wird, soll zur besseren Orientierung hier vorab ein kurzer Überblick über die Architektur gegeben werden. Das Modell der hier präsentieren Game-Engine besteht wie in Abbildung 3.1 dargestellt aus drei Bereichen.

Die Modularität wird über den *Plug-In Mechanismus* gewährleistet. Der *Core* verwaltet die zur Verfügung stehenden Plug-Ins. Über ihn können weitere Plug-Ins je nach Bedarf hinzugefügt oder vorhandene entfernt werden. Jedes Plug-In bindet ein Modul an die Engine, welches eine bestimmte Funktionalität zur Verfügung stellt. Die Plug-Ins und Module können von den Entwicklern der Engine selbst sowie von Drittanbietern stammen. Die Applikation (das Spiel) erhält über den *Core* Zugriff auf jedes Modul und somit auf die gesamte vorhandene Funktionalität innerhalb der Engine. Hauptaufgabe einer *Application* ist die Konfiguration der Module.

Die *Spielobjekt-Verwaltung* ist ebenfalls als Modul in die Engine eingebettet. Der *GameObjectManager* ermöglicht das erzeugen, abfragen sowie die Zerstörung von Spielobjekten (implementiert als *GameObject*). Die Komposition der Spielobjekte wird unter Verwendung des komponentenbasierten Ansatzes durchgeführt.

Die *Kommunikation* erfolgt eingebettet in das Komponenten-Modell über Nachrichten (*Messages*). Als Basis dienen dafür die Komponenten, über welche die Nachrichten versandt und empfangen werden können, sowie die *GameObjects* über welche die Nachrichten verteilt und weitergeleitet werden. Der *EventManager* steht als Modul zur Verfügung und dient zum versenden von Events an mehrere zuvor registrierte Benutzer. Für die datenlose Kommunikation, hauptsächlich zwischen Komponenten, wird eine Abwandlung von *Signal and Slot* Systems verwendet, welches z. B. in *QT* verwendet wird.

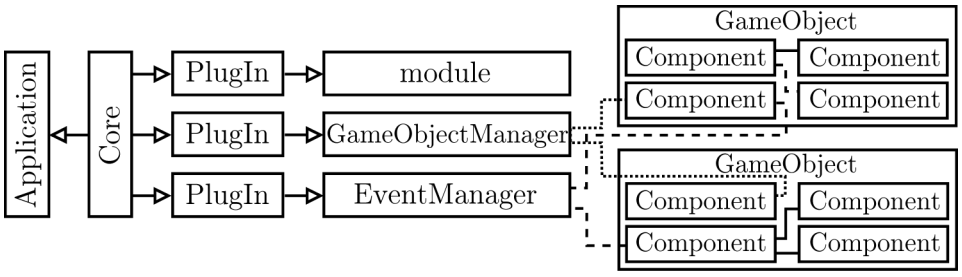


Abbildung 3.1: Darstellung des Grundmodells mit dem Plug-In Mechanismus, den Modulen für die komponentenbasierte Spielobjekt-Verwaltung und dem Versand von Events sowie der eingebetteten Kommunikation mittels Nachrichten (strichliert), Events (gepunktet) und *Signals and Slots* (Linien).

3.3 Plug-In Mechanismus

3.3.1 Anforderungen

Erweiterbarkeit

Die grundlegendste Anforderung an einen Plug-In Mechanismus ist, dass sich über ihn problemlos weitere Funktionalitäten in die Architektur einfügen lassen. Dabei spielt es keine Rolle, ob diese Erweiterungen manuell im Quellcode oder automatisiert über Konfigurationsdateien vorgenommen werden. Letzteres hat den Vorteil, dass es nicht mehr notwendig ist Programmierkenntnisse zu haben und den Nachteil, dass dadurch evtl. auch der Spieler Änderungen vornehmen kann.

Zur Erweiterbarkeit gehört in weiterer Folge auch der Zugriff auf die Funktionalitäten um sie in der Engine zu verwenden. In diesem Rahmen muss der Plug-In Mechanismus Möglichkeiten zur Identifikation von Erweiterungen und deren Abfrage bieten.

Verwaltung von Abhängigkeiten

Bei der Entwicklung einer auf einem Plug-In Mechanismus basierenden Engine werden die einzelnen Teil-Engines weitgehend voneinander unabhängig entwickelt. Dennoch ergeben sich zwischen den Teil-Engines gewöhnlich Abhängigkeiten. Dies ergibt sich daraus, dass bestimmte Funktionalitäten in mehreren Teil-Engines benötigt werden und sich durch deren Auslagerung Verdoppelungen im Quellcode vermeiden lassen. Dies muss jedoch auch von den jeweiligen Implementierungen der Teil-Engines unterstützt werden, was im Fall von vorgefertigten Produkten von Drittanbietern nicht immer der Fall ist.

Im Rahmen der Verwaltung von Abhängigkeiten muss auch eine Möglichkeit zur Bekanntmachung der Abhängigkeiten gegeben sein. Ebenso muss das Verhalten der Engine definiert sein, wenn während der Auflösung Fehler auftreten. Ein vorzeitiges Beenden der Engine ist nicht zwangsläufig notwendig. Ein Spiel auf Basis der Engine könnte auch gespielt werden, wenn das Partikelsystem nicht zur Verfügung steht.

Im Zusammenhang mit den Abhängigkeiten ergibt sich auch der Bedarf der Eruierung einer Startreihenfolge. Dabei ist der Fall, dass eine Teil-Engine bereits konfiguriert bzw. gestartet werden muss bevor die davon abhängigen ihren Dienst aufnehmen kann. Ein verständliches Beispiel dafür findet sich im Bereich der Netzwerkprogrammierung. Dort muss zuerst die Teil-Engine für die Herstellung der Verbindung zum Server und die reine Datenübertragung gestartet werden. Erst danach kann eine darauf aufbauende Teil-Engine sich darüber über den Server synchronisieren.

Schutzmechanismus

Die Zusammenstellung der Funktionalitäten bietet eine einfache Erweiterung der Engine. Zur Laufzeit kann eine Modifikation der Sammlung jedoch aufgrund der Abhängigkeiten zu schwerwiegenden Fehlern und Systemabstürzen führen. Daher ist es notwendig, dass die Veränderung nur dann zugelassen wird, wenn dies möglichst bedenkenlos stattfinden kann. Im Rahmen dieses Schutzmechanismus ist es nicht nur notwendig die Methoden zum hinzufügen und entfernen daraufhin auszurichten, sondern auch etwaige Zugriffe auf die gesamte Sammlung. Ein gern gemachter Fehler ist die Bereitstellung der Sammlung von Funktionalitäten ohne diese Sammlung gegen Modifikationen zu schützen.

Zu Bedenken gilt, dass sobald ein Element der Engine Zugriff auf den Plug-In Mechanismus hat, hat es Zugriff auf die gesamte Funktionalität und kann nach Belieben Konfigurationen vornehmen. In Fällen wo ein Element keinen umfassenden Zugriff haben soll bedeutet dies jedoch auch, dass der Zugriff nicht auf den gesamten Plug-In Mechanismus, sondern nur auf separierte Module gewährt werden darf. Dies kann zwar unter bestimmten Bedingungen vorkommen, ist aber nicht die Regel. Im Normalfall haben die Teil-Engines ohnehin nur Zugriff auf jene Teil-Engines, zu denen Abhängigkeiten bestehen.

Einfache Nutzung

Die einfache Nutzung des Plug-In Mechanismus beinhaltet eine Automatisierung von routinemäßigen Aufgaben sowie die Bereitstellung von Identifikatoren, welche möglichst wenig Wartung und geringe Dokumentation erfordern.

Der erste Ansatzpunkt für eine einfachere Nutzung ist eine mögliche automatisiertere Verteilung von Abhängigkeiten. Dadurch entfällt für die Entwickler von Teil-Engines eine manuelle Anforderung der Abhängigkeiten. Auch wird der Quellcode übersichtlicher, da die Fehlerbehandlung für fehlende Abhängigkeiten bereits zentralisiert stattfindet.

Auch die Festlegung, welche Abhängigkeiten bestehen, soll ohne großen Aufwand möglich sein. Bestehende Systeme erfordern hier einen hohen Dokumentationsaufwand. Dies ergibt sich aus der Wahl von Identifikatoren mit wenig Aussagekraft über die zugehörige Teil-Engine (was das debuggen erschwert) sowie die Wartung der Dokumentation wenn sich die Identifikatoren ändern.

Der Plug-In Mechanismus definiert als zentralstes Element auch die Art und Weise in der die Engine gestartet und beendet wird. Für diese Aufgabe benötigt man in der Regeln einen tieferen Einblick in internen Abläufe um dies korrekt durchführen zu können. Im Sinne einer einfacheren Nutzung sollten diese Aufgaben soweit wie möglich dem Benutzer abgenommen und automatisiert werden.

3.3.2 Bestandteile

Die Architektur des Plug-In Mechanismus besteht im Wesentlichen aus vier Elementen die hier infolge näher erklärt werden. Die Zusammenhänge zwischen den einzelnen Elementen werden in Abbildung 3.2 dargestellt. Der Kern verwaltet eine Sammlung von Plug-Ins. Jedes Plug-In bindet ein Modul an die Engine. Abhängigkeiten zwischen den Modulen werden im Plug-In definiert, vom Kern an die Plug-Ins verteilt und von diesen an die Module weitergereicht. Die Beschreibung des Vorgangs bei der Ermittlung der Startreihenfolge erfolgt in Abschnitt 4.2.4.

Die Funktionen der einzelnen Bestandteile soll nun näher erklärt werden.

Kern

Der Kern (als Teil der Implementierung als *Core* bezeichnet) übernimmt die eigentliche Funktionalität des Plug-In Mechanismus und stellt das zentrale Element dar, über das auf alle Funktionalitäten zugegriffen werden.

Er übernimmt die Verwaltung der Sammlung an Plug-Ins und Modulen. Dazu werden die Plug-Ins vor dem initialisieren registriert. Anschließend wird die Startreihenfolge aufgrund der in den Plug-Ins definierten Abhängigkeiten ermittelt. Während dem hochfahren verteilt er die Referenzen an die Plug-Ins.

Zusätzlich stellt der Kern das Framework für das starten einer Applikation bereit. Sobald eine Applikation über den Kern gestartet wird, übernimmt dieser den Aufruf der entsprechenden Methoden zum richtigen Zeitpunkt.

Der eingebaute Schutzmechanismus gegen unerwünschte Änderungen zu kritischen Zeitpunkten erlaubt ein hinzufügen oder entfernen von Plug-Ins nur vor dem starten und nach dem beenden des Kerns. Dazu erhält die Applikation während ihres Lebenszyklus die Gelegenheit. Sobald die erste Phase der Applikation abgeschlossen ist verwehrt der Kern Änderungen bis zum eintreten in die letzte Phase.

Applikation

Im Rahmen des Modells stellt die Applikation (im Bezug auf die Implementierung das Interface *Application*) die Schnittstelle dar, welche vom Framework des Kerns aufgerufen wird. Die Schnittstelle beinhaltet für jede Phase des Lebenszyklus eine Methode.

Die Aufgaben einer implementierten Applikation umfassen das hinzufügen der benötigten Plug-Ins sowie die Konfiguration der Module. Das Anfordern der zu konfigurierenden Module muss im derzeitigen Stand des Modells manuell im Quellcode vorgenommen werden, kann jedoch nach Erweiterungen über Konfigurationsdateien erfolgen (siehe Abschnitt 3.3.7).

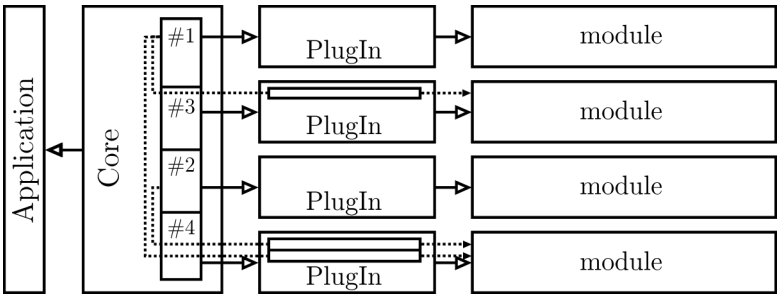


Abbildung 3.2: Darstellung des Modells des Plug-In Mechanismus mit den Bestandteilen *Core*, *Application*, *PlugIn* und den dazugehörigen Modulen sowie den Abhängigkeiten zwischen den Plug-Ins und der daraus resultierenden Startreihenfolge.

Plug-In

Das Plug-In (im Bezug auf die Implementierung das Interface *PlugIn*) stellt die Schnittstelle zwischen dem Kern und einem angebundenen Modul dar. Auch für die Plug-Ins fungiert der Kern als Framework. Dadurch ist es für ein Plug-In nicht notwendig, die API des Kerns zu kennen.

Die sich aus der Trennung von Plug-In und Modul resultierende Abstraktionsschicht erlaubt es auch Module einzubinden, welche nicht gezielt für diese Engine entwickelt wurden. Ein konkretes Modul muss über keine bestimmten Methoden verfügen und keine Interfaces implementieren. Auch muss der Kern nicht die API des Moduls kennen. Alle benötigten Informationen zum Einbinden in die Engine werden vom Plug-In bereit gestellt und die notwendigen Schritte vom Kern aufgerufen. Sollten sich an der Architektur des Plug-In Mechanismus Änderungen ergeben, wirken sich diese nur auf das Plug-In, jedoch nicht auf das Modul, aus.

Modul

Ein Modul ist eine gesamte Teil-Engine, oder eine einzelne Klasse, welche eine bestimmte Funktionalität zur Verfügung stellt. Da die Architektur des Plug-In Mechanismus unabhängig von konkreten Implementierungen entworfen wurde, stellt die hier beschriebene Engine keine Anforderungen an ein konkretes Modul. Dennoch muss berücksichtigt werden, dass ein Plug-In nur die Referenz auf eine einzige Klasse zurück liefern kann. Soll eine Teil-Engine mehrere Klassen zur Verfügung stellen, müssen dafür mehrere Plug-Ins implementiert werden.

Jede in der Engine zur Verfügung stehende Funktionalität, abgesehen jener des Plug-In Mechanismus selbst sowie generellen Hilfsklassen, wird in Form von Modulen implementiert und mittels Plug-Ins eingebunden.

3.3.3 Abgedeckte Funktionalitäten

Hinzufügen und entfernen

Plug-Ins müssen beim Kern registriert werden um ihr zugehöriges Modul innerhalb der Engine zur Verfügung zu stellen. Alleine betrachtet erscheint dieser Schritt trivial. In Verbindung mit Abhängigkeiten wird dies jedoch um einiges komplexer.

Bei der Erstellung eines Spiels werden die benötigten Plug-Ins einmal zu Beginn innerhalb der Applikation hinzugefügt. Während dem Betrieb des Spiels besteht keine Notwendigkeit, weitere Plug-Ins hinzuzufügen, auszutauschen oder zu entfernen. Dieser Umstand erlaubt es, die Zusammenstellung der Plug-Ins und die Verwaltung der Abhängigkeiten in zwei getrennten Schritten zu erledigen. Ansonsten, müssten die Abhängigkeiten bereits während der Registrierung überprüft werden.

Im vorliegenden Modell wird diese Trennung durch eine Kombination der Lebenszyklen sowie dem im Kern integrierten Schutzmechanismus erreicht. Veränderungen an der Sammlung sind nur vor dem starten und nach dem herunterfahren des Kerns erlaubt. Für diese Schritte sind die erste und letzte Phase des Lebenszyklus der Applikation vorgesehen. Dazwischen verhindert der Kern jeden Eingriff. Dies hat den Nachteil, dass ein Plug-In nicht die Fähigkeit hat, weitere Plug-Ins zu registrieren was einen minimal höheren Aufwand bei der Erstellung einer Applikation nach sich zieht.

Anfordern von Plug-Ins und Modulen

Während der Laufzeit wird der Plug-In Mechanismus dazu verwendet um Zugriff auf die Module und deren Funktionalitäten zu erhalten. Nachdem über ein Plug-In der Zugriff auf ein Modul erfolgen kann ist es nicht zwingend notwendig Funktionalität zum Anfordern eines Moduls zur Verfügung zu stellen. Da jedoch der Zugriff auf ein Modul der häufigste Fall ist, ist es auf Grund einer bequemeren Handhabung von Vorteil wenn die Architektur die Funktionalität für beide Fälle zur Verfügung stellt.

Zu berücksichtigen ist, dass nicht nur Zugriff auf einzelne Module notwendig ist, sondern auch auf die gesamte Sammlung benötigt wird. Ein einfaches Anwendungsbeispiel ist die Protokollierung aller hinzugefügten Module beim starten. Eine komplexere Aufgabe, welche diese Funktionalität benötigt, wäre ein Modul, welches eine externe Konfigurationsdatei lädt und die hinzugefügten Module beim starten entsprechend anpasst. Der einzelne Zugriff auf Module ist jedoch der häufigste Fall.

Da die Sammlung von Modulen einer der kritischsten Elemente der Engine ist, muss diese Sammlung vor Modifikationen geschützt werden. Ein Teil des weiter oben beschriebenen Schutzmechanismus des Kerns ist es nur lesenden Zugriff auf die Sammlung zu gewähren. Andernfalls wäre es möglich, den Schutzmechanismus des Kerns zu umgehen.

Identifikation von Plug-Ins und Modulen

Das vorliegende Modell sieht vor, dass jedes Plug-In und Modul eindeutig identifizierbar ist. Dies ist nicht nur für die Abfrage eines bestimmten Moduls zwangsläufig notwendig, sondern auch für die Definition der Abhängigkeiten zwischen den Modulen. Zusätzlich muss bedacht werden, dass im Falle von Abstraktionsschichten in Form von vordefinierten Schnittstellen der Kern auch nach diesen suchen können muss.

Da die Architektur so gestaltet ist, dass an die Module keine Anforderungen gestellt werden, übernimmt die Bereitstellung von Identifikatoren das Plug-In. Der Kern kann bei der Suche nach einem Modul beim Plug-In abfragen welche Schnittstellen angeboten werden. Zusätzlich erhält das Plug-In selbst einen eindeutigen Identifikator. Im Falle von Abstraktionsschichten in Form von vorgegebenen Schnittstellen, muss berücksichtigt werden, dass für ein Modul mehrere Identifikatoren zutreffen.

Die Wahl eines geeigneten Identifikators variiert je nach gewählter Programmiersprache, weshalb hier in der theoretischen Beschreibung des Modells keine eindeutige Empfehlung abgegeben werden kann. Eine Beschreibung der Möglichkeiten befindet sich in Abschnitt 2.4.

Die Auswahlkriterien für ein geeignetes System umfassen neben der Möglichkeit, mehrere Identifikatoren für ein Modul anzubieten, auch die schnelle Vergleichbarkeit der Identifikatoren sowie die Lesbarkeit. Bei komplexen Vergleichen kann es bei häufigen Abfragen von Modulen zu geringen Leistungseinbrüchen führen. In vielen Sprachen trifft dies auf die Vergleiche von Zeichenketten zu. Das Kriterium der Lesbarkeit bezieht sich darauf, ob der Identifikator von einem Menschen lesbar ist und von diesem auf das zugehörige Modul geschlossen werden kann. Ein negatives Beispiel dafür stellt eine *UUID* als Identifikator dar. Zwar ist die Zeichenkette *eff537f0-a61a-11df-981c-0800200c9a66* vom Menschen erkennbar, jedoch lässt sie keinen Rückschluss auf das Modul oder die Schnittstelle zu.

Letzterer Ansatz zieht auch einen hohen Dokumentations- und Wartungsaufwand nach sich. Die *UUID* für jedes Modul und jede Schnittstelle müssen erzeugt und in den Code eingebaut werden. Zusätzlich muss diese in der Dokumentation eigens angeführt werden. Tritt der Fall ein, dass sich eine *UUID* ändert, was tatsächlich sehr selten eintrifft, muss diese Änderung wieder im Quellcode und in der Dokumentation durchgeführt werden.

Im Idealfall werden die Identifikatoren bei Änderungen automatisch von der benutzten IDE[1] mit verwaltet und automatisch in die Dokumentation übernommen.

[1] Integrated Development Enviroment wie z. B. Microsoft Visual Studio oder Eclipse

Verwaltung von Abhängigkeiten

Für die Verwaltung von Abhängigkeiten muss eine Möglichkeit bestehen, diese dem System mitzuteilen. Dabei spielen die im vorherigen Abschnitt erklärten Identifikatoren für die Module eine entscheidende Rolle. Eine Kennzeichnung von mehreren Abhängigkeiten ist notwendig, da ein Modul meist nicht nur eine, sondern häufig mehrere Abhängigkeiten aufweist. Zusätzlich muss das Modell einen Weg beinhalten, auf dem es möglich ist, die Referenzen für die Module zu setzen. Der letzte benötigte Bestandteil übernimmt die Aufgabe, die Definitionen der Abhängigkeiten abzufragen, sie auszuwerten und falls möglich aufzulösen sowie sie anschließend zu verteilen.

Die Auswertung und Auflösung der Abhängigkeiten wird im vorliegenden Modell vom Kern übernommen. Während der Initialisierung einer Applikation und vor dem eigentlichen Starten des Kerns findet die Auflösung statt. Zu diesem Zeitpunkt ist bereits der Schutzmechanismus aktiv, welcher weitere Änderungen an der Sammlung der Module unterbindet.

Als erstes fragt der Kern die Definition der Abhängigkeiten ab. Die Definition der Abhängigkeiten erfolgt im vorliegenden Modell wie in Abbildung 3.2 dargestellt in den Plug-Ins. Dabei kann ein Plug-In eine beliebige Anzahl von Abhängigkeiten definieren.

Der zweite Schritt ist die Ermittlung der Startreihenfolge für die Plug-Ins. Dabei gilt, dass Plug-Ins ohne Abhängigkeiten zuerst und zwar in der Reihenfolge ihrer Registrierung gestartet werden. Für die Module mit Abhängigkeiten gilt, dass ein Modul immer nach den Modulen zu denen Abhängigkeiten bestehen in die Startreihenfolge eingereiht wird. Anschließend erfolgt das Vorbereiten der Plug-Ins gemäß der ermittelten Reihenfolge. Im Fehlerfall macht das Modell keine Unterscheidung ob das Spiel ohne ein fehlendes Plug-In laufen könnte und beendet die Initialisierung der Applikation.

Die Verteilung der Referenzen erfolgt im Zuge der Vorbereitungs-Phase der Plug-Ins. Betritt ein Plug-In, welches Abhängigkeiten definiert, diese Phase, werden noch vor Aufruf der entsprechenden Methode die Referenzen für die Abhängigkeiten gesetzt. Dadurch erhält ein Plug-In nur Zugriff auf Module welche bereits für eine Verwendung vorbereitet sind. Diese können dann anschließend von Plug-In an die Module weitergereicht werden.

Das Setzen der Abhängigkeiten erfolgt im vorliegenden Modell über eine eigene *Setter*-Methode für jede Abhängigkeit. Im Umgang mit Sprachen wo dies nicht möglich ist, kann dies auch über eine gemeinsame Methode erfolgen. Das explizite Löschen der Referenzen ist im Modell nicht vorgesehen, da eine Applikation nach der Verwendung geschlossen wird und die Module nicht mehr benötigt werden. Ein solches Löschen kann jedoch in der letzten Phase des Lebenszyklus der Plug-Ins erfolgen.

3.3.4 Lebenszyklus der Plug-Ins

Die Lebenszyklen der Plug-Ins entstanden aus der Notwendigkeit heraus, das konfigurieren und starten von abhängigen Plug-Ins in mehreren Schritten durchzuführen. Diese Trennung ermöglicht eine verschachtelte Durchführung dieser Aufgaben welche sonst nicht möglich wäre. Die Reihenfolge der einzelnen Phasen innerhalb des Plug-In Mechanismus wird in Abbildung 3.3 dargestellt.

Die *Prepare*- und *Start*-Phase werden für die Trennung beim Starten benötigt. Generell gilt, dass die Plug-Ins in jener Reihenfolge gestartet werden, welche während der Auflösung der Abhängigkeiten ermittelt wurde. Dadurch werden die abhängigen Plug-Ins später gestartet und können bereits auf die Konfigurationen und Dienste ihrer Abhängigkeiten zugreifen.

Das Beenden der Engine umfasst die *Stop*- und *CleanUp*-Phase. Hier werden die Plug-Ins in umgekehrter Reihenfolge aufgerufen. Dies ist z. B. notwendig, damit ein abhängiges Modul beim Beenden noch Daten übertragen kann bevor das Modul die Verbindung zum Server trennt.

Die Aufgaben und Bedingungen zu den einzelnen Phasen werden hier nun näher erklärt.

Prepare: Die *Prepare*-Phase dient für die Allokation von Ressourcen sowie der Konfiguration vor dem eigentlichen Start. Vor Beginn der Phase wurden die Abhängigkeiten aufgelöst und stehen bereits zur Verfügung. Nach Abschluss der Phase muss ein Plug-In vollständig konfiguriert sein. Ein Beispiel ist die Bekanntgabe der IP-Adresse des Servers sowie die Erzeugung des Sockets für eine Netzwerk-Engine.

Start: Diese Phase ist dazu gedacht um den eigentlichen Startvorgang durchzuführen sowie für weitere Konfigurationen die erst nach dem starten eines Plug-Ins möglich sind. Nach Abschluss dieser Phase ist das Plug-In aktiv und voll einsatzfähig und kann seine Dienste innerhalb der Engine zur Verfügung stellen. In Erweiterung des vorherigen Beispiels wird in dieser Phase die Verbindung zum Server aufgebaut.

Stop: Die erste Phase beim Herunterfahren der Engine dient zum korrekten beenden der Dienste. Bei Abhängigkeiten werden zuerst die abhängigen Plug-Ins gestoppt. Im bestehenden Netzwerk-Beispiel wird hier die Verbindung entsprechend dem gewählten Protokoll beendet und die Verbindung zum Server getrennt.

CleanUp: Die letzte Phase des Lebenszyklus bietet dem Plug-In die Möglichkeit Aufräumarbeiten durchzuführen. Dazu gehört die Freigabe von allokierten Hardware-Ressourcen. Als Abschluss des Beispiels wird hier der in der *Prepare*-Phase erzeugte Socket wieder zerstört.

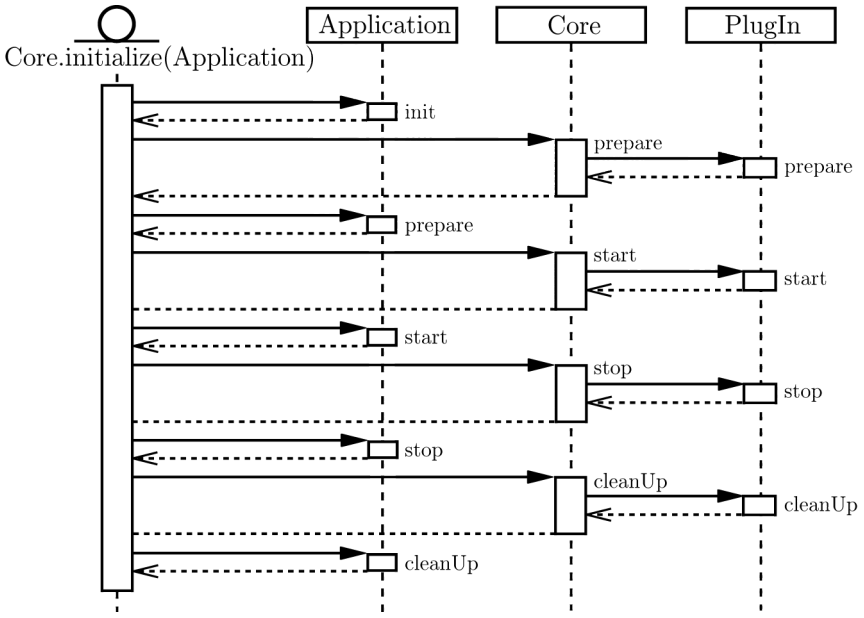


Abbildung 3.3: Ablaufdiagramm der Lebenszyklen beginnend vom initialisieren einer *Application* bis zu ihrem beenden.

3.3.5 Lebenszyklus des Kerns

Das vom Kern gebildete Framework umfasst die gleichen vier Phasen des Lebenszyklus wie das Plug-In. In jeder Phase werden die entsprechenden Funktionen der Plug-Ins aufgerufen. Die zusätzlichen Aufgaben und Bedingungen hier nun kurz im Überblick.

Prepare: Vor Eintritt in diese Phase werden die Abhängigketen aufgelöst und die Startreihenfolg ermittelt. Das setzen der Abhängigkeiten erfolgt vor der Vorbereitung eines jeden Plug-Ins. Ab dem Betreten dieser Phase lässt der Schutzmechanismus kein hinzufügen oder entfernen von Plug-Ins mehr zu.

Start: Der Aufruf der Methoden des Plug-Ins erfolgt in der festgelegten Startreihenfolge.

Stop: Der Aufruf der Methoden des Plug-Ins erfolgt in umgekehrter Reihenfolge.

CleanUp: Nach Abschluss dieser Phase wird der Schutzmechanismus deaktiviert und das entfernen von Plug-Ins möglich.

3.3.6 Lebenszyklus der Applikation

Zu Beginn des Entwurfs des Modells ergab sich die Applikation aus der Erstellung der *main*-Methode. Durch Erfahrungswerte aus anderen Projekten ist bekannt, dass die Anwendung des Kerns als Bibliothek ein Hindernis darstellt. Dabei wird genaueres Wissen über die Reihenfolge der Phasen benötigt. Um die Handhabung zu erleichtern und den Aufruf der Methoden zu automatisieren wurde das Framework des Kerns auf dieses Aufgabengebiet ausgedehnt und das Modell erweitert. Die Phasen werden, wie in Abbildung 3.3 dargestellt, entweder vor oder nach denen des Kerns aktiv.

Ein Überblick über die Aufgaben und Rahmenbedingungen für die einzelnen Phasen ist hier nun gegeben.

Init: Die *Init*-Phase ist die Erweiterung der anderen Lebenszyklen. Sie dient in erster Linie um die gewünschten Plug-Ins zu registrieren und gegebenenfalls erste Konfigurationen vorzunehmen. Dazu wird ein Zugriff auf den Kern benötigt. Dieser wird dem Konstruktor der Applikation übergeben und nicht automatisch vom Kern gesetzt. Nach Abschluss dieser Phase wird der im Kern integrierte Schutzmechanismus aktiv.

Prepare: Der Aufruf erfolgt nach der *Prepare*-Phase der Plug-Ins. Gegebenenfalls wurden von diesen bereits Ressourcen allokiert und können konfiguriert werden. Da das Framework im Fehlerfall abbrechen würde, kann angenommen werden, dass die Vorbereitung der Plug-Ins erfolgreich war. Diese Phase dient um weitere Konfigurationen vorzunehmen zu denen die Plug-Ins bereits vorbereitet sein müssen.

Start: Die *Start*-Phase der Plug-Ins dient zum aktivieren der Plug-Ins. Hier können eine Render-Engine angewiesen werden, das Fenster zu öffnen, Assets geladen und die Spiel-Schleife gestartet werden. Hier beinhaltet die Architektur einen Wartepunkt. Diese Phase darf erst verlassen werden, wenn die Engine heruntergefahren werden soll. Dies ergibt sich je nach Implementierung aus der Natur der Spielschleife von selbst.

Stop: Während dieser Phase können explizite abschließende Anweisungen an die Plug-Ins erfolgen. In der Regel kann jedoch davon ausgegangen werden, dass Plug-Ins für das Beenden ihrer Dienste keine weiteren Eingriffe erfordern.

CleanUp: Die abschließende Phase kann dazu genutzt werden, um die registrierten Plug-Ins wieder vom Kern zu entfernen, da der Schutzmechanismus zu diesem Zeitpunkt bereits deaktiviert ist. Zu diesem haben die Plug-Ins bereits alle Ressourcen wieder freigegeben und ihren Dienst eingestellt.

3.3.7 Erweiterungsmöglichkeiten

Das dargestellte Modell des Plug-In Mechanismus bietet noch Möglichkeiten die Funktionalität zu erweitern und an bestimmte Bedürfnisse anzupassen. Diese wurden zwar im vorliegenden Modell nicht berücksichtigt, da sie für die Erfüllung der Anforderungen nicht zwangsläufig notwendig sind, sollen jedoch an dieser Stelle noch kurz erklärt werden. Die folgenden Erweiterungen beinhalten Versuche der Automatisierung von noch verbleibenden Routine-Aufgaben. Diese zielen darauf ab das Schreiben von Code bei der Erstellung einer eigenen Applikation überflüssig zu machen. Die hier vorgeschlagenen Erweiterungsmöglichkeiten haben sich in der Praxis bewährt oder werden in der jeweils angegebenen weiterführenden Literatur im Detail erklärt.

Automatisiertes Laden von Modulen

Erster Schritt bei der Erstellung einer Applikation mit einer dem Modell entsprechenden Engine ist die Registrierung der vorhandenen Module. Ein Ansatzpunkt zur Erweiterung des Modells ist somit den Vorgang der Registrierung zu automatisieren. Die gewünschten Module können in einem vordefinierten Unterverzeichnis abgelegt (je nach Wahl der Sprache in einem geeigneten Format wie z. B. .dll oder .jar) und von dort aus geladen werden. Alle sich im Verzeichnis befindenden Module werden nach dem Laden beim Kern registriert. Diese Erweiterung bietet zwei Vorteile. Zum einen entfällt die routinemäßige Tätigkeit der manuellen Registrierung. Zum anderen bedeutet dies, dass das Gesamtpaket des Quellcodes auf kleinere Teile aufgeteilt wird. Dadurch können einzelne Module während eines Updates ausgetauscht werden, ohne dass dabei die gesamte Engine erneuert wird.

Externe Konfiguration

Werden die Module automatisch geladen, ist der zweite Schritt einer Erweiterung die Konfiguration ohne Quellcode. Größere Module benötigen Informationen zum korrekten Starten. Ein Beispiel dafür, ist eine Netzwerk-Engine, welche die IP-Adresse des Servers benötigt bevor sie verwendet werden kann. Für die Realisierung einer solchen Konfiguration gibt es unterschiedliche Möglichkeiten.

Eine Möglichkeit besteht darin, dass jedes Modul eine eigene Konfigurationsdatei besitzt. Diese Datei kann in einem für alle Konfigurationsdateien einheitlichen Unterverzeichnis liegen. Dies bietet den Vorteil, dass ein Entwickler der mit der Engine vertraut ist, weiß wo er sie ablegen oder danach suchen soll. Alternativ kann jedes Modul seine Datei dort ablegen wo es will. Der Nachteil den diese Art beinhaltet ist, dass jedes Modul Standard-Aufgaben wie das Auffinden, Laden und Auslesen der Datei selbst übernehmen muss, sofern die ersten beiden Schritte nicht von einem separaten Modul

übernommen werden. Dies führt zu Verdopplungen des Quellcodes, was sich negativ auf dessen Qualität auswirkt und die Wartung erschwert.

Die Module lassen sich auch über eine zentrale Konfigurationsdatei manipulieren. Dieser Ansatz ist jedoch in der Modellierung schwieriger. Ein solches Modul benötigt eine einheitliche Schnittstelle um die Konfigurationen vorzunehmen. Dies bedeutet im Rückschluss jedoch auch, dass eine einheitliche Schnittstelle von jedem Modul, das Möglichkeiten zur Konfiguration bietet, implementiert werden muss.

Eine losere Koppelung zwischen den Modulen, jedoch bei gleicher Funktionalität, ließe sich wieder im Zusammenhang mit *Reflection* und *Annotations* finden. Wird die automatisierte Konfiguration auf dem selben Weg gelöst wie die Deklaration von Abhängigkeiten, kann ein zentrales Modul diese Aufgaben übernehmen. Beide Ansätze beinhalten jedoch auch die Eigenschaft, dass die Dateien vom Benutzer (dem Spieler) manipuliert werden können. Ob sich diese Eingriffsmöglichkeit positiv oder negativ auswirkt, sei der Entscheidung des Entwicklers eines solchen Moduls überlassen.

Dynamisches Plug-In Management

Seltener benötigt wird die dynamische Verwaltung von Plug-Ins. Im vorgestellten Modell wurde das hinzufügen und entfernen von Modulen zur Laufzeit durch den im Kern vorgesehenen Schutzmechanismus verhindert. In Spielen die nur auf dem Endgerät des Spielers laufen findet es in der Regel keine Anwendung. Dennoch findet diese Erweiterung ein Anwendungsgebiet bei Netzwerkspielen. Wird die Engine zum Betrieb von MMO-Spielen (*Massive Multiplayer Online*) mit einer konsistenten Welt verwendet, bei denen die Engine den Server beinhaltet und rund um die Uhr verfügbar sein muss, ist diese Erweiterung notwendig. Dadurch können Updates der Engine vorgenommen werden ohne den Server zu beenden.

Das Problem bei dieser Erweiterung ist es, dass sie nicht nur im Plug-In Mechanismus alleine implementiert werden kann. Sie stellt auch neue Anforderungen an die Architektur der Module. So ist es im vorgestellten Modell bisher nicht notwendig, dass Module Methoden zum sicheren Entfernen von Abhängigkeiten zur Verfügung stellen. Die Abhängigkeiten werden zu Beginn gesetzt und beim beenden der Engine nicht mehr entfernt. Ein so gestaltetes Modul kann sich somit darauf verlassen, dass eine einmal gesetzte Referenz nicht mehr entfernt wird. Modifikationen in Richtung dynamischen Plug-In Management setzen jedoch voraus, dass Module darauf reagieren können die bereits gesetzten Abhängigkeiten, zumindest vorübergehend, wieder zu entfernen.

Abstraktionsschichten

Im vorgestellten Modell arbeitet man direkt mit der API eines jeweiligen Moduls. Die Module sind zwar austauschbar, ziehen dabei jedoch Refactoring-Arbeiten nach sich. Um die Flexibilität zu erhöhen können für die Module Interfaces vorgegeben werden. Wird ein Modul mit der Absicht entwickelt in der Engine verwendet zu werden, kann dieses bereits die Schnittstellen implementieren. Erzeugt man ein Modul als Anbindung an eine bereits bestehende Teil-Engine (Ogre, PhysX, FMOD,...) kann zusätzlich zum Plug-In eine Klasse erzeugt werden, welche die Schnittstelle implementiert.

Die Erstellung von Abstraktionsschichten gestaltet sich in mehrerer Hinsicht als schwierig. Im Falle der Anbindung von Teil-Engines entstehen Methoden, deren einzige Aufgabe es ist, die Aufrufe an andere Methoden weiterzuleiten. In Verbindung mit dynamischer Bindung, welche in Sprachen wie Java nicht frei wählbar ist, erweisen sich Aufrufe von Methoden als rechenintensiv. Je nach gewünschter Leistungsfähigkeit der Engine und verwendeter Sprache kann sich dies bedingt negativ auswirken und sollte daher bereits bei der Planung berücksichtigt werden.

Ein zusätzliches Problem stellt die Definition der Schnittstellen dar. Unterschiedliche Engines für ein bestimmtes Gebiet (z. B. Rendering) bieten unterschiedliche Funktionalität und unterschiedliche Umsetzungen für diese. Am Beispiel des Renderings erklärt bedeutet das, wenn die Schnittstelle für OpenGL entwickelt wurde, erfordert sie bei der Anbindung eines DirectX-Rendermoduls einige Umwege bei der Nachbildung derselben Funktionalität.

Die Unterschiedlichkeit von Modulen wird besonders bei der Konfiguration vor ihrer Verwendung auffällig. Obwohl die Basisfunktionalitäten meist gleich sind (beim Rendering z. B. die Fenstergröße) werden bei unterschiedlichen Funktionalitäten zwangsläufig auch unterschiedliche Einstellungsmöglichkeiten zur Verfügung gestellt. Diese auf eine gemeinsame Schnittstelle zu reduzieren wäre zwar dem Wunsch nach Abstraktion dienlich, erweist sich jedoch als wenig sinnvoll. Mit der auf Seite 32 beschriebenen externen Konfiguration entfällt auch dieses Problem. Sollen die Module im Quellcode konfigurierbar sein, ist es von Vorteil in der Entwicklung den Kompromiss einzugehen, dass wiederum mit der konkreten Schnittstelle gearbeitet wird.

Solche Abstraktionsschichten wurden in *Cogaen* konsequent umgesetzt (siehe Abschnitt 2.5.4). Durch die Abstraktionen mussten die Komponenten nur einmal implementiert werden und standen für alle Anbindungen zur Verfügung. Dies reduzierte wesentlich den Aufwand bei der Erstellung neuer Anbindungen, da nur die bestehenden Schnittstellen implementiert werden mussten und die bestehenden Komponenten wiederverwendet werden konnten.

3.4 Spielobjekt-Verwaltung

3.4.1 Anforderungen

Komposition der Spielobjekte

Zu den allgemeinen Aufgaben zählt die Erzeugung und Zerstörung der Spielobjekte. Das Hauptaugenmerk liegt dabei auf der Art und Weise wie diese zusammengestellt werden. Die Hauptkriterien für die Wahl eines geeigneten Systems sind eine lose Koppelung der notwendigen Module, ein möglichst hoher Grad der Wiederverwendbarkeit sowie die Flexibilität bei Änderungen. Eine Beschreibung der üblichen Ansätze findet sich in Abschnitt 2.1.3.

Für das vorliegende Modell wurde die Komposition mittels Komponenten gewählt, da diese den Kriterien am ehesten entsprach. Eine auf dieses System bezogene Anforderung erscheint bei der Erstellung komplexer Spielobjekte, welche es erfordern verschachtelte Spielobjekte erstellen zu können.

Zugriff auf die Spielobjekte

Im Rahmen der Verwaltung müssen die einzelnen Spielobjekte innerhalb der Engine aufrufbar sein. Die Spielobjekt-Verwaltung muss entsprechende Methoden zur Verfügung stellen.

Benötigt wird dies zum Beispiel für die Inter-Objekt-Kommunikation. Werden zwischen zwei Spielobjekten Daten über Nachrichten ausgetauscht, muss der Zugriff auf den Empfänger möglich sein. Ein weiteres Beispiel findet sich bei der Verwendung einer Skript-Engine. In vielen Fällen ist es möglich, bestimmte Spielobjekte direkt über ihren Namen anzusprechen. Dabei bezieht die Skript-Engine im Vorfeld die konkrete Instanz des Spielobjektes über ihren Identifikator von der Spielobjekt-Verwaltung.

Verknüpfung der Komponenten

Eine der Routinetätigkeiten bei der Erstellung von Objekten ist die Verknüpfung der Komponenten zwischen denen Abhängigkeiten bestehen. Eine automatisierte Verknüpfung kann diese Schritte erleichtern oder ganz erledigen. Voraussetzung dafür ist, dass die Komponenten innerhalb eines Spielobjektes eindeutig identifizierbar sind. Einfachster Ansatz hierfür ist, von jedem Typ nur eine Komponente innerhalb eines Spielobjektes zu erlauben. In der Engine *Unity3D* wird das Problem über diese Limitierung gelöst. Diese Limitierung erschwert jedoch die Erstellung von komplexen Spielobjekten enorm. Berücksichtigt man zusätzlich die von modernen Editoren gebotenen Möglichkeiten kann abgewogen werden, ob dieser Schritt automatisiert werden sollte, wenn dadurch Limitierungen entstehen.

Lebenszyklus

Der Lebenszyklus ist insofern von Bedeutung, da er alle notwendigen Aufgaben bei der Entwicklung von Komponenten abdecken muss. Die Problematik dabei ist, dass ähnlich dem Lebenszyklus der Plug-Ins vor dem Betreten eines Abschnittes bestimmte Voraussetzungen erfüllt sein müssen. Eine davon ist, dass die Komponente bereits bei der Initialisierung Zugriff auf den Kern und somit auf die Module haben muss. Die Module ihrerseits müssen zu diesem Zeitpunkt bereits gestartet und einsatzbereit sein. Dies ist notwendig, damit die Komponenten bereits bei er Initialisierung auf die Funktionalität ihrer Module zugreifen können.

Andererseits muss klar definiert werden, in welcher Reihenfolge die Komponenten initialisiert werden. Dadurch kann ein Entwickler Probleme bei Abhängigkeiten zwischen den Komponenten eines Spielobjektes besser vermeiden.

Tagging

Im Zuge der Identifikation von Komponenten innerhalb eines Spielobjektes, ist ein verbreiteter Ansatz das sogenannte *taggen*. Dazu kann einer Komponente ein Name gegeben werden. Dieser wird dementsprechend als Zeichenkette umgesetzt. Eine so gekennzeichnete Komponente kann anschließend über diesen Namen abgerufen und angesprochen werden. Je nach Implementierung kann dies jedoch zu unnötigem Speicherverbrauch führen. Dies ist der Fall wenn die Komponente den Namen speichert. Wird nun für eine Komponenten kein Tag benötigt wird eine leere Zeichenkette gespeichert. Dieser Ansatz hat jedoch auch den Nachteil, dass eine Komponente nur unter einem Namen gefunden werden kann, die Definition mehrerer Tags ist nicht möglich.

Identifikation von Spielobjekten

Im Zuge der Identifikation von Spielobjekten muss sowohl ein jeder Typ der Spielobjekte als auch eine jede Instanz eindeutig identifizierbar sein. Ein Spielobjekt benötigt somit zwei Identifikatoren.

Der Identifikator des Typs spielt im vorliegenden Modell bei der Erzeugung von Spielobjekten eine Rolle. Spielobjekte werden über den *GameObjectManager* unter Angabe ihres Typs erzeugt.

Die Identifikation der Instanzen muss zwei Anforderungen gleichzeitig gerecht werden. Zum einen muss es für einen Level Designer möglich sein bestimmten Objekten gezielt einen lesbaren Namen geben zu können. Diese Namen treten meist bei speziellen Spielobjekten auf, welche anschließend in einem Skript für die Logik aufgerufen werden. Zum anderen muss es möglich sein, für die häufige Erzeugung von Spielobjekten automatisch einen Namen zu vergeben.

3.4.2 Probleme

Doppelte Datenspeicherung

Die doppelte Speicherung von Daten tritt im Zusammenhang mit zwei unterschiedlichen Ursachen auf. Die erste davon ist ein unvorteilhaft gestalteter Lebenszyklus. Bei der Initialisierung von Komponenten werden ihnen bei Bedarf Werte übergeben, die an ihr zugehöriges Modul weitergeleitet werden müssen. Steht dieses Modul zum Zeitpunkt der Initialisierung noch nicht zur Verfügung, müssen die Werte zwischengespeichert werden. Nach der Weitergabe bleiben die Werte in den dafür angelegten Variablen erhalten.

Die zweite Ursache liegt in der Notwendigkeit, dass dieselben Daten in unterschiedlichen Komponenten gespeichert werden müssen. Einfachstes Beispiel hierfür ist die Logik-Komponente des Avatars, welche die noch vorhandenen Lebenspunkte verwaltet, und die *Visual*-Komponente des *HUD*, welche diese anzeigt. Da beide Komponenten voneinander unabhängig sind, werden die Daten über vorhandene Kommunikationskanäle (z. B. Messages) synchronisiert, aber doppelt gespeichert.

In beiden Fällen entsteht ein zusätzlicher Verbrauch von Speicher. Dieser ist im zweiten Beispiel noch vernachlässigbar. Tritt dieser Aufgrund eines schlechten Lebenszyklus bei zu vielen Komponenten auf, kann dies zu einem ernsten Problem werden.

Multiple Berechnungen

Ein Problem, welches besonders im Zusammenspiel mehrerer Teil-Engines von Drittanbietern auftritt, ist die Existenz von unterschiedlichen Klassen mit derselben Funktion. Viele Teil-Engines verfügen über ihre eigenen Implementierungen für mathematische Klassen und verlangen diese in ihrer API. Ein typisches Beispiel dafür sind Matrizen zur Speicherung der Position. Verwendet man eine Grafik- und eine Physik-Engine mit unterschiedlichen Implementierungen für die Matrix muss ein Spielobjekt beide zur Verfügung stellen können.

Ungünstigste Lösung ist eine Klasse zu entwickeln, welche eine Implementierung beinhaltet und auf Anfrage eine Instanz der zweiten erzeugt und zurück liefert. Da diese Abfragen innerhalb eines Frames mehrmals stattfinden können, würde sich dies unvorteilhaft auf die Performance auswirken.

Ein ähnlicher Ansatz ist Implementierung einer Klasse, welche sich bei der Abfrage zuerst synchronisiert. Wiederum besteht hier das Problem, dass die Abfragen mehrmals pro Frame erfolgen können. In weiterer Folge bedeutet dies zwar, dass die Initialisierung einer neuen Instanz entfällt, aber eine Synchronisation durchgeführt wird, unabhängig davon ob diese notwendig ist oder nicht.

Das Modell muss somit Vorkehrungen treffen, dass sich im Falle solcher zusammenhängender Werte Synchronisationen nur bei Änderungen ergeben.

3.4.3 Bestandteile

Das Modell der Spielobjekt-Verwaltung definiert wie in Abbildung 3.4 dargestellt fünf Elemente. Das Kernstück stellt das Modul des *GameObjectManager* dar, welches sich als Plug-In in die Engine integriert. Ebenso werden das *GameObject* und die *Components* definiert, welche für die Komposition der Spielobjekte verwendet werden. Dabei können vordefinierte *GameObjects*, mit *Components* gleichwertig, der Komposition hinzugefügt werden. Besteht der Bedarf einer Synchronisation bei Änderungen können die Komponenten über *ComponentChangedListener* verbunden werden.

GameObjectManager

Die Spielobjekt-Verwaltung wird gemäß dem Modell aus Abschnitt 3.3 als Modul modelliert und als Plug-In in die Engine integriert. Dadurch ist man nicht zwangsläufig an die Verwendung einer konkreten Implementierung gebunden. Bei der Planung eines Austauschs für ein bestehendes Projekt sollte jedoch berücksichtigt werden, dass dies eine komplette Neugestaltung der bereits implementierten Spielobjekte zur Folge hat.

Der *GameObjectManager* stellt die notwendige Funktionalität zur Verfügung. Diese werden im Detail in Abschnitt 3.4.4 beschrieben. Im Weiteren bietet der *GameObjectManager* als einziger die Möglichkeit Spielobjekte im korrekten Ablauf ihres Lebenszyklus zu erzeugen. Dies bedeutet, dass jedes Element, welches Spielobjekte erzeugen oder zerstören will, Zugriff auf das Modul benötigt. Im vorliegenden Modell ergibt sich dies aus dem Umstand dass die Logik ebenfalls über Komponenten entwickelt wird und Komponenten automatisch Zugriff auf den *GameObjectManager* haben. Im Falle der Erweiterung durch eine Skript-Engine muss diese entsprechende Methoden bereitstellen.

Der *GameObjectManager* verwaltet alle erstellten Spielobjekte. Im Gegensatz dazu werden die Komponenten nur in den umgebenden Spielobjekten und gegebenenfalls in den dazugehörigen Modulen verwaltet. In der Regel ist es von Vorteil mehrere Sammlungen zu erstellen. Eine dient für einen schnellen Zugriff über die Identifikatoren auf das jeweilige Spielobjekt und eine weitere beinhaltet die Spielobjekte in der Reihenfolge ihrer Erzeugung. Letzteres kann beim Laden von Spielständen notwendig sein, wenn zwischen mehreren Objekten Abhängigkeiten bestehen, welche sich nur durch die richtige Reihenfolge ihrer Erzeugung auflösen lassen.

Im Falle von untergeordneten Spielobjekten (in Abbildung 3.4 das *GameObject C*) wird dies vom *GameObjectManager* gleichwertig behandelt. Dies ergibt später im Modell der Kommunikation den Vorteil, dass jedes untergeordnete Spielobjekt ebenfalls als Empfänger dienen kann.

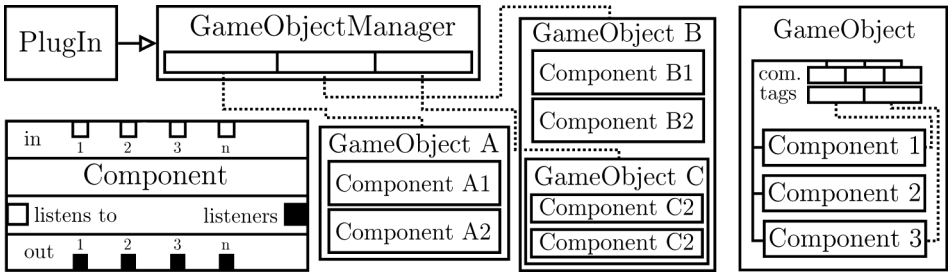


Abbildung 3.4: Darstellung des Modells der Spielobjekt-Verwaltung. Der *GameObjectManager* ist als Modul entworfen. Er verwaltet die vereinfacht dargestellten Spielobjekte *GameObject A*, *B* und *C*. *GameObject A* und *B* verfügen über je zwei Komponenten wobei *B* zusätzlich noch das untergeordnete *GameObject C* beinhaltet. *Component* und *GameObject* bilden vollständige Darstellungen des Modells der entsprechenden Bestandteile.

Spielobjekt

Das Spielobjekt (bezogen auf die Implementierung als *GameObject* bezeichnet) ist der Container für die Komponenten. Aufgabengebiet des Spielobjekts ist die Verwaltung der Komponenten. Dazu beinhaltet es eine Sammlung aller Komponenten sowie eine weitere um den Zugriff über *Tags* zu erlauben. Das in Abbildung 3.4 dargestellte *GameObject* beinhaltet in seiner Sammlung drei Komponenten wobei *Component 1* und *Component 3* über *Tags* erreichbar sind.

Das Modell sieht vor, dass für die Erzeugung von Spielobjekten im Quellcode jedes Spielobjekt eine eigene Klasse erhält, welche von *GameObject* abgeleitet ist. Dies ist auch für die automatisierte Erzeugung der Identifikatoren von Vorteil, da diese sich aus dem Namen des Typs sowie einer fortlaufenden Nummerierung zusammen setzen.

Auch in Hinsicht auf eine potenzielle Erweiterung wie die Erzeugung von Spielobjekten über Konfigurationsdateien ergeben sich durch diesen Ansatz Vorteile. Am Beispiel von XML-Dateien, kann eine Klasse *XMLGameObject* abgeleitet werden, welche das XML als String entgegen nimmt die weitere Erzeugung durchführt. Weitere Veränderungen ergeben sich am Modell durch diesen Ansatz keine. Es muss lediglich darauf geachtet werden, dass bei Bedarf die automatische Erzeugung der Identifikatoren überschrieben wird.

Ebenso ermöglicht wird der gleichzeitige Betrieb von mehreren Systemen zur Erzeugung von Spielobjekten wie z. B. den erwähnten XML-Dateien oder binär codierten Dateien. Dies bietet den Vorteil, dass Objekte, welche noch nicht vollständig implementiert sind, über noch konfigurierbare Dateien geladen werden können. Für die Auslieferung an einen Kunden können dann Binär-Dateien verwendet werden, welche sich nur schwer verändern lassen.

Komponente

Die Komponente (bezogen auf die Implementierung als *Component* bezeichnet) übernimmt die Bereitstellung der Funktionalitäten der dazugehörigen Module. Die Komposition der Spielobjekte erfolgt über das zusammenfügen beliebiger Komponenten. Das in Abbildung 3.4 dargestellte Modell der Komponente verfügt dabei über vier Eingänge (*in*) sowie vier Ausgänge (*out*). Diese stellen Abhängigkeiten zu anderen Komponenten oder deren Eigenschaften dar und stellt selbst vier Eigenschaften zur Verfügung.

Um Synchronisationen zwischen den Komponenten zu ermöglichen, steht zusätzlich ein *Listener* zur Verfügung. Eine jede Komponente implementiert den *ComponentChangedListener* und kann somit nicht nur verständigen sondern auch über Änderungen verständigt werden.

Im Zuge der Entwicklung der unterschiedlichen Module ist es notwendig, dass diese auch Komponenten für ihre Funktionalität bereit stellen. Dadurch entsteht zwangsläufig eine Bindung an das Komponenten-Modell. Damit diese Bindung nicht zu starr wird und ein Austausch ohne große Refactoring-Arbeiten unmöglich wird, empfiehlt sich die Verwendung von Interfaces. Die Module stellen für die von ihnen benötigten Elemente Interfaces zur Verfügung, welche von den Komponenten implementiert werden. Dadurch können die Module mit anderen Implementierungen der Interfaces auch ohne das Komponenten-Modell benutzt werden.

ComponentChangedListener

Um die Verständigung von Komponenten über Änderungen am Zustand zu ermöglichen und im Zuge dessen multiple Berechnungen zu vermeiden (siehe Abschnitt 3.4.2), steht ein Listener zu Verfügung. Dies ermöglicht es jedem Element nach der Registrierung bei einer Komponente über eine Veränderung dessen Zustand informiert zu werden. Der häufigste Fall ist dabei die Verständigung über Zustandsänderungen von Komponenten zu denen Abhängigkeiten bestehen und die über die eingehenden Referenzen verbunden sind. Wann die Listener verständigt werden, muss bei der Implementierung einer konkreten Komponente berücksichtigt werden und kann nicht generell festgelegt werden.

Eine Möglichkeit zur Unterscheidung zwischen veränderbaren Teilen ist nicht vorgesehen. Unterschiedliche Listener die auf Veränderungen unterschiedlicher Variablen reagieren sollen, obliegt es selbst, eine Unterscheidung vorzunehmen. Im Falle mehrerer veränderbarer Werte kann es somit durchaus vorkommen, dass die Listener mehrmals verständigt werden. Auch dieser Umstand sollte bei der Verwendung berücksichtigt werden.

3.4.4 Abgedeckte Funktionalitäten

Erzeugung von Spielobjekten

Die Erzeugung von Spielobjekten erfolgt über den *GameObjectManager* und nicht über den Aufruf deren Konstruktoren. Dazu übergibt man dem Manager den Identifikator des gewünschten Typs. Dieser erzeugt eine neue Instanz und liefert diese zurück. Dieser Ansatz erlaubt es Teile des Erstellungs-Prozesses zu automatisieren und es wird sichergestellt, dass die Vorbedingungen für die erste Phase des Lebenszyklus gegeben sind. So wird nach der Erzeugung der Instanz des Spielobjekts diese automatisch beim Manager registriert. Somit erhält ein Spielobjekt bereits vor der eigentlichen Initialisierung während der *Init*-Phase Zugriff auf den Kern und somit auch auf die Module.

Zusätzlich wird zu diesem Zeitpunkt bereits ein Identifikator erzeugt und im Spielobjekt gesetzt. Soll an Stelle des automatisch generierten Identifikator ein speziellerer verwendet werden, kann diese dem *GameObjectManager* bereits mitgegeben werden. Somit ist sichergestellt, dass ein Spielobjekt gleich nach dessen Erzeugung über seinen Identifikator aufgerufen werden kann. Die Erzeugung von untergeordneten Spielobjekten kann während der *Init*-Phase gemeinsam mit den Komponenten stattfinden.

Der weitere Ablauf des Lebenszyklus, mit Ausnahme der Zerstörung, wird vom Spielobjekt selbst übernommen. Wünschenswert wäre es auch diese Schritte vom Manager erledigen zu lassen. Dazu wird jedoch die in Abschnitt 3.4.7 erklärte vereinheitliche *Init*-Methode benötigt. Zum derzeitigen Stand des Modells beinhaltet jedes Spielobjekt eine *Init*-Methode mit jeweils unterschiedlichen Parametern welche in weiterer Folge an die beinhalteten Komponenten weitergeleitet werden. Die Verknüpfung der Komponenten erfolgt direkt nach Abschluss der Initialisierung des Spielobjektes.

Zerstörung von Spielobjekten

Die Zerstörung von Spielobjekten erfolgt über den *GameObjectManager* unter Angabe einer Referenz auf das Spielobjekt oder über den Identifikator. Die Angabe der Referenz ist vorzuziehen, wenn dem Element (z. B. die Logik-Komponente), welches die Zerstörung auslöst, die Referenz schon zur Verfügung steht, da dadurch das Objekt nicht erst gesucht werden muss. Der zweite Weg ist jedoch der häufigste, da die Komponenten meist nur den Identifikator speichern.

Bei der Zerstörung wird das Spielobjekt vom Manager entfernt und die *Destroy*-Phase des Lebenszyklus eingeleitet. Das Entfernen von Referenzen in etwaigen Logik-Komponenten muss vom Entwickler anderweitig gelöst werden (in der Regel über einen *Destroy-Event*). Die Loslösung der Komponenten von ihren zugehörigen Modulen obliegt den Komponenten selbst.

Generierung von Identifikatoren

Die automatische Generierung von Identifikatoren ist Teil der Erzeugung von Spielobjekten im *GameObjectManager*. Damit anstelle der automatisch erzeugten Identifikatoren andere verwendet werden können, besteht die Möglichkeit den Gewählten dem Manager mitzugeben. Als Typ wurde ein String gewählt und der Wert setzt sich aus dem Namen des Typs des Spielobjektes und einer fortlaufenden Nummerierung, welche mit einem #-Zeichen getrennt sind, zusammen (z. B. *Player#0* für die erste Instanz des Typs *Player*).

Taggen von Komponenten

Die Verwaltung von Tags übernimmt im erstellten Modell das Spielobjekt. Dieses speichert dazu die zusammengehörenden Paare von Tag und Komponente in einer eigenen Sammlung. Würden die Tags direkt in den Komponenten gespeichert werden, würde eine jede Komponente dafür eine Member-Variable benötigen. Wird einer Komponente kein Tag zugewiesen, würde die Referenz jedoch immer noch Speicherplatz erfordern. Durch die Verwaltung im Spielobjekt ist der Speicherverbrauch bei Kompositionen ohne Tags reduziert, da er von der Anzahl der Komponenten unabhängig ist.

Die Zuweisung der Tags erfolgt während der *Init*-Phase der Spielobjekte. Es ist jedoch auch möglich, dass eine Komponente sich selbst einen Tag zuweist. Im Falle des Bedarfs von vordefinierten Tags würde dies die Implementierung erleichtern, da sie nicht jedes Mal vom Entwickler gesetzt werden müssen.

Auflösung der Abhängigkeiten

Die Verknüpfung zwischen den Komponenten erfolgt in Anschluss an die Initialisierung während der *Link*-Phase der Spielobjekte. Im Gegensatz zu anderen bestehenden Modellen erfolgt diese nicht automatisiert. Die Nachteile in Bezug auf eine Limitierung der Kompositionen (z. B. nur eine Komponente eines jeden Typs pro Spielobjekt), welche sich aus einer Automatisierung ergeben, führten zu dieser Entscheidung. Die Verknüpfung wird während der Erstellung der Spielobjekte vom Entwickler manuell im Quellcode vorgenommen. Auch in Aussicht auf einen Editor zieht dies, unter Berücksichtigung der Möglichkeiten moderner Tools, keinen Nachteil nach sich. Die vereinfachte Erstellung komplexer Kompositionen überwiegt den zusätzlichen Aufwand für die manuelle Verknüpfung. Die Verknüpfung der Komponenten umfasst sowohl das Setzen der richtigen Referenzen zwischen den Komponenten als auch die Registrierung der gewünschten Listener.

3.4.5 Lebenszyklus der Komponenten

Der Lebenszylus der Komponenten beinhalten drei Phasen, welche die Initialsierung, die Verknüpfung zwischen den Komponenten und die Zerstörung kontrollieren.

Ähnlich der Erzeugung eines Spielobjektes wird eine Komponente nicht über ihren Konstruktor, sondern über das *GameObject* unter Angabe des gewünschten Typs erzeugt. Dieser Schritt erlaubt es, bereits vor der eigentlichen Initialisierung die Verbindung zum Kern herzustellen. Das in Abschnitt 3.4.2 erklärte Problem der doppelten Datenspeicherung entfällt zum Teil, da die Initialisierungswerte sofort an die zugehörigen Module weitergereicht werden können. Zusätzlich lässt sich auf diesem Weg, die Erzeugung und Verwendung von Komponenten auf die Klasse *GameObject* limitieren und die Komponenten von der restlichen Engine verbergen.

Die Aufgaben der einzelnen Phasen des Lebenszyklus sowie zu beachtende Rahmenbedingungen sind wie folgt:

Init: Während der *Init*-Phase erhält eine Komponente die benötigten Werte für ihre Konfiguration. Da zu diesem Zeitpunkt bereits Zugriff auf den Kern und die Module besteht, kann sich eine Komponente während dieser Phase bei ihrem Modul registrieren. Dafür wird vom zugehörigen GameObject der Kern angefordert und über diesen das Modul bezogen. Ab diesem Zeitpunkt, arbeitet ein Modul mit den Informationen der Komponente. Am Beispiel des Renderings, bedeutet dies, dass die Komponente beim nächsten Render-Vorgang gerendert wird. Zu beachten gilt, dass eine Komponente zu diesem Zeitpunkt alle Eigenschaften zur Verfügung stellen muss, welche von anderen Komponenten als Abhängigkeiten verwendet werden können.

Linked: Beim Eintritt in die *Linked*-Phase beinhaltet das Spielobjekt bereits alle vollständig initialisierten Komponenten. Ebenfalls wurden vom Spielobjekt auch bereits alle Abhängigkeiten zwischen den Komponenten aufgelöst. Unter Umständen können sich Komponenten erst zu diesem Zeitpunkt bei ihren Modulen registrieren, sollte dies nur mit aufgelösten Abhängigkeiten möglich sein.

Typische Aufgaben dieser Phase sind die gesetzten Abhängigkeiten auszuwerten, deren Informationen an das Modul weiter zu reichen sowie Berechnungen mit den Werten der Abhängigkeiten durchzuführen.

Destroy: Bevor ein Spielobjekt vollends zerstört wird, wird für alle ihre Komponenten die *Destroy*-Phase eingeleitet. Dies gibt der Komponente die Möglichkeit, sich beim zugehörigen Modul wieder abzumelden. Zusätzlich sollten alle Referenzen zu Abhängigkeiten gelöscht werden.

3.4.6 Lebenszyklus der Spielobjekte

Der Lebenszyklus der Spielobjekte beinhaltet, gegeben durch die Architektur des Komponenten-Modells, auch jenen der Komponenten. Dies ist notwendig, da die Möglichkeit besteht, ein Spielobjekt einem anderen als Komponente unterzuordnen. Die drei Phasen der Komponenten decken sich mit denen der Spielobjekte. Zusätzlich steht im Spielobjekt die *Link*-Phase zur Verfügung.

Welche Bedingungen für die einzelnen Phasen zutreffen und deren Aufgaben sind wie folgt:

Init: Diese Phase dient dazu, die gewünschten Komponenten zu erstellen, dem Spielobjekt hinzuzufügen und zu initialisieren. Dafür verfügt das Spielobjekt bereits über einen Zugriff zum Kern, was auch für die Initialisierung der Komponenten notwendig ist. Untergeordnete Spielobjekte werden gemeinsam mit den Komponenten an dieser Stelle erzeugt. Nach Abschluss dieser Phase muss ein Spielobjekt alle Komponenten bzw. Spielobjekt beinhalten.

Link: Während der *Link*-Phase erfolgt die Auflösung der Abhängigkeiten zwischen den Komponenten. Im vorliegenden Modell wird diese vom Ersteller des Spielobjekts manuell durchgeführt(siehe Abschnitt 3.4.4). Im Anschluss an diese Phase wird für jede Komponente sowie jedes untergeordnete Spielobjekt die *Linked*-Phase eingeleitet. Nach Abschluss dieser beiden Phasen ist ein Spielobjekt vollständig zusammengesetzt und einsatzbereit.

Linked: Die *Linked*-Phase stammt aus dem Lebenszyklus der Komponenten und wird bei normalen Spielobjekten in der Regel ignoriert. Verwendung findet sie bei untergeordneten Spielobjekten welche Abhängigkeiten aufweisen. Gemäß der entsprechenden Phase der Komponenten kann hier die Auswertung der Abhängigkeiten erfolgen.

Destroy: Bei der Zerstörung eines Spielobjekts werden zuerst alle Komponenten zerstört. Dadurch beinhalten die Komponenten keine Verbindung mehr zu ihren Modulen. Abschließend wird das Spielobjekt aus dem Manager entfernt. Ein Sonderfall tritt hier in Verbindung mit untergeordneten Spielobjekten auf. Eine einfache Durchführung der *Destroy*-Phase würde diese nicht aus dem Manager entfernen. Somit ist es notwendig, während dieser Phase zwischen tatsächlichen Komponenten und untergeordneten Objekten zu unterscheiden. Letztere werden in dieser Phase wieder über den Manager zerstört.

3.4.7 Erweiterungsmöglichkeiten

Das hier vorgestellte Modell verfügt noch über Potenzial für Erweiterungen. Die vielversprechendsten sollen hier nun kurz vorgestellt werden. Die aufgezeigten Möglichkeiten zielen darauf ab wichtige Funktionalitäten hinzuzufügen oder die bestehenden zu verbessern.

Dependency Injection

Der Plug-In Mechanismus verfügt bereits über eine *Dependency Injection* auf Basis von *Annotations* und *Reflection*. Das vorliegende Modell für die komponentenbasierte Spielobjekt-Vewaltung verzichtet zwar auf eine automatisierte Verknüpfung. Sollte diese jedoch gewünscht werden ließe sich diese ebenfalls über *Dependency Injection* lösen (siehe [18]).

Bei der Umsetzung dieser Erweiterungsmöglichkeit sollte jedoch bedacht werden, dass in einem Spiel häufig Spielobjekte erzeugt werden. Die Verwendung von *Reflection* kann bei gewissen Anwendungen die Performance reduzieren. Bevor diese Erweiterung auf das Modell übernommen werden, müssten noch eingehende Performance Tests durchgeführt werden. Weitere Richtlinien für die Evaluierung finden sich in [6, Kap. 9].

Source Bridge

Im Zusammenspiel mit einem Editor ist es notwendig dem Editor Definitionen der Komponenten zur Verfügung zu stellen. Diese Definitionen beinhalten Informationen über vorhandene Variablen und deren Standardwerte sowie Meta-Informationen über deren Funktionalität und Zugehörigkeit zu Modulen.

In der vorhandenen Literatur werden diese Informationen über XML-Dateien zur Verfügung gestellt [19, Abschnitt 4.3]. Dieser Ansatz erfüllt seine Funktion durchaus. Der Nachteil ist jedoch, dass die XML-Dateien unabhängig von den eigentlichen Komponenten sind. Werden diese verändert, muss die Änderung zusätzlich auch in die XML-Datei übernommen werden. Dieser zusätzliche Schritt stellt eine potenzielle Fehlerquelle dar.

Einen in Java entwickelten Editor vorausgesetzt, besteht die Möglichkeit diese *Source Bridge* ebenfalls über *Reflection* zu lösen. Die Definitionen können auf diesem Weg direkt aus den vorhandenen Klassen extrahiert werden. Die Kennzeichnung von im Editor sichtbaren Variablen kann in einem solchen Fall mittels *Annotation* erfolgen.

Der Plug-In Mechanismus der vorliegenden Arbeit verwendet *Reflection* hauptsächlich nur während dem Starten der Engine. Etwaige performancekritische Aufgaben werden somit nur zu Beginn des Spiels ausgeführt.

Vereinheitlichte *Init*-Phase

Im vorliegenden Modell sieht der Lebenszyklus für Komponenten und Spielobjekte eine *Init*-Methode vor. Diese unterscheidet sich von Komponente zu Komponente (bzw. von Spielobjekt zu Spielobjekt) aufgrund der notwendigen Parameter. Eine Erweiterungsmöglichkeit des Modells ist die Vereinheitlichung dieser Methode. Dadurch ließe sie sich besser in den ansonsten automatisierten Prozess der Initialisierung einbinden.

In *CogaenJ* wurde die Initialisierung bereits vereinheitlicht. Dazu wurde ein Parameter-System verwendet, welches es erlaubte über eine Instanz auf einen sogenannten *Buffer* alle notwendigen Werte zu übergeben. Ein solches Parameter-System wird in [9] beschrieben. Während der Entwicklung von *CogaenJ* und der Anwendung davon für diverse Technik-Demos zeigte sich jedoch, dass diese Schnittstelle für die Erzeugung von Spieleobjekten direkt im Quellcode sehr unhandlich ist. Dies ergibt sich aus der Tatsache, dass vor jeder Initialisierung eines Spielobjektes im Quellcode erst ein *Buffer* initialisiert und befüllt werden musste bevor dieser an das Objekt übergeben wurde.

Die Einführung einer solchen Schnittstelle ist erst dann wirklich sinnvoll, wenn sie im Zusammenhang mit der Erzeugung von Spielobjekten über externe Skripte oder Konfigurationsdateien geschieht. Dann wird der *Buffer* durch das auslesen der Dateien befüllt. Für die Erzeugung von Spielobjekten im Quellcode sollte jedoch auch weiterhin eine einfachere Methode, wie die hier vorgestellten individuellen *Init*-Methoden, vorhanden bleiben.

Externe Dateiformate

Im vorliegenden Modell geschieht die Definition und Erzeugung von Spielobjekten nur über den Quellcode. Die wesentlichste Erweiterung zielt darauf ab, die vorgestellte Architektur *data driven* zu gestalten (siehe [8, Abschnitt 13.3] und [21, Seite 629]). Dazu muss ein Weg bereit gestellt werden, die Spielobjekte über externe Dateien zu definieren. In engem Zusammenhang steht hierbei auch die Art und Weise wie die Spielobjekte erzeugt werden (z. B. in einer Datei für einen ganzen Level). Die Wahl des Formats macht *Gregory* von den Präferenzen und Fähigkeiten des Entwicklerteams abhängig.

Ebenso entscheidend für die Wahl ist die Phase des Projekts. Während der Entwicklung werden Formate benötigt, die gegebenenfalls vom Entwickler manuell erstellt bzw. korrigiert werden. Hierfür hat sich XML bewährt und wird in zahlreichen Projekten verwendet. Nach der Auslieferung an den Kunden (den Spieler) empfiehlt sich ein Format zu wählen, dass nicht so leicht modifiziert werden kann. Dies erschwert (besonders wichtig im Multiplayer-Sektor) ungewollte Modifikationen.

3.5 Kommunikation

3.5.1 Anforderungen

Vereinheitlichte Schnittstelle

Um eine Kommunikation innerhalb der Engine und zwischen den unabhängigen Modulen zu ermöglichen, wird eine vereinheitliche Schnittstelle benötigt. Diese Schnittstelle ist für alle an der Kommunikation teilnehmenden Elemente verbindlich. Das einzelne Module für die Kommunikation zwischen den internen Bestandteilen eigene Schnittstellen implementieren ist zwar möglich, davon sollte jedoch abgesehen werden.

Die Definition der Schnittstelle muss eine einheitliche Datenstruktur zum Austausch von Informationen bereitstellen. Eine große Schwierigkeit stellen hierbei der unterschiedliche Umfang der versandten Information sowie deren unterschiedliche Datentypen dar. Die in Abschnitt 2.2 vorgestellten und im vorliegenden Modell verwendeten Möglichkeiten bieten auf diesem Gebiet jeweils ihre Vor- und Nachteile sowie ihre geeigneten Anwendungsfälle. Bei der Wahl der Datenstruktur, gilt es auch zu berücksichtigen, dass bei der Entwicklung eines konkreten Spiels konkrete Datenpakete entstehen werden. Es besteht somit der Bedarf, nicht nur eine übergreifende einheitliche Datenstruktur zu verschicken, sondern ebenso speziell entwickelte, welche einen leichten und schnellen Zugriff auf die beinhaltete Information bieten. Der gemeinsame Nenner dieser Datenstruktur ist der konfigurierbare Identifikator eines Datenpaketes. Dadurch kann ein Empfänger mit derselben Schnittstelle zwischen unterschiedlichen Datenpaketen unterscheiden.

Zusätzlich muss beim Versand die Möglichkeit zur Festlegung eines Empfängers bestehen. Jeder Teilnehmer verfügt dabei über einen eindeutigen Identifikator über den er gezielt angesprochen werden kann. Dies ermöglicht es, dass mehrere Teilnehmer auf den Erhalt derselben Information reagieren können, durch den gezielten Versand jedoch unabhängig voneinander sind.

Die Definition des Empfängers ergibt sich aus dessen Adressierbarkeit über seinen eindeutigen Identifikator sowie der Möglichkeit die einheitliche Datenstruktur zu übergeben. Dies erfolgt in der Regel über die Implementierung einer einheitlichen Schnittstelle für die Empfänger.

Die explizite Definition der Schnittstelle eines Senders kann entfallen. Jedes Element, welches Informationen in das einheitliche Format packen kann und Zugriff auf einen Kanal zum Versand hat, kann als Sender fungieren. Die explizite Definition von Kanälen ist nur in manchen Fällen notwendig. Als valider Kanal kann jede Form dienen, die es ermöglicht den richtigen Empfänger abzufragen und die Schnittstelle zum Empfang der Information aufzurufen. Sinnvoll erweist sich die Definition, in Anwendungsfällen mit multiplen Empfängern die dem Sender unbekannt sind. Dabei kennt der Sender nur den Kanal über den er die Informationen verschickt, unabhängig davon wie viele bzw. ob ein Empfänger auf die Information wartet.

Reaktive Kommunikation

Einen Sonderfall stellt die *Reaktive Kommunikation* dar. Diese zeichnet sich durch ihren völlig datenlosen Kommunikationsprozess aus. Das Ziel dabei ist es, einen Kommunikationskanal zur Verfügung zu stellen, der einzig und allein zur Auslösung von Reaktionen dient. Der vollständige Verzicht auf Daten ermöglicht es ohne großen Konfigurationsaufwand unterschiedlichste Elemente zu verbinden. Einzig und allein durch die Verbindung ergibt sich ein logischer Zusammenhang. Ein beispielhafter Anwendungsfall für diesen Kanal stellt eine Logik-Komponente dar, welche darauf wartet ein Ereignis auszulösen (z. B. die Detonation einer Bombe). Für die Logik-Komponente ist es dabei unwesentlich, ob sie den Befehl zur Reaktion durch einen Tastendruck, ein zeitlich gesteuertes Element (z. B. Timer) oder einer anderen Logik-Komponente erhält. Würden für diese Anwendungsfälle die weiter oben beschriebene vereinheitlichte Schnittstelle verwendet werden, müsste die Logik-Komponente auf eine vordefinierte Nachricht reagieren. Die Sender würden Informationen über den Empfänger sowie die Zusammensetzung der Information benötigen und alle beteiligten Elemente müssten per Konfiguration aufeinander abgestimmt werden. Für diese eigentlich simple Kommunikation soll dieser Aufwand so weit wie möglich entfallen.

Broadcasts

Einen weiteren Sonderfall in bestehenden Kommunikationsmodellen bilden die sogenannten *Broadcasts*. Diese dienen zur einfachen Kommunikation mit allen relevanten Empfängern. Die Schwierigkeit bei der Umsetzung dieser Kommunikationsmöglichkeit liegt in der Suche nach einer sinnvollen Definition von *alle*. Der Versand von Daten kann sich in großem Umfang mit zahlreichen Empfängern durchaus als aufwändig erweisen. Zusätzlich ist es unter Berücksichtigung der definierten Schnittstellen einleuchtend, dass der Versand von Information an Empfänger, welche mit den Daten nichts anfangen können, völlig überflüssig ist.

Anwendung finden *Broadcasts* dennoch, da sie eine Möglichkeit zur Kommunikation mit mehreren Empfängern bieten ohne diese zu kennen. Wird ein zusätzlicher Empfänger hinzugefügt entsteht kein weiterer Konfigurationsaufwand beim Sender.

Ein Anwendungsfall findet sich in *Unity3D*. Dort ist es möglich ein Spielobjekt mit mehreren Skripten auszustatten. Jedes Skript stellt dabei einen Empfänger dar und jede Methode im Skript ist eine Nachricht auf die reagiert wird, mit den gewünschten Parametern als Daten. Dadurch ist es möglich, dass ein Skript für die Logik die Nachricht *Die* als *Broadcast* versendet. Empfangen wird diese von allen Skripten, wobei die Methode *Die* aufgerufen wird. Dies erlaubt eine flexible und unabhängige Zusammenstellung der Logik.

3.5.2 Bestandteile

Message<T>

Als Datenkapsel dient im vorliegenden Modell die Nachricht (bezogen auf die Implementierung als *Message* bezeichnet). Sie enthält einen String als Identifikator, welcher es den Empfängern ermöglicht, zu erkennen um welche Nachricht es sich handelt. Dies spielt im Bezug auf die Zusammenstellung der Daten eine wesentliche Rolle.

Zusätzlich zum Identifikator enthält die Nachricht eine Referenz auf das generische Datenpaket. Der Datentyp wurde generisch gewählt, da dies ermöglicht, jede beliebige Information zu verpacken und zu verschicken. Die Bedingung dafür ist, dass der Empfänger anhand des Identifikators weiß, welche Datenstruktur er empfangen hat und wie diese auszuwerten ist. Grundlegend gilt sowohl für die Empfänger als auch die Sender, dass sie über mindestens eine konfigurierbare Message-ID verfügen die auf eine vordefinierte Datenstruktur wartet. Die Message-ID ist konfigurierbar um das Zusammenspiel mehrerer Komponenten zu erleichtern. Versuche die das gleiche Zusammenspiel erzielen sollten, bei der jedoch jeweils nur Sender oder Empfänger konfigurierbar waren, scheiterten. Anzumerken ist noch, dass das Datenpaket auch leer sein kann, wenn die Nachricht keinerlei Daten benötigt.

Durch die Wahl eines generischen Modells können sowohl bei der unabhängigen Implementierung der Teil-Engines als auch bei der Entwicklung eines konkreten Spiels Vorteile erzielt werden. Bei den Teil-Engines wird die vereinheitlichte Datenstruktur verschickt, wogegen bei der Erstellung von konkreten Logik-Komponenten vereinfachte Strukturen verwendet werden können. Ein Beispiel dafür ist, dass bei einem konkreten Spiel die Nachricht *PlayerDied* versandt wird. Nachdem zu diesem Zeitpunkt das Spielobjekt *Player* bereits existiert und für alle Komponenten bekannt ist, müssen die Informationen welcher Spieler starb, nicht in einem gesonderten Schritt neu verpackt werden, sondern es kann die Instanz des Spielers selbst verschickt werden und jede Komponente kann die gewünschten Daten selbst daraus auslesen. Im vorliegenden Modell wird dies dadurch begünstigt, dass jedes Spielobjekt eine eigene Klasse hat welche um entsprechende Methoden erweitert werden kann.

Bei der Wahl der übergreifenden Datenstruktur muss darauf geachtet werden, dass sie getrennt befüllt und gelesen wird. Um eine möglichst fehlerfreie Kommunikation zu ermöglichen sollten den Empfängern keine Schreibrechte zur Verfügung stehen. In *Cogaen* wird dafür ein *Reader*-Interface benutzt, welches den Zugriff auf die Daten ähnlich einem Iterator erlaubt. Das Interface beinhaltet für jeden primitiven Datentyp eine entsprechende *read*-Methode. Nach jedem Auslesen eines Wertes schaltet der *Reader* automatisch auf den nächsten Datentyp.

MessageHandler

Die Schnittstelle, welche im Modell einen Empfänger auszeichnet, ist der *MessageHandler*. Sie definiert eine Methode der sowohl der Identifikator der Nachricht als auch die *Message* übergeben wird. Die separate Angabe des Identifikators geschieht nicht nur aus Bequemlichkeit. Erste Versuche ihn an die erste Stelle des Datenpakets zu packen und dort von jedem Empfänger auslesen zu lassen führte zu Problemen bei der Vererbung der Empfänger. In einem konkreten Beispiel war *MessageHandler B* von *MessageHandler A* abgeleitet. Dabei erhielt *B* die Nachricht und reichte sie zuerst an *A* weiter. Dieser las den Identifikator aus, wodurch der *Reader* weiter schaltete. Als *B* nun die Message-ID auslesen wollte, kam es zum Fehler, da der *Reader* nicht mehr an die Stelle des Identifikators zeigte.

EventManager

Gegeben durch die Vorbedingung, dass sich ein Empfänger für den Empfang von Events zuvor registrieren muss, entstand der *EventManager*. Dieser wird regulär über ein Plug-In als Modul beim Kern registriert und steht somit in der gesamten Engine zur Verfügung. Bei diesem Element können sich die *MessageHandler* für einen oder mehrere Events unter Angabe des dazugehörigen Identifikators registrieren. Die Events selbst sind *Messages* die über den *EventManager* an die registrierten Empfänger verschickt werden. So ist es möglich mehrere unbekannte Empfänger gleichzeitig anzusprechen.

Signals and Slots

Die Umsetzung der *Reaktiven Kommunikation* erfolgt in Anlehnung an das *Singnal and Slot* System (kurz *SnS*) von *Trolltech QT*. Da dieses System im vorliegenden Modell jedoch für die datenlose Kommunikation verwendet wird (s. oben), erfolgt dies von bestehenden Beschreibungen und Implementierungen abweichend [9].

Im vorliegenden Modell ist das adaptierte SnS-System in die Komponenten eingebettet. Eine Komponente kann Signale empfangen, wodurch die *Signals* für die Eingänge stehen und die *Slots* für die Ausgänge. Verbunden werden sie direkt miteinander während der *Link*-Phase eines Spielobjekts. Dabei können einem *Slot* beliebig viele *Signals* hinzugefügt werden. Angeboten werden die Ein- bzw. Ausgänge von den Komponenten selbst. Durch die direkte Verbindung ohne Umwege ist eine schnelle Kommunikation gewährleistet, da keinerlei Suchvorgänge (wie z. B. für das empfangende Spielobjekt) notwendig sind. Wann immer möglich sollte auf dieser Kanal für die Kommunikation verwendet werden.

3.5.3 Verknüpfung mit dem Komponenten-Modell

Das Kommunikationsmodell wurde in das Komponenten-Modell eingebettet. Der Grund für diese Entscheidung ist der Umstand, dass im vorliegenden Modell die Komponenten sowohl die Empfänger als auch die Sender darstellen. Für ihre Rolle als Empfänger implementieren die Komponenten, welche Nachrichten empfangen können, das Interface *MessageHandler*. Während der *Init*-Phase ihres Lebenszyklus können sie sich bei ihrem zugehörigen Spielobjekt für den Empfang einer Nachricht registrieren. Die Registrierung erfolgt konkret für den Empfang einer Nachricht mit einer bestimmten Message-ID. Sollte die Komponente mehrere Nachrichten empfangen können, muss sie sich für alle Nachrichten einzeln registrieren. Insgesamt bieten sich einer Komponente somit vier verschiedene Wege zur Kommunikation mit anderen Komponenten, welche sich auch in anderen Spielobjekten befinden können (siehe Abb. 3.5 (a)). Diese sind die direkten Referenzen, *Signals and Slots*, Nachrichten und Events. Die *ComponentChangedListener* werden dabei nicht als Kommunikationskanal gewertet.

Das Spielobjekt agiert dabei als Sammelstelle für die Registrierungen und übernimmt die Verteilung der eingehenden Nachrichten. Das Spielobjekt selbst wertet keine Nachrichten aus. Als Empfänger wird immer ein Spielobjekt angegeben da diese im Modell bereits über einen eindeutigen Identifikator verfügt. Das Spielobjekt selbst implementiert ebenfalls das Interface *MessageHandler* wodurch es sich als Empfänger für Nachrichten und Events qualifiziert. Dafür verwaltet es zu den bisherigen Bestandteilen des Komponenten-Modells (siehe Abb. 3.4) noch die Sammlung der als Empfänger registrierten Komponenten. In Abb. 3.5 (b) hat sich *Component 2* für den Empfang einer Nachricht registriert.

Im Fall von untergeordneten Spielobjekten werden alle Empfänger im obersten Spielobjekt registriert. Untergeordnete Spielobjekte verfügen über einen eigenen Identifikator und können somit selbst Nachrichten empfangen. Damit diese Nachricht jedoch im gesamten zusammengesetzten Spielobjekt verteilt wird, werden beim Empfang die Nachrichten an das oberste Element weitergereicht und von diesem verteilt. Dadurch wird sichergestellt, dass die Nachrichten an alle Komponenten in der gesamten Hierarchie verteilt werden, unabhängig davon ob das oberste Spielobjekt oder ein untergeordnetes adressiert wird.

Die Registrierung erfolgt, wie bereits erwähnt, während der *Init*-Phase des Lebenszyklus. Dabei registriert sich eine Komponente zuerst bei ihrem Spielobjekt für den Empfang von Nachrichten. Wird dieses Spielobjekte einem anderen untergeordnet, registriert dieses wiederum alle ihm bis dato bekannten bei dem übergeordneten Spielobjekt. Dadurch ergibt es sich, dass alle Spielobjekte die Empfänger der untergeordneten kennen, jedoch nicht diese, welche sich auf derselben Ebene oder darüber befinden.

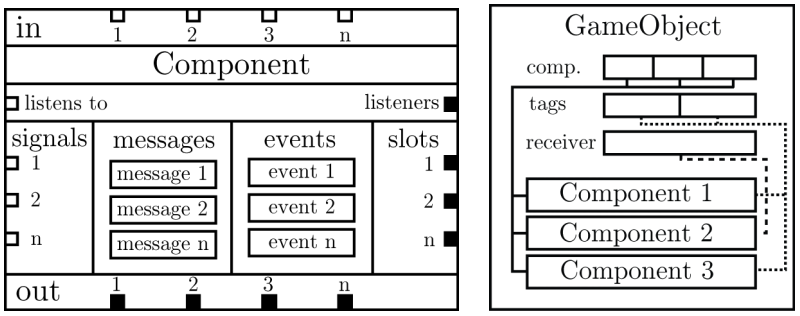


Abbildung 3.5: Darstellung der um die Kommunikationskanäle erweiterten Modelle für die Komponenten und die Spielobjekte. Die ursprüngliche Form findet sich in Abbildung 3.4.

3.5.4 Kommunikationskanäle

Nachrichten

Die Kommunikation erfolgt im vorliegenden Modell in erster Linie über die Nachrichten. Die dafür benötigte Funktionalität steht direkt in der Komponente zur Verfügung. Zusätzlich zur Nachricht muss eine versendende Komponente noch den Identifikator des empfangenden Spielobjekts speichern. Der Empfänger wird nicht in der Nachricht selbst gespeichert, da einerseits eine Nachricht an mehrere Empfänger versandt werden kann oder im Falle der Events an keinen konkreten Empfänger gerichtet ist.

Für die Kommunikation mittels Nachrichten ergeben sich die in Folge beschriebenen unterschiedlichen Kanäle:

Inter-Objekt-Kommunikation: Der Versand einer Nachricht über die Grenzen eines Spielobjektes hinaus erfordert zwingend die Angabe eines Empfängers. Dabei wird vom *GameObjectManager* anhand des übergebenen Identifikators das entsprechende Spielobjekt gesucht und diesem die Nachricht zugestellt. Ob sich im entsprechenden Spielobjekt für die Nachricht Empfänger registriert haben ist nicht relevant. Ebenso wird der Versand an einen nicht existierenden Empfänger (ev. entstanden durch einen falschen Identifikator) ohne Fehlermeldung verworfen. Dies ist notwendig um die gewünschte Flexibilität zu erreichen. Dadurch können Empfänger erst später entwickelt und hinzugefügt bzw. in weiteren Verlauf ausgetauscht werden. Ein Nachteil ergibt sich dadurch beim debuggen, da ein Entwickler erst dem Lauf der Nachricht folgen muss um im Fehlerfall die Ursache für eine nicht ausgelöste Reaktion zu finden.

Intra-Objekt-Kommunikation: Der Versand einer Nachricht innerhalb des eigenen Spielobjektes bzw. in untergeordneten Spielobjekten erfolgt ohne Beteiligung des *GameObjectManager*. Die Suche nach einem speziellen Spielobjekt kann je nach Implementierung und Anzahl der existierenden Spielobjekte aufwändig sein. Darum wird diese Suche von vornherein ausgeschlossen. Einerseits steht für diesen Kanal eine eigene Methode zur Verfügung, andererseits wird auf dem normalen Weg vor der Suche überprüft ob sie an das eigene Objekt verschickt wird. Der Empfang und die Weiterleitung erfolgen in beiden Fällen gleich.

Events

Bei der Kommunikation über Events erfolgt sowohl innerhalb als auch über die Grenzen eines Spielobjekts hinaus auf die gleiche Weise. Streng genommen kann eine Komponente auf diesem Weg auch mit sich selbst kommunizieren. Die Registrierung erfolgt über den *EventManager* und steht jedem Element zur Verfügung, welches das Interface *MessageHandler* implementiert. Für eine vereinfachte Handhabung stehen in der Komponente entsprechende Methoden für die Registrierung den Versand von Events zur Verfügung.

Der Versand erfolgt einfach durch die Übergabe der zu sendenden Nachricht. Die Message-ID kennzeichnet dabei den Typ des Events und wird vom *EventManager* ausgelesen. Anschließend werden alle für diesen Event-Typ registrierten Empfänger gesucht und in der Reihenfolge ihrer Registrierung benachrichtigt.

Dieser Kanal ist am besten geeignet für die Inter-Objekt-Kommunikation mit mehreren für den Sender unbekannten Empfängern. Auch auf diesem Kommunikationsweg wird kein Fehler gezeigt, wenn es für den versandten Event-Typ keine registrierten Empfänger gibt.

Signals and Slots

Das SnS-System stellt den letzten Kanal dar und eignet sich am besten für die Intra-Objekt-Kommunikation, kann jedoch auf Umwegen auch für die Inter-Objekt-Kommunikation verwendet werden.

Vorteile bietet dieser Kanal durch seinen schnellen Übertragungsweg ohne Suche und durch den Entfall der Konfiguration von Message-IDs. Der Nachteil entsteht durch den Bedarf der direkten Verbindung zwischen Sender und Empfänger. Bei den Nachrichten und den Events können Empfänger ausgetauscht werden ohne dass dadurch die Konfiguration eines Spielobjektes geändert werden muss. Beim SnS-System müssen bei einer Änderung auch die Verbindungen modifiziert werden. Zusätzlich müssen die direkten Verbindungen während der *Destroy*-Phase des Lebenszyklus getrennt werden um Verbindungen zu gelöschten Komponenten zu vermeiden.

3.5.5 Erweiterungsmöglichkeiten

Datentransport über *Signals and Slots*

Im vorliegenden Modell wird das SnS-System für die datenlose Kommunikation benutzt, was für die derzeitigen Anwendungsfälle des Modells durchaus ausreichend ist.

Von Bedeutung kann diese Fähigkeit sein, wenn das Modell der Engine um einen Editor erweitert wird. Als Beispiel dafür kann *Unreal Kismet*[2] genannt werden, welches Bestandteil des Editors für die *Unreal Engine* ist. *Kismet* ist eine visuelle Programmiersprache mit auch die Logik über einzelne Bausteine erstellt wird. Besteht nun die Anforderung die Logik-Komponenten über das SnS-System zu verbinden und sie dabei Daten austauschen müssen, muss das System um den Transport von Daten erweitert werden (siehe [9]).

Nachrichtenbasierte Engine

In [10] beschreibt *Harmon* ein System mit dem die ganze Engine über Nachrichten gesteuert werden kann. Die Nachrichten dienen darin nicht mehr nur zur Kommunikation sondern z. B. auch für den Ersatz einer Spielschleife. Als Vergleich heißt es darin:

> A good metaphor for this is the Win32 windowing API. Everything the Windows GUI does is in response to messages that can come from the application, other windows, or the operating system. Entities, like windows, have no knowledge of each other''s internal data or implementation. They simply send and respond to a set of well-defined messages.

In Bezug auf das vorliegende Modell ergeben sich Bedenken auf Grund der großen Unabhängigkeit der Spielobjekte. Ein weiteres bestehendes Problem ist die doppelte Speicherung von Daten (siehe Abschnitt 3.4.2). Dieses Problem wird von *Harmon* nicht berücksichtigt. Dennoch könnte dieses System in sehr wenigen und sehr speziellen Anwendungsfällen interessant sein. Ein Beispiel dafür ist eine Engine die auf mehreren Rechnern gleichzeitig läuft. Anstelle des direkten Aufrufs der Update-Zyklen könnte der Server die Clients über das Netzwerk verschickte Nachrichten steuern. Interaktive Installationen die auf zahlreichen Monitoren ganze Räume einnehmen könnten von diesem System profitieren.

[2] http://www.unreal.com/features.php?ref=kismet

Kapitel 4

Implementierung

Dieses Kapitel beschreibt die Details der Implementierung von *Java Game Components* (JGC), welche im Zuge des praktischen Teils dieser Arbeit entstand. Bei der Entwicklung lag der Fokus auf dem Beweis der Realisierbarkeit des vorgestellten Modells inklusive der Verwendung aller Features von Reflection und Annotations. Der Umfang beinhaltet neben dem Kern, der Spielobjekt-Verwaltung und den Klassen für die Kommunikation auch noch rudimentäre Prototypen einiger Teil-Engines (z. B. für das Rendering), welche für die Implementierung einer Demo notwendig waren.

4.1 Setup

Die Wahl der Sprache für die Implementierung fiel auf Java. Der Hauptgrund für die Entscheidung ist, dass Java alle benötigten Sprachfeatures aus Abschnitt 2.3 zur Verfügung stellt. Zusätzlich besteht für den Autor aus vorangegangenen Projekten ein großer Erfahrungswert in der Programmierung mit Java. Entwickelt wurde mit dem JDK 1.6 (Java Development Kit). Generell ist die Implementierung bis zur Java-Version 1.5 abwärtskompatibel. Als Entwicklungsumgebung wurde Eclipse 3.6 Helios mit dem Subclipse Plug-In verwendet. Die Erstellung der Builds wurde mittels Ant realisiert. Die Builds schlossen auch die automatisierten Unit-Tests mit dem JUnit Framework 4.4 ein.

Um ungewollte Abhängigkeiten zu vermeiden, wurde für jedes Modul ein eigenes Projekt angelegt dem im Build Path nur das Projekt des Kerns sowie die als Abhängigkeit definieren Projekte hinzugefügt wurden.

4.2 Plug-In Mechanismus

4.2.1 Architektur

Die Implementierung ist die Umsetzung des vorgestellten Modells inklusive aller Elemente und Funktionalitäten (siehe Abschnitt 3.3). An manchen Stellen wurde aus Gründen einer sauberen Implementierung (z. B. die Bündelung aller Funktionalität die Reflection benötigt in eine Klasse) das Modell um zusätzliche Elemente erweitert. Die Annotation zur Definition von Abhängigkeiten wird in Abschnitt 4.2.3 beschrieben. Ein vollständiges UML-Diagramm des Plug-In Mechanismus findet sich in Abschnitt A.2.

`Core`

Der `Core` ist die Implementierung des Kerns. Er übernimmt die Bereitstellung der Funktionalität der Verwaltung. Die bei ihm registrierten Plug-Ins werden in einer einfachen Liste gesammelt. Zu Beginn ist die Liste nach der Reihenfolge der Registrierung sortiert, wird jedoch während der Initialisierung einer `Application` in die Startreihenfolge der Plug-Ins umsortiert. Eine weitere Speicherung der Plug-Ins ist nicht vorgesehen. Unter Umständen würde es für die Abfrage der Module Geschwindigkeits-Vorteile bieten, würden die Module zusätzlich anhand ihres Identifikators gespeichert werden. Dies gestaltet sich jedoch auf Grund der gewählten Identifikatoren schwierig (siehe Abschnitt 4.2.2).

Der `Core` durchläuft von der Instanzierung bis zu seiner Zerstörung mehrere Status. Diese Status sind Teil des eingebauten Schutzmechanismus. Dabei sind für jede Phase des Lebenszyklus zwei unterschiedliche Status, einer zum Beginn der Phase und einer am Ende, sowie ein Status vor Beginn der ersten Phase vorgesehen. Manipulationen an der Sammlung von Plug-Ins sind nur bei den Status `cold` und `clean` möglich. Eine Liste aller Status und ihr Eintreten zeigt Tabelle 4.1. Bei jedem Wechsel des aktuellen Status werden die `CoreStatusListener` in der Reihenfolge ihrer Registrierung benachrichtigt. Darin nicht berücksichtigt ist der erste Status (`cold`) da dieser bereits im Konstruktor gesetzt wird. Für den Fall, dass Listener auch über den ersten Status informiert werden müssen, steht ein eigener Konstruktor zur Verfügung, dem die Listener übergeben werden.

`CorePlugIn`

Ein im Modell nicht beschriebenes Element ist das `CorePlugIn`. Diese Plug-In ermöglicht es, dass sich der Kern selbst als Modul verwaltet und somit über den normalen Weg zur Definition von Abhängigkeiten angefordert werden kann. Das Plug-In wird vom Kern selbst registriert und steht somit immer zur Verfügung.

Tabelle 4.1: Die nach ihrem Eintreten geordnete Liste der verschiedenen Status in denen sich der `Core` während dem Lebenszyklus befindet.

Status	*Eintreten des Status*
`cold`	vor Beginn des Lebenszyklus
`preparing`	nach Betreten der Phase `prepare`
`prepared`	vor Verlassen der Phase `prepare`
`starting`	nach Betreten der Phase `start`
`started`	vor Verlassen der Phase `start`
`stopping`	nach Betreten der Phase `stop`
`stopped`	vor Verlassend er Phase `stop`
`cleaning up`	nach Betreten der Phase `clean up`
`clean`	vor Verlassen der Phase `clean up`

`CoreStatusListener`

Die `CoreStatusListener` können beim `Core` registriert werden und werden über die Wechsel des Status (siehe Tabelle 4.1) benachrichtigt. Einfachstes Anwendungsbeispiel für einen solchen Listener ist das loggen der Übergänge. Grundsätzlich hätte ein Status pro Phase des Lebenszyklus gereicht, jedoch kann mit dem implementierten Ansatz ein Listener vor Beginn der eigentlichen Aufgaben (z. B. dem Vorbereiten der Plug-Ins) und nach Abschluss dieser benachrichtigt werden. Somit ist der eigentliche Verwendungszweck der Listener es zu ermöglichen, dass Module ihre Aufgaben außerhalb des Lebenszyklus erledigen. Ein komplexeres Anwendungsbeispiel ist die Konfiguration der Module anhand externer Konfigurationsdateien zwischen den Phasen `prepare` und `start`.

`Application`

Für das korrekte Starten einer Applikation stellt der Kern ein Framework zur Verfügung. Diese Applikation muss dazu das Interface `Application` implementieren. Ein korrekter Betrieb des Kerns auf einem anderen Weg ist nicht möglich.

Das Interface umfasst für jede Phase des Lebenszyklus eine entsprechende parameterlose Methode. Die Aufgaben, welche in den einzelnen Methoden auszuführen sind, finden sich in Abschnitt 3.3.6. Die Übergabe einer Referenz auf den zugehörigen `Core` wurde nicht eingeplant. Einerseits kann eine Applikation mehrere Kerne gleichzeitig betreiben (z. B. bei Netzwerkspielen mit je einem Kern für Server und Client), andererseits kann der jeweilige Kern der Applikation anderweitig übergeben werden. Das vollständige Interface zeigt Programm 4.1.

Programm 4.1: Das Interface für die Anwendungen welche vom `Core` gestartet und verwaltet werden.

```
1 public interface Application {
2
3   public void init();
4   public void prepare();
5   public void start();
6   public void stop();
7   public void cleanUp();
8
9 }
```

`DependencyManager`

Der `DependencyManager` dient dazu die benötigte Funktionalität aus Reflection in einer Klasse zu bündeln. Er übernimmt die Extraktion der Information aus den `Class<T>`-Objekten inklusive dem Auslesen der definierten Abhängigkeiten. Im Weiteren zählen auch die Injektion der Abhängigkeiten sowie das Erstellen der Startreihenfolge zu seinen Aufgaben. Durch diese Trennung ist es möglich, bei Bedarf Teile der Funktionalität (z. B. die Injektion der Abhängigkeiten) zu ändern ohne dass dafür der `Core` verändert werden muss. Dies ist jedoch nur möglich, solange weiterhin der gleiche Mechanismus zur Identifikation verwendet wird, da dieser bei den öffentlichen Methoden des Kerns verwendet wird.

`PlugIn<T>`

Das Plug-In beinhaltet für jede Phase des Lebenszyklus eine entsprechende Methode. Zusätzlich wird noch eine Methode benötigt um Zugriff auf das zur Verfügung gestellte Modul zu erhalten. Für die Umsetzung des Interfaces wurde ein generischer Ansatz gewählt, wodurch der Rückgabetyp dem des Moduls entspricht (Programm 4.2 Zeile 3). Zwar wäre es möglich den Rückgabetyp einheitlich als `Object` zu definieren aber dieser Ansatz bietet Vorteile. Der Plug-In Mechanismus, genauer der `DependencyManager`, benötigt einen Weg um den Typ des zur Verfügung gestellten Moduls zu ermitteln. In Verbindung mit der generischen Implementierung, Reflection und der Identifizierung über die Klasse kann diese Information jedoch aus dem generischen Rückgabewert extrahiert werden.

Der Lösungsweg ohne den generischen Rückgabe-Typ würde eine zusätzliche Methode erfordern, welche die Klassen-Information liefert. Das Abfragen des Klassen-Objekts aus dem zurückgegebenen `Object` ist nicht möglich, da die Information benötigt wird, bevor die Methode `prepare()` aufgerufen wird und somit nicht sichergestellt ist, dass nicht `null` zurückgeliefert wird.

Programm 4.2: Das Interface des Plug-In mit den Methoden für den Lebenszyklus sowie der Methode zum Zugriff auf das Modul in Zeile 3.

```
1 public interface PlugIn<T> {
2
3   public T getModule();
4
5   public void prepare();
6   public void start();
7   public void stop();
8   public void cleanUp();
9
10 }
```

Fehlerklassen

Jeder Fehler der während dem Hoch- bzw. Herunterfahren des Plug-In Mechanismus auftritt wird in der vorliegenden Implementierung generell als kritisch eingestuft. Dies lässt sich dadurch begründen, dass der Anwender (also der Spieler) keinen Einfluss auf die Zusammenstellung der Plug-Ins hat. Hier auftretende Fehler müssen bereits vom Entwickler vermieden werden. Daher ist jede der vier Fehlerklassen von `RuntimeException` abgeleitet. Folgende Fehlermöglichkeiten sind abgedeckt:

`CoreException:` Diese Ausnahme wird geworfen, wenn versucht wird, eine Applikation zu initialisieren obwohl bereits eine andere initialisiert wurde. Dies ist notwendig, da der `Core` nur eine Anwendung gleichzeitig betreiben kann.

`PlugInException:` Die Registrierung eines bereits hinzugefügten Plug-Ins, die Entfernung eines unbekannten Plug-Ins sowie das Anfordern eines unbekannten Plug-Ins sind abgedeckt.

`DependencyException:` Diese Ausnahmen treten bei der Verwaltung der Abhängigkeiten auf, wenn ungültige Definitionen (zirkuläre Abhängigkeiten) erkannt werden, die obligatorische Setter-Methode fehlt oder die Injektion aus anderen, meist Java spezifischen, Gründen fehlschlägt.

`ModuleException:` Da der Plug-In Mechanismus keine Voraussetzungen an ein konkretes Modul stellt, ist die einzige bestehende Fehlerquelle das Anfordern eines unbekannten Moduls.

4.2.2 Identifikation von Modulen und Plug-Ins

Die Identifikation der Plug-Ins und der Module wird in der präsentierten Implementierung über das `java.lang.Class<T>`-Objekt realisiert. Ähnliche Ansätze sind in einigen Sprachen, welche ebenfalls über eine Reflection-API verfügen, möglich. Ein kurzer Überblick über die Verfügbarkeit von Reflection in anderen Sprachen findet sich in Abschnitt 2.3. Steht dieser Ansatz in einer anderen Sprache nicht direkt zur Verfügung (z. B. ECMA-Script), ist die Verwendung des Klassen-Namens als String eine erste Alternative.

Eindeutigkeit: Da von Java für jede Klasse ein zugehöriges `Class<T>`-Objekt erzeugt wird und Klassen nur einmal vorhanden sein können ergibt sich automatisch eine Eindeutigkeit. Fehler durch die Verwendung von gleichen Identifikatoren werden vermieden, da zwei gleiche Klassen nicht möglich sind und der Compiler einen Fehler werfen würde. Zusätzlich steht die Information über die Klasse immer zur Verfügung, ein zusätzlicher (wenn auch minimaler) Speicheraufwand durch die Speicherung von zusätzlichen statischen Membervariablen entfällt.

Multiple Identifikatoren: Zusätzlich beinhaltet das `Class<T>`-Objekt nicht nur Informationen über die eigentliche Klasse sondern auch über alle implementierten Interfaces und alle Klassen von denen abgeleitet wurde. Somit entfallen in dieser Implementierung Schnittstellen die dazu benötigt würden, auf mehrere implementiere Interfaces zu überprüfen, da dies mit der Überprüfung eines `Class<T>`-Objektes realisiert werden kann. Ein sich daraus ergebender Nachteil ist, dass die Implementierung von Verwaltungs- und Kontrollaufgaben erschwert wird. Dies ist z. B. bei der Verwaltung der Module im `Core` der Fall. Da jedes Modul beliebige Interfaces implementieren kann, müsste bei der Kontrolle, ob ein konkretes Interface bereits zur Verfügung steht, jedes Interface eines jeden registrierten Moduls überprüft werden. Automatisch kann diese Überprüfung auch nicht erfolgen, da diese Kontrolle mit großer Wahrscheinlichkeit positiv ausfällt, bedenkt man, dass mehrere Klassen z. B. das Interface `Serializable` implementieren können.

String-Kompatibilität: In manchen Fällen, wie z. B. bei der Implementierung von Konfigurationsdateien, müssen die Identifikatoren als String darstellbar und auswertbar sein. Dies ist bei diesem System über den vollqualifizierten Klassennamen (engl. fully qualified class name) gegeben. Erzeugt werden kann dieser durch den Aufruf der Methode `getName()` des jeweiligen `Class<T>`-Objekts. Die Abfrage des Objekts anhand des Namens der Klasse erfolgt über die statische Methode `Class.forName(String name)`. Bei der Verwaltung der Module entfällt somit die Speicherung einer String-zu-Klasse-Sammlung.

Lesbarkeit: Die String-Representation einer jeden Klasse ermöglicht einen leichten Rückschluss auf die Klasse auf die sich der Identifikator bezieht. Im Vergleich dazu ist dies bei der Verwendung von UUIDs nicht gegeben. Dies erleichtert bei der Entwicklung das Debuggen, da bei Fehlermeldungen nicht erst recherchiert werden muss, um welche Klasse es sich handelt. Bei den UUIDs könnte die Lesbarkeit über eine Übersetzung (UUID zu Objektnamen) nachgebildet werden.

Einfacher Zugriff: Die Verwendung als Identifikator kann bei der Anforderung eines Moduls statisch erfolgen und steht im gesamten Projekt zur Verfügung. Wird z. B. der `GameObjectManager` angefordert, kann dies im gesamten Projekt durch `GameObjectManager.class` erfolgen. Wird von der Instanz eines Objektes die Klasse kann die Methode `getClass()` verwendet werden.

Einfache Instanzierbarkeit: Zusätzlich zur Information über die Klasse steht auch eine Vielfalt an Methoden zur Verwaltung zur Verfügung, teilweise auch über den zugehörigen `ClassLoader`. Einen parameterlosen Standard-Konstruktor vorausgesetzt, steht auf diesem Weg auch eine Factory-Methode zur Verfügung. Über sie lassen sich unkompliziert neue Instanzen erzeugen. Eine aufwändige Verwaltung von Factory-Klassen entfällt ebenso wie deren Implementierung und Wartung im weiteren Verlauf des Projekts.

Änderungsfreundlichkeit: Eine gute IDE[1] vorausgesetzt bietet dieser Ansatz einen erstaunlich geringen Wartungsaufwand bei Änderungen. Bei größeren Refactoring-Arbeiten (besonders in einem frühen Stadium der Entwicklung) tritt regelmäßig der Fall auf, dass Klassen umbenannt oder in andere Packages verlagert werden. Diese Änderungen können von der IDE automatisiert werden und eine manuelle Änderung der Identifikatoren ist nicht nötig.

Schnelle Vergleichbarkeit: Die `Class<T>`-Objekte werden von Java verwaltet und existieren nur einmal. Dies erlaubt eine schnelle Überprüfung auf Gleichheit mit dem `==`-Operator. Eingeschränkt wird dieser Vorteil durch die multiple Identifizierbarkeit. Ein einfacher Vergleich der Klassen-Objekte führt nicht zwangsläufig zum Erfolg, da unter Umständen nur nach einem implementierten Interface gesucht wird. Zur Verfügung gestellt wird die Funktionalität zum überprüfen auf implementierte Interfaces von Java selbst durch die statische Methode `Class.isAssignableFrom(Class<T> cls)`.

[1] für die Implementierung wurde *Eclipse Helios* verwendet

4.2.3 Verwaltung der Abhängigkeiten

Die Annotation `@Dependency`

Für die Kennzeichnung von Membervariablen eines Plug-Ins, welche die Abhängigkeiten darstellen, wird die Annotation `@Dependency` verwendet. Zwar kann die Annotation auch in anderen Klassen verwendet werden, jedoch sucht der `DependencyManager` lediglich in den registrierten Plug-Ins danach. Das Plug-In muss die übergebenen Referenzen anschließend an das dazugehörige Modul weiterreichen. Das Plug-In dient für die Abhängigkeiten somit als Schnittstelle wodurch es möglich wird, für die Verwaltung der Abhängigkeiten keine Anforderungen an das Modul zu stellen.

Das die Membervariablen für die Kennzeichnung verwendet werden (siehe unten stehenden Quelltext in Zeile 2) entstand durch den Umstand, dass ein Plug-In mehrere Abhängigkeiten definieren kann. Jede Klasse kann jedoch nur mit einer Annotation vom selben Typ versehen werden. Der Ansatz die Annotation um ein String-Array als Parameter zu erweitern, welches mit den jeweiligen Abhängigkeiten befüllt wird, und damit die Klasse zu versehen hätte eine Alternative dargestellt. Einerseits hätte dies den Vorteil, dass die Annotation auch in die JavaDocs übernommen würde, da die Plug-Ins öffentliche Klassen sind wohingegen die Felder nicht öffentlich sind und somit meist nicht in die Dokumentation einfließen. Andererseits wäre dieser Ansatz fehleranfälliger, da die Definition über den zusätzlichen Parameter separat als String angegeben wird. Ändert sich der Name der Abhängigkeit muss dieser eigens im Parameter angeglichen werden, was bei dem Ansatz ohne Parameter nicht der Fall ist.

Ein weiterer kleiner Nachteil ist, dass für jede Abhängigkeit eine Membervariable existieren muss. Die vorgesehene Verwendung der Abhängigkeiten ist, dass diese vor dem Aufruf der Methode `prepare()` im Plug-In gesetzt und gespeichert werden. Somit können die Abhängigkeiten bereits dem Konstruktor des Moduls übergeben werden. Besteht nun die Möglichkeit das Objekt des Moduls ohne die Abhängigkeiten zu erzeugen, könnten diese über die Setter-Methoden direkt an das Modul weiter gereicht werden. Die Definition der Membervariablen müsste jedoch dennoch erfolgen, auch wenn sie somit nicht benutzt würden.

Die über die Annotation zur Verfügung gestellte Meta-Information wird zur Laufzeit vom `DependencyManager` ausgewertet, weshalb von den drei Möglichkeiten (Source, Class, Runtime) nur `RetentionPolicy.RUNTIME` (siehe Zeile 1 des folgenden Quelltexts) verwendet werden kann. Die vollständige Implementierung der Annotation beschränkt sich auf die folgenden Zeilen:

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.FIELD)
3 public @interface Dependency {}
```

Die Definition und Injektion von Abhängigkeiten

Der `DependencyManager` ermittelt den Typ des gewünschten Moduls über den Datentyp der Membervariable, welche mit der Annotation `@Dependency` versehen wurde. Zusätzlich zur Kennzeichnung muss für jede Abhängigkeit eine entsprechende Setter-Methode vorhanden sein, damit diese injiziert werden kann. Der Name für die Setter-Methode entspricht der in Java üblichen Konvention. Die derzeitige Implementierung wurde *case-insensitive* gewählt um die Anfälligkeit auf Fehler zu reduzieren.

Ein Beispiel für die Deklaration von Abhängigkeiten zu Modulen und zum Kern zeigt das Programm 4.3. In diesem Beispiel fordert das Plug-In sowohl eine Referenz auf den `Core` als auch auf den `GameObjectManager` an. Dies zeigt auch, dass die Abhängigkeiten zum `Core` gleich denen zu regulären Modulen gehandhabt werden.

Um dies zu ermöglichen, registriert der Kern für sich selbst ein Plug-In (das `CorePlugIn`) wodurch er sich selbst als Modul verwaltet und ohne weitere aufwändige Implementierungen verteilt werden kann. Die Alternative, welche in einem frühen Stadium der Entwicklung des Projektes auch verwendet wurde, war eine eigene Annotation für die Abhängigkeit zum Kern. Dies erwies sich jedoch als umständlich, da an mehreren Stellen nicht nur nach der Annotation `@Dependency` gesucht werden musste, sonder auch noch der für den Kern, wodurch sich später dieser vereinfachte Ansatz durchsetzte.

Dieser Ansatz profitiert stark von der Verwendung des Klassen-Objekts als Identifikator der Module, da dieser automatisch vom Quellcode extrahiert werden kann. Ein zusätzlicher Vorteil ist, dass bei Änderungen der Abhängigkeiten diese sofort mit übernommen werden. Die Modifikatoren (z. B. `private` oder `static`) der Membervariablen haben auf die Funktionsfähigkeit des Systems keinen Einfluss. Lediglich die Setter-Methode sollte öffentlich sein.

Die Abfrage der Abhängigkeiten erfolgt während der Ermittlung der Startreihenfolge und während der Injektion der Referenzen. Dabei werden mittels Reflection die Membervariablen der Klasse des jeweiligen Plug-Ins explizit auf die Annotation `Dependency` hin überprüft, gesammelt und für die weitere Abarbeitung zurück gegeben. Den Quelltext für die Ermittlung der gekennzeichneten Felder zeigt Programm 4.4.

Zur Injektion der Abhängigkeiten ermittelt der `DependencyManager` zuerst alle Membervariablen die Abhängigkeiten definieren. Danach sucht er für eine jede die dazugehörige Setter-Methode und bezieht den Typ der Variable das Modul auf welches sich die Abhängigkeit bezieht. Zum Abschluss wird über Reflection die Setter-Methode aufgerufen und die Referenz gesetzt. Den beschriebenen Quelltext zeigt Programm 4.5 mit Ermittlung des Moduls über den Typ der Membervariable in Zeile 10.

Programm 4.3: Beispiel für die Definition von Abhängigkeiten.

```
1 public class ExamplePlugIn implements PlugIn<Example> {
2
3   @Dependency private Core core;
4   @Dependency private GameObjectManager manager;
5
6   public void setCore(Core core) { this.core = core; }
7
8   public void setManager(GameObjectManager manager) {
9     this.manager = manager;
10  }
11 }
```

Programm 4.4: Die Methode zur Ermittlung der Abhängigkeiten.

```
1 private List<Field> getDependencyFields(Class<?> clazz){
2
3   List<Field> dependencies = new LinkedList<Field>();
4   Field[] fields = clazz.getDeclaredFields();
5
6   for (Field field : fields) {
7     if (field.isAnnotationPresent(Dependency.class)) {
8       dependencies.add(field);
9     }
10  }
11
12  return dependencies;
13 }
```

Programm 4.5: Quelltext zur Injektion der Abhängigkeiten.

```
1 protected void injectDependencies(PlugIn<?> plugIn, List<PlugIn<?>>
      availablePlugIns) {
2
3     List<Field> dependencies = getDependencyFields(plugIn.getClass());
4     Method setterMethod = null;
5     Object dependency = null;
6
7     //try
8     for (Field field : dependencies) {
9       setterMethod = getSetterMethod(plugIn.getClass(), field);
10      dependency = getModule(field.getType(), availablePlugIns);
11      setterMethod.invoke(plugIn, dependency);
12    }
13    //catch
14 }
```

4.2.4 Ermittlung der Startreihenfolge

Eine Aufgabe des Plug-In Mechanismus ist die Ermittlung der Reihenfolge in denen die Plug-Ins gestartet werden. Diese ergibt sich aus den definierten Abhängigkeiten. Die Startreihenfolge darf nicht mit jener Reihenfolge verwechselt werden, in denen später die Module ihre Aufgaben (innerhalb der Spielschleife) erledigen. Einziges Kriterium für die Reihenfolge beim Starten ist, dass Plug-Ins erst im Anschluss, an die Plug-Ins zu denen sie Abhängigkeiten definieren, behandelt werden. Die konkrete Reihenfolge ändert sich bei unterschiedlichen Plug-Ins.

Der Algorithmus zur Ermittlung

Der Algorithmus zur Ermittlung der Startreihenfolge benötigt zwei Listen (sortiert und unsortiert) sowie einen Stack für die aktuell bearbeiteten Plug-Ins. Zu Beginn enthält die unsortierte Liste die Plug-Ins in der Reihenfolge ihrer Registrierung und die unsortierte Liste sowie der Stack sind leer.

Im folgenden Beispiel gegeben sind die Liste *U* für die unsortierten, die Liste *S* für die sortierten und der Stack sowie Plug-In *P1* bis *P4*. Die Plug-Ins *P1* und *P3* weisen keine Abhängigkeiten auf, Plug-In *P2* hat eine Abhängigkeit zu *P1*. Das Plug-In *P4* hat zwei Abhängigkeiten zu *P1* und *P2*. Der Ablauf wird in Abbildung 4.1 (a) dargestellt.

Da Plug-Ins ohne Abhängigkeiten jederzeit und ohne Vorbedingung vorbereitet bzw. gestartet werden können, werden diese im ersten Schritt aus der Liste *U* in die Liste *S* verschoben. Die Reihenfolge in der Liste *S* entspricht dabei der Reihenfolge aus Liste *U*. Somit befinden sich nur mehr Plug-Ins mit Abhängigkeiten in Liste *U*. Der `DependencyManager` nimmt das erste Element aus Liste *U* (*P2*) und legt es auf den Stack. Bei der Suche nach Abhängigkeiten wird zuerst in der Liste *S* gesucht. Da sich *P1* bereits in *S* befindet und *P2* keine weiteren Abhängigkeiten aufweist, wird *P2* nun ebenfalls nach *S* verschoben. Nun wird das verbleibende Plug-In *P3* auf den Stack gelegt. Alle Abhängigkeiten befinden sich bereits in Liste *S*. Somit kann auch *P3* eingeordnet werden. Die vollständige Reihenfolge ist somit *P1*, *P3*, *P2*, *P4*.

Zur Demonstration der Erkennung von ungültigen Abhängigkeiten wird nun *P3* um eine Abhängigkeit zu *P4* erweitert (Abb. 4.1 (b)). Die anfänglichen Schritte übersprungen befindet sich bei der Bearbeitung von *P3* in Liste *U* noch *P4*, auf dem Stack liegt *P3* und in Liste *S* befinden sich *P1* und *P2*. Da *P3* eine Abhängigkeit zu *P4* aufweist und dieses noch nicht in die Startreihenfolge eingegliedert wurde, wird dieses an die oberste Stelle des Stacks gelegt. *P4* sucht nun erfolglos nach *P3* in den Listen *S* und *U* wodurch ein Problemfall erkannt wird. *P3* befindet sich am Stack womit eine zirkuläre Abhängigkeit besteht die nicht aufgelöst werden kann. Befindet sich *P3* auch nicht am Stack wurde das gesuchte Plug-In nicht registriert.

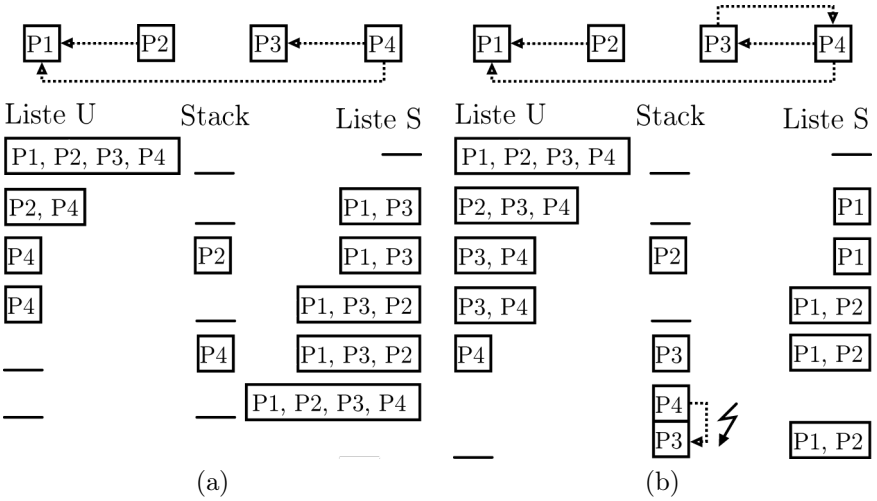


Abbildung 4.1: Darstellung der Ermittlung der Startreihenfolge mittels zwei Listen und einem Stack für die aktuell behandelten Plug-Ins. Einmal mit gültigen Abhängigkeiten (a) und einmal mit einer nicht auflösbaren zirkulären Abhängigkeit (b).

4.2.5 Der eingebaute Schutzmechanismus

Der Mechanismus dient zum Schutz der registrierten Plug-Ins und somit zur Gewährleistung des korrekten Betriebs. Er verhindert das Hinzufügen und Entfernen von Plug-Ins zur Laufzeit, da sich dadurch die Startreihenfolge ändern kann.

Die erste Maßnahme zum Schutz der Plug-Ins, ist die Einführung der Status des Kerns (siehe Tabelle 4.1) und die damit verbundene Limitierung des Hinzufügens und Entfernens auf den ersten und letzten Status. Dazu erfolgt bei jedem Aufruf einer entsprechenden Methode eine Kontrolle welcher Status aktuell vorliegt und ob eine Modifikation zulässig ist. Dies garantiert, dass sich die anfänglich ermittelte Startreihenfolge während dem Betrieb nicht ändert. Abgedeckt ist auch die indirekte Änderung über den Zugriff auf die Liste der Plug-Ins. Diese wird nur schreibgeschützt nach außen ausgegeben (mittels `Collections.unmodifiableList(List<? extends T> list)`).

Zum Teil trägt auch die Einführung der `Application` und dem dazugehörigen Framework des Kerns zum Schutz bei. Dadurch wird die sinnvolle Registrierung von Plug-Ins auf eine Methode im gesamten Spiel begrenzt was eine sauberere Implementierung unterstützt. Eine Steigerung des Schutzes wäre durch die Erweiterung des Frameworks möglich. Dabei würde die Applikation lediglich die zu registrierenden Plug-Ins übergeben und die Registrierung wird vom Kern übernommen. Darauf wurde verzichtet um die Architektur nicht zu unflexibel gegen Erweiterungen zu machen.

4.2.6 Die Initialisierung einer Applikation

Für die Initialisierung einer Applikation stellt der `Core` ein Framework zur Verfügung welches den korrekten Ablauf erleichtern soll. Um dieses Framework verwenden zu können, muss die Applikation das Interface `Application` implementieren.

Die eigentliche Initialisierung erfolgt über den Kern durch den Aufruf der Methode `initializeApp(Application app)`. Dadurch werden die Methoden der Applikation in ihrer korrekten Reihenfolge aufgerufen und zum richtigen Zeitpunkt die Startreihenfolge ermittelt. Der Ablauf beinhaltet die gesamte Lebensdauer der Applikation bis zu ihrem Beenden. Dies muss auch ein Modul für die Spielschleife (engl. game loop) berücksichtigen. Wenn diese in einem eigenen Thread betrieben wird, muss der Haupt-Thread (in dem die Applikation gestartet wird) geblockt werden. Ist dies nicht der Fall, werden sofort nach dem Starten der Plug-Ins diese wieder gestoppt und heruntergefahren. In Verbindung mit Java tritt dieses Problem z. B. bei der Verwendung der OpenGL-Anbindung *JOGL* auf, da dort im Hintergrund ein weiterer Thread gestartet wird (der `Animator`).

4.2.7 Die Verwaltung der Plug-Ins

Die An- bzw. Abmeldung eines Plug-Ins erfolgt direkt über den Kern. Dafür stehen die Methoden `registerPlugIn(PlugIn<?> plugIn)` und (für die Abmeldung) `unregisterPlugIn(PlugIn<?> plugIn)` zur Verfügung. Das Entfernen durch die Übergabe der Instanz ist in der Handhabung etwas umständlich, da in der Regel außerhalb des Kerns keine Referenz darauf gespeichert ist. Diese muss erst vom Kern bezogen werden.

Bei der Abfrage der Plug-Ins und Module finden die als `Class<T>`-Objekt realisierten Identifikatoren Verwendung (siehe Abschnitt 4.2.2). Diese Funktionalität wird von den Methoden `requestModule(Class<?> clazz)` und `requestPlugIn(Class<PlugIn<?>> clazz)` bereitgestellt. Zu Berücksichtigen gilt es, dass die Anfrage von Modulen erst nach der `prepare`-Phase des Lebenszyklus erfolgen darf, da zuvor nicht gewährleistet ist, dass die Module einsatzfähig sind. Zusätzlich ist es bei der häufigen Verwendung eines Moduls sinnvoll, diese nicht immer neu abzufragen sondern eine Referenz darauf zu speichern. Die derzeitige Implementierung durchsucht jedes Plug-In ob es die gewünschte Klasse bereitstellt, wodurch häufige Abfragen unnötige Performance verursachen. An dieser Stelle hat die Implementierung noch Potenzial für Verbesserungen in dem man die Abfrage schneller gestaltet.

Zuletzt kann auch die gesamte Liste der Plug-Ins abgefragt werden. Die Sortierung der Liste entspricht zu Beginn der Reihenfolge der Registrierung der Plug-Ins und ab dem Status `preparing` des Kerns die finale Startreihenfolge.

4.3 Spielobjekt-Verwaltung

4.3.1 Architektur

Die in den folgenden Abschnitten näher erörterte Architektur ist die Implementierung des in Abschnitt 3.4 vorgestellten Modells. Ein UML-Diagramm der Architektur befindet sich im Anhang in Abbildung A.3.

`Component`

Die Komponente ist der kleinste Baustein aus denen sich Spielobjekte zusammensetzen. Zusätzlich können Spielobjekte auch noch untergeordnete Spielobjekte enthalten. Das `GameObject` benötigt für die Verwaltung seiner Komponenten (und untergeordneten Spielobjekte) lediglich die Methoden des Lebenszyklus. Dadurch bietet es sich an, die Definition des `Component` als Interface zu implementieren. Im Vergleich zur Umsetzung über Vererbung ist bei gleich bleibender Funktionalität eine bessere Trennung von Komponente und Spielobjekt möglich. Im Weiteren erlaubt dieser Ansatz auch dass weitere Objekte als Komponente realisiert und mit verwaltet werden. Dies ist bei der Realisierung eines Editors von Vorteil, da Elemente, die für die Verwendung des Editors notwendig sind (z. B. Gizmos) nicht zusätzlich verwaltet werden müssen. Vor der Veröffentlichung des Spiels können diese dann wieder entfernt werden.

`AbstractComponent`

Die `AbstractComponent` ist die Basisklasse aller Komponenten. Sie implementiert das Interface `Component` und stellt zusätzlich eine Reihe von weiterer Funktionalität zur Verfügung. Zu diesen Funktionalitäten gehört die Verwaltung der `ComponentChangedListener` sowie die Bereitstellung einer Methode, welche diese verständigt. An welcher Stelle die Listener aufgerufen werden obliegt den davon abgeleiteten Komponenten selbst.

Ebenso stellt sie über die notwendigen Methoden für die Verbindung mit dem zugehörigen `GameObject` bereit welche für die Verwaltung der Komponenten benötigt werden. Auch der Zugriff auf den `Core` wird hier bereits für alle Komponenten einheitlich geregelt. Die eingebettete Funktionalität für die Kommunikation wird ab Abschnitt 4.4.2 näher beschrieben.

`ComponentChangedListener`

Der Listener für die internen Veränderungen an einer Komponente kann dazu verwendet werden, passive Synchronisationen zwischen den Komponenten durchzuführen. Der gegenüberstehende aktive Ansatz wäre es die Synchronisation über die Spielschleife gesteuert jeden Frame durchzuführen.

`GameObject`

Das `GameObject` dient als Container für die Komponenten. Es speichert seinen eindeutigen Identifikator als String, welcher während der Erzeugung des Spielobjekts gesetzt wird. Im Weiteren hält jedes Spielobjekt eine Referenz auf den Manager von dem es verwaltet wird. Dies ermöglicht es nicht nur über den Manager den `Core` zu beziehen sondern auch direkt auf die Funktionalität des Managers zuzugreifen. Benötigt wird die Funktionalität des Managers beim der Zerstörung von untergeordneten `GameObjects` bzw. dessen selbst.

Die Verwaltung der Komponenten (und der untergeordneten Spielobjekte) erfolgt in einer einfachen Liste in der Reihenfolge ihrer Registrierung beim Spielobjekt. Anders als bei bisherigen komponentenbasierten Architekturen für Game-Engines besteht in der vorliegenden keine Hierarchie. Da auf eine automatisierte Verknüpfung der Komponenten verzichtet wurde und dafür eine zusätzliche Phase im Lebenszyklus eingeführt wurde, besteht auch kein Bedarf einer Hierarchie. Alle Komponenten sind gleichberechtigt und können miteinander (manuell) verknüpft werden.

Für das Taggen der Komponenten beinhaltet das Spielobjekt eine `Map` mit einem String (dem Tag) als Schlüssel und der getagten Komponente als Wert. Da nicht zwangsläufig jedes Spielobjekt Tags beinhaltet, wird die Sammlung erst bei der Registrierung des ersten Tags initialisiert.

Um es zu ermöglichen, dass Spielobjekte selbst verschachtelbar sind, muss das Interface `Component` implementiert werden. Um dies zu erleichtern wurden die Lebenszyklen so gestaltet, dass sich die Namen der Phasen gleichen. Dadurch müssen nur vier Methoden für den Lebenszyklus implementiert werden, anstelle von sieben (drei für den Lebenszyklus der Komponenten sowie vier für den des Spielobjektes). Der Manager macht zwischen normalen und untergeordneten Spielobjekten keinen Unterschied. Beide werden auf gleiche Weise vom Manager verwaltet, verfügen über einen eigenen Identifikator über den sie angesprochen werden können.

Für die Spielobjekte, die oft erzeugt und wieder zerstört werden müssen, steht ein *Pooling* Mechanismus zur Verfügung. Dieser muss einmal für jeden Typ beim Manager aktiviert werden. Die gepoolten Instanzen von Spielobjekten erhalten bei jeder Verwendung einen neuen Identifikator um Überschneidungen mit bisher verwendeten zu vermeiden. Da ein Spielobjekt mehrere Init-Methoden zur Verfügung stellen, welche auch zu unterschiedlichen Verknüpfungen der Komponenten führen können, durchlaufen alle gepoolten Spielobjekte vor ihrer neuen Verwendung den kompletten Lebenszyklus erneut.

Zusätzlich beinhaltet ein Spielobjekt auch noch Funktionalität für die Kommunikation. Die Einbettung der Kommunikation in das Komponenten-System wird in Abschnitt 4.4.2 erörtert.

GameObjectManager

Der GameObjectManager stellt die Funktionalität für die Erzeugung, Zerstörung, Abfrage sowie das Pooling und die automatische Erzeugung (über die GameObjectController) zur Verfügung. Dafür verwaltet er insgesamt vier Sammlungen.

Zwei der Sammlungen dienen zur Verwaltung der Spielobjekte. Einerseits wird jedes Spielobjekt in einer nach ihrer Erzeugung sortierten Liste gespeichert. Konkret wurde hierfür eine LinkedList gewählt, da diese für die häufigen Änderungen am geeignetsten ist und keine Suche nach beinhalteten Objekten erfolgt. Für einen schnelleren Zugriff auf ein konkretes Spielobjekt anhand des Identifikators, werden diese zusätzlich in einer Map<String, GameObject> gespeichert.

Da die Möglichkeit besteht, dass Spielobjekte von der Zerstörung aller Spielobjekte ausgeschlossen (als persistent markiert) werden, werden die persistenten Objekte in einer weiteren Liste gespeichert. Da in der Regel der Großteil der Objekte nicht persistent ist, wurde die Markierung über eine Liste im Manager gelöst. Dadurch benötigt nicht jede einzelne Instanz eines Spielobjekts eine eigene Membervariable dafür. Für die Verwaltung dieser Objekte, wurde die ArrayList gewählt, da häufiger Suchen darauf ausgeführt werden, als das sich die Liste beim Löschen oder Erzeugen neuer Instanzen ändert. Gegen die Verwaltung einer eigenen Liste für jeden Typ im dazugehörigen GameObjectController sprach die Implementierung des Löschvorganges. Für die derzeitige Implementierung wird kein Controller benötigt was bei einer Änderung bedeutet, dass dieser zusätzlich noch gesucht werden müsste.

Zusätzlich zu den Spielobjekten werden die GameObjectController verwaltet. Für die Speicherung wird eine Map verwendet um den Zugriff anhand des Typs des Spielobjektes zu gewährleisten. Die GameObjectController werden bei ihrer ersten Anforderung erzeugt, sofern sie bis dahin noch nicht vorhanden sind. Verwendung finden sie während der Erzeugung eines neuen Spielobjektes. Sollte kein Identifikator für das Spielobjekt übergeben werden, werden diese vom Controller bezogen und bei aktiviertem Pooling wird zuerst überprüft ob eine neue Instanz erzeugt werden muss, oder ob eine aus dem Pool genommen werden kann.

GameObjectController

Für jeden Typ eines GameObject wird automatisch vom Manager ein zugehöriger GameObjectController angelegt. Dieser übernimmt die automatische Erzeugung der Identifikatoren sowie das Pooling der Instanzen dieses Typs. In einer Architektur, welche die Erzeugung der Spielobjekte über *Factories* löst, kann er auch diese verwalten.

4.3.2 Die Implementierung eigener Komponenten

Bei der Erstellung eigener Komponenten gibt es eine Reihe von Dingen zu berücksichtigen. Angefangen von der Wahl, ob von `AbstractComponent` abgeleitet oder das Interface `Component` implementiert wird, dem richtigen Zeitpunkt zur Erzeugung von Objekten bis zum Löschen der richtigen Referenzen bei der Zerstörung.

Auf den folgenden Seiten werden die Möglichkeiten bei der Erstellung einer Komponente anhand eines Beispiels erklärt. Das Beispiel geht davon aus, dass es für die Verwendung mit dem Modul `Module` erzeugt wird. Um mit dem Modul zu arbeiten implementiert es das Interface `IExample` mit der Methode `doSomething()`. Bei der Entwicklung von Modulen ist es von Vorteil, nicht direkt mit den Komponenten zu arbeiten, sondern Interfaces bereitzustellen welche von den Komponenten implementiert werden. Dies ermöglicht es, das Modul auch für andere Architekturen anzuwenden. Den Quelltext für das Beispiel zeigt Programm 4.6.

Grundlegend ist der einfachste Weg um eine eigene Komponente zu erstellen von `AbstractComponent` abzuleiten. Diese stellt bereits alle Funktionalitäten zur Verfügung. Das Interface `Component` wurde zur Verfügung gestellt, damit innerhalb der Architektur auch Objekte aufgenommen werden können, deren Vererbungshierarchie es nicht ermöglicht von `AbstractComponent` abzuleiten. Die `AbstractComponent` stellt für alle Methoden bereits standardmäßige Implementierungen zur Verfügung. Die Implementierung wurde dennoch abstrakt gewählt um zu verhindern, dass sie als normale Komponente verwendet wird.

Die Methoden des Lebenszyklus

Zum Lebenszyklus gehören nicht nur die drei Methoden für jede Phase (siehe Abschnitt 3.4.5) sondern auch ein parameterloser Standard-Konstruktor. Dieser wird vom `GameObjectManager` benötigt um die Instanzen der Komponenten zu erzeugen. Die Verwendung von Konstruktoren ist nicht vorgesehen. Der Standard-Konstruktor kann jedoch dennoch verwendet werden um interne Objekte zu erzeugen. Im Zusammenhang mit der Erzeugung über den Manager und dem Pooling sind noch weitere Kriterien zu berücksichtigen. Der Manager erzeugt beim Erstellen des Pools die gewünschte Menge an Instanzen, welche Wahlweise auch Null sein kann. Dabei wird nur der Konstruktor aufgerufen. Sollte die Startmenge des Pools sehr groß sein und interne Objekte im Konstruktor erzeugt werden, befinden sich diese bereits vor ihrer Verwendung im Speicher. Beim Überschreiben der Methoden des Lebenszyklus müssen auch die Methoden aus `AbstractComponent` aufgerufen werden.

Das Zusammenspiel mit einem Modul

Die Registrierung beim zugehörigen Modul erfolgt während den Phasen *Init* oder *Linked*. Die Schnittstelle ist dabei vom konkreten Modul bzw. vom angeforderten Interface abhängig. Es gilt anzumerken, dass nicht jede Komponente zwangsläufig mit einem Modul in Verbindung steht. Häufig finden sich auch Komponenten die nur bestimmte Eigenschaften zur Verfügung stellen (z. B. die Position eines Spielobjektes). Diese Eigenschafts-Komponenten (engl. Property Components) werden nur vom Spielobjekt selbst verwaltet.

Im Beispiel stellt die Komponente eine Initialisierungs-Methode zur Verfügung, der ein Double-Wert übergeben wird (Programm 4.6 Zeile 7). Dieser soll nicht in der Komponente gespeichert werden, sondern wird vom Modul gemeinsam mit dem Identifikator des Spielobjekts während der Registrierung benötigt. Da sowohl der Kern als auch der Identifikator des Spielobjektes beim Aufruf einer der zur Verfügung gestellten Init-Methoden zur Verfügung stehen, kann das Problem der doppelten Datenspeicherung zum Teil gelöst werden (siehe Abschnitt 3.4.2).

Die eigentliche Registrierung erfolgt ab Zeile 9. Dazu wird über die Methode `getCore()` aus der `AbstractComponent` der Kern angefordert und über diesen das Modul bezogen. Eine Überprüfung ob das Modul zur Verfügung steht, ist nur dann notwendig, wenn alternative Möglichkeiten bestehen. Diese müssen dann separat angefordert werden. In diesem Beispiel steht keine Alternative zur Verfügung. Eine erfolglose Abfrage resultiert in einem entsprechenden Fehler im Kern. Es kann somit davon ausgegangen werden, dass das Modul zur Verfügung steht (oder das Spiel abgebrochen wurde).

Die Registrierung einer Komponente während der `linked()`-Methode ist dann notwendig, wenn externe Referenzen benötigt werden. Diese eingehenden Referenzen wurden im Modell der Komponente als *In* bzw. Eingänge bezeichnet (siehe Abbildung 3.4). Diese stehen erst während der *Linked*-Phase des Lebenszyklus zur Verfügung. Eine gleichzeitige Verwendung der Eingänge als auch eine direkte Weiterleitung während einer der Initialisierungs-Methoden ist nicht möglich. In diesem Fall muss das Modul eine Möglichkeit bieten, die Eingänge erst später zu setzen oder diese bei der Verwendung der Komponente nach ihrer vollständigen Konstruktion abfragen.

Da während dem Spiel das Modul von der Komponente keine Verwendung findet, wird darauf auch keine Referenz gehalten. Jede Komponente die sich bei einem Modul registriert und von diesem referenziert wird, muss sich während seiner Zerstörung auch wieder von dort entfernen. Dies geschieht in der Methode `destroy()` auf nahezu gleichem Weg wie die Registrierung.

Das Zusammenspiel mit den Modulen während der Lebenszyklen hat auch negativen Einfluss auf das Pooling der Spielobjekte. Die genauere Beschreibung findet sich in Abschnitt 4.3.4. Dennoch sprach die Flexibilität durch die Wahlmöglichkeit für die Beibehaltung dieses Wegs.

Die Ein- und Ausgänge einer Komponente

Für die direkte Kommunikation mit anderen Komponenten definiert das Modell Ein- und Ausgänge (*In* und *Out*). Die Verwendung von Referenzen ist dann von Vorteil, wenn sich dadurch häufige Updates einsparen lassen. Die referenzierten Klassen müssen dabei so gewählt werden, dass sie in jedem verwendeten Modul zur Verfügung stehen. Dies ist einerseits gegeben, wenn die Klasse von Java aus definiert ist (z. B. `Point`) oder sie andererseits aus einem der Hilfspakete (z. B. `jgc.math.Vertex2D`) stammt.

Um die Verlinkung zu ermöglichen, muss die Komponente für die Ein- bzw. Ausgänge eine entsprechende `get-` bzw. `set`-Methode anbieten über die sie vom Spielobjekt verknüpft werden können. Im Weiteren muss die Verfügbarkeit der Referenzen gewährleistet werden. Dazu ist es notwendig, das die *Getter*-Methode für einen Ausgang während der Verlinkung nicht `null` zurückliefert. Die Ausgänge müssen somit spätestens während einer der Initialisierungs-Methoden deklariert werden. Für die Eingänge gilt, dass eine Verfügbarkeit erst in der `linked()`-Methode gegeben ist. Die Vorgangsweise einer Komponente, wenn zu diesem Zeitpunkt keine gültige Referenz gesetzt wurde, kann unterschiedlich sein. Falls die Möglichkeit besteht, kann sie selbst Standard-Werte erzeugen oder es wird eine Fehlermeldung geworfen.

Die Verwendung des `ComponentChangedListener`

Im vorliegenden Beispiel wurde sowohl für den Ein- als auch den Ausgang eine Referenz auf eine Instanz der Klasse `Point` verwendet. Über diese ist es mehreren Komponenten möglich durch auslesen bzw. schreiben zu interagieren. Jedoch stellt die Komponente `AComponent` aus dem Beispiel eine Funktionalität zur Verfügung, welche dann verwendet werden soll, wenn sich der Eingang ändert. In jedem Frame zu überprüfen ob sich der Punkt geändert hatte, würde nicht nur häufige und unter Umständen unnötige Überprüfungen erfordern, sondern auch die Speicherung des alten Wertes.

Um dies zu vermeiden verwendet die fiktive Komponente (aus der die Referenz stammt) sowie `AComponent` den `ComponentChangedListener` der immer dann aufgerufen wird, wenn sich der `Point` ändert. `AComponent` verständigt ihre Listener über den Aufruf von `notifyChanged()` über Änderungen ihrerseits (Zeile 38). Für die Reaktion auf die Änderungen wird die Methode `changed()` überschrieben. Damit der Listener für mehrere Komponenten verwendet werden kann, wird die jeweilige geänderte Komponente mit übergeben. Die Auswertung welche Komponente sich geändert hat (falls das notwendig ist) muss jede Komponente selbst implementieren.

Programm 4.6: Beispielhafte Implementierung einer Komponente die sich bei einem Modul registriert. Zusätzlich bietet sie einen Ein- und einen Ausgang an und verwendet den `ComponentChangedListener`.

```
1 public class AComponent extends AbstractComponent implements IExample {
2
3   private Point in;
4   private Point out;
5
6   //Methoden des Lebenszyklus
7   public void init(double value) {
8     super.init();
9     Module module = (Module)getCore().requestModule(Module.class);
10    module.register(this, getParent().getId(), value);
11    in = new Point();
12  }
13
14  @Override
15  public void linked() {
16    syncAndFlipPoints();
17    super.linked();
18  }
19
20  @Override
21  public void destroy() {
22    Module module = (Module)getCore().requestModule(Module.class);
23    module.unregister(this);
24    super.destroy();
25  }
26
27  @Override
28  public void doSomething() {      //Methode des Interface IExample
29    //eigentliche Funktionalität der Komponente für das Modul
30  }
31
32  //Methoden für die Ein- und Ausgänge
33  public void setInPoint(Point in) { this.in = in; }
34  public Point getOutPoint() { return out; }
35
36  public void syncAndFlipPoints() {
37    out.setLocation(in.y, in.x);
38    notifyChanged();    //Aufruf eigener Listener
39  }
40
41  //Methode des ComponentChangedListener
42  @Override
43  public void changed(Component component) {
44    syncAndFlipPoints();
45  }
46 }
```

4.3.3 Die Erzeugung der Komponenten

Die Erzeugung der Komponenten wird über die Methoden des `GameObject` realisiert. Dies erlaubt es, die Erzeugung der Komponenten auf die Spielobjekte zu limitieren und das Komponenten-System nach außen zu verbergen. Zusätzlich bietet dies die Möglichkeit vor der Initialisierung das dazugehörige Spielobjekt zu setzen, was den Zugriff auf den Kern und den Identifikator des Spielobjektes erlaubt. Ebenso werden auf diesem Weg die Komponenten gleich dem Spielobjekt hinzugefügt. Erzeugt wird eine Komponente durch den Befehl:

```
1 ((AComponent)createComponent(AComponent.class)).init(1.1);
```

Das `GameObject` übernimmt hierbei nur die Erzeugung der Instanz der Komponente. Der Aufruf einer der entsprechenden Initialisierungs-Methoden erfolgt extern. Eine Möglichkeit zur Verbesserung des System ist die Vereinheitlichung der Erzeugung, damit diese ebenfalls automatisiert werden kann (siehe Abschnitt 3.4.7). Dadurch würde der Type-Cast sowie der externe Aufruf von `init()` entfallen. Die weiteren Methoden des Lebenszyklus einer Komponente werden vom `GameObject` zu einem späteren Zeitpunkt aufgerufen. Die eigentliche Erzeugung ist verhältnismäßig kurz und wenig spektakulär:

```
1 protected Component createComponent(Class<? extends Component> type) {
2     //try
3       Component component = type.newInstance();
4       component.setParent(this);
5       components.add(component);
6       return component;
7     //catch
8   }
```

An dieser Stelle ergibt sich auch noch eine weitere Möglichkeit zur Verbesserung und Erweiterung der Architektur. Betrachtet man die Häufigkeit mit denen Komponenten erzeugt und zerstört werden erkennt man, dass relativ häufig Module vom Kern angefordert werden, wenn sich die Komponenten zu diesen hinzufügen oder von diesen entfernen. Verbunden mit der Komplexität der Suche nach einem Modul ergibt dies den Ansatzpunkt für die Optimierung. Eine Art *Dependeny Injection* für die Komponenten, bei der einmal definiert wird, welche Module in einem Komponenten-Typ benötigt werden, könnte die Suchanfragen reduzieren. Die Implementierung scheiterte bisher an der Notwendigkeit, dabei die Referenzen auf die Module in den Komponenten zu speichern was soweit wie möglich vermieden werden wollte. Ebenso würde die Suche nach dem Modul nur durch die Suche nach dem Typ der Komponente ersetzt werden.

4.3.4 Die Implementierung von Spielobjekten

Einfache Hierarchien

Die Entwicklung von Spielobjekten erfolgt im Rahmen des komponentenbasierten Systems durch die Zusammenstellung (Komposition) der gewünschten Funktionalität aus den vorhandenen Komponenten. Durch die Interaktion der Komponenten in der Spielwelt ergibt sich das gewünschte Spielobjekt.

Die Implementierung eines einfachen Spielobjektes soll hier anhand eines Beispiels erklärt werden. Dabei soll ein Spielobjekt erstellt werden, welches ein Bild über den Bildschirm bewegt (die Art der Bewegung ist dabei nicht näher spezifiziert). Den Quelltext zu dem folgenden Beispiel zeigt Programm 4.7.

Der erste Schritt, um in der vorliegenden Architektur ein Spielobjekt zu implementieren, ist das Anlegen einer eigenen Klasse die von `GameObject` abgeleitet wird. In der vorliegenden Architektur ist die Erstellung einer eigenen Klasse zwingend notwendig. Dies bietet sowohl Vor- als auch Nachteile. Der erste Vorteil ist der Erhalt der Übersichtlichkeit im Projekt. Da für jede Komposition eine eigene Klasse existiert, besteht während der Entwicklung großer Projekte nicht die Gefahr, dass Spielobjekte versteckt in einem großen Pool an Skripten oder Logik-Klassen erstellt werden. Im Weiteren ist somit für jedes Spielobjekt eindeutig geregelt wo sie bestimmte Funktionalitäten anbietet. Angenommen eine Logik-Komponente fordert ein Spielobjekt an und will den Namen des Spielers ermitteln. Ohne die Kapselung in einer Klasse müsste die Logik-Komponente die Komposition des Spielobjektes kennen, sich die richtige Komponente suchen und von dort den Namen auslesen. Ändert sich die Komposition muss auch die Logik-Komponente geändert werden. Im verwendeten Ansatz ist es möglich, eine Methode `getName()` zu erstellen welche den Namen aus der Komponente ausliest und zurückliefert. Ändert sich die Komposition, muss nur die Methode geändert werden. Der sich daraus ergebende Nachteil ist, dass es nicht ohne weiteres möglich ist, komplett neue Spielobjekte zur Laufzeit des Spieles zu erzeugen. Um dies zu ermöglichen bedarf es eines speziellen Spielobjektes, das öffentliche Methoden zum hinzufügen von Komponenten hat.

Als nächstes werden die Komponenten innerhalb der `init()`-Methode erzeugt und zum Spiel hinzugefügt. Die Details zur Erzeugung einer Komponente befinden sich in Abschnitt 4.3.3. Im Beispiel für das sich bewegende Bild werden vier Komponenten benötigt. Die Anzahl sowie der Typ der Komponenten ist von den jeweiligen Implementierungen abhängig. Die `Position2D`, das `ImageVisual2D` und die `MovingLogic` werden gleichzeitig mit einem Tag versehen über den sie später auch wieder aufrufbar sind. Die `ResourceImage`-Komponente wird ohne Tag erzeugt und ist somit nur über ihren Index erreichbar. Wenn sich die Komposition eines Spielobjektes noch oft ändert, ist es von Vorteil Tags zu verwenden.

Programm 4.7: Implementierung eines Spielobjektes für ein sich selbständig bewegendes 2D-Bild. Durch die Komponenten lässt sich innerhalb eines Spielobjekts das *MVC*-Pattern realisieren. Position, Bild und Logik werden mit Tags hinzugefügt und bei der Verknüpfung über diese angefordert. Die Bild-Resource wird über ihren Index verwaltet.

```
1 public class MovingImage extends GameObject {
2
3   public void init(double x, double y) {
4     ((Position2D)createComponent(Position2D.class, "M")).init(x,y);
5     ((ImageVisual2D)createComponent(ImageVisual2D.class, "V")).init();
6     ((MovingLogic)createComponent(MovingLogic.class, "C")).init();
7     ((ResourceImage)createComponent(ResourceImage.class)).init("name");
8   }
9
10  @Override
11  public void link() {
12    Position2D pos = (Position2D)getTagged("M");
13    ImageVisual2D vis = (ImageVisual2D)getTagged("V");
14    MovingLogic logic = (MovingLogic)getTagged("C");
15    ResourceImage res = (ResourceImage)getComponent(3);
16
17    logic.setPosition(pos.getPosition());
18    vis.setPosition(pos.getPosition());
19    vis.setResource(res.getResource());
20  }
21 }
```

Die Komponenten `Position2D`, `ImageVisual2D` und `MovingLogic` zeigen hier, dass sich das *Model-View-Controller*-Pattern auch innerhalb der Spielobjekte realisieren lässt. Die `Position2D` ist dabei das Modell und die `MovingLogic` der Controller, der die Position steuert. Durch die Referenzen wird das `ImageVisual2D` als View immer die aktuelle Position verwenden.

An dieser Stelle noch einige kurze Anmerkungen zum Zusammenspiel der Komponenten `ImageVisual2D` und `ResourceImage` sowie einer Eigenheit von Java. Das Visual benötigt zum korrekten Rendern nicht nur die Position sonder auch ein Bild in Form eines `Image` (der Prototyp des Renderings wurde für *Java2D* entwickelt). Da das Visual nur die Aufgaben des Renderings übernehmen soll wurde das Laden des Bildes in die Komponente `ResourceImage` ausgelagert. Dies ermöglicht eine saubere Trennung der Module des Renderings und der Ressourcen-Verwaltung. Als Schnittstellte dient die Referenz auf die Klasse `Resource<Image>`. Würde das Visual direkt das Bild referenzieren, würde die Ressourcen-Verwaltung die Möglichkeit verlieren, dieses aus dem Speicher zu entfernen. Die Resourcen-Verwaltung kann die Referenz nicht löschen, wodurch sie trotz *Garbage Collector* am Leben gehalten würde.

Die letzten Aufgaben während der Erstellung ist das Verknüpfen der Komponenten. Im hier präsentierten System, wird die Verknüfpung der Komponenten vom Entwickler des Spielobjektes übernommen. Auf eine automatisierte Verknüpfung wurde verzichtet (siehe Auflösung der Abhängigkeiten in Abschnitt 3.4.4). Dafür werden die Referenzen direkt über entsprechende *Getter*- und *Setter*-Methoden gesetzt. Welche Ein- und Ausgänge zur Verfügung stehen ist abhängig von den konkreten Komponenten. Ein Schritt dies im Quelltext leichter ersichtlich zu machen, ist eine Namenskonvention, bei der jede Setter-Methode für einen Eingang nicht mit „set“ sonder mit „in“ beginnt bzw. „out“ für alle Ausgänge. Da dafür keine zwingende Notwendigkeit besteht, wurde bisher darauf verzichtet.

Das Überschreiben der Methode destroy() ist in diesem Beispiel nicht notwendig. Alle abschließenden Aufgaben bei der Zerstörung eines Spielobjektes werden von den Komponenten selbst übernommen. Notwendigkeit ergibt sich in bestimmten Fällen bei der Erstellung von untergeordneten Spielobjekten. Da diese das Verhalten von Komponenten simulieren können, müssen sie sowohl Ein- bzw. Ausgänge zur Verfügung stellen können. Entsteht dadurch die Notwendigkeit die Verbindungen nach außen beim Zerstören wieder zu kappen, ist dies durch das Überschreiben von `destroy()` möglich.

Untergeordnete Spielobjekte

Untergeordnete Spielobjekte werden ähnlich der Komponenten direkt über das `GameObject` erzeugt und hinzugefügt. Dies erfolgt mittels folgendem Befehl:

```
1 ((MovingImage)createGameObject(MovingImage.class)).init(69, 69);
```

Auf diesem Weg erzeugte Spielobjekte werden dabei vollständig über den `GameObjectManager` erzeugt. Dies gewährleistet, dass sie bei der Initialisierung bereits über einen Identifikator verfügen und über den Manager erreichbar sind. Bevor sie zur Initialisierung zurückgeliefert werden erhalten sie noch die Referenz auf das übergeordnete Spielobjekt.

Pooling

Für Spielobjekte die häufig zerstört und neu erzeugt werden (z. B. die Kugeln eines MGs) besteht die Möglichkeit des Poolings. Jedoch werden zurzeit nur die Instanzen des Spielobjektes gepoolt, aber nicht die intern vorhandenen Komponenten. Dies ergibt sich aus dem Umstand, dass ein gepooltes Spielobjekt vor ihrer erneuten Verwendung den Lebenszyklus durchlaufen muss. Nur so können sich die Komponenten wieder bei ihrem Modul registrieren. Würden sich diese außerhalb des Lebenszyklus bei ihrem Modul an und wieder abmelden, könnten auch die Instanzen der Komponenten gepoolt werden, was die Effektivität des Poolings erhöhen würde.

4.3.5 Die Erzeugung von Spielobjekten

Die Erzeugung von Spielobjekten erfolgt über den `GameObjectManager` unter Angabe des gewünschten Typs. Dafür stellt der Manager zwei Methoden mit unterschiedlichen Parametern zur Verfügung. Eine für die Erzeugung mit einem selbst definierten Identifikator und eine weitere, welche den Identifikator automatisch generiert. Den Quelltext für die Erzeugung zeigt Programm 4.8. Die Fehlerbehandlung sowie Überprüfungen auf gültige Werte (z. B. doppelte Identifikatoren) wurde weggelassen.

Damit die Erzeugung eines Spielobjektes auf diesem Weg möglich ist wird ein öffentlicher Standard-Konstruktor vorausgesetzt. Über diesen wird die Instanz des Spielobjektes erzeugt. Während der `GameObjectManager` die Instanz vorbereitet, ist diese in einem kritischen Zustand, da der Lebenszyklus noch nicht angelaufen ist. Das Spielobjekt beinhaltet somit auch noch keine Komponenten. Dieser Umstand sprach dafür, die Erzeugung sowie Vorbereitung in einem Schritt durchzuführen. Die potenzielle Fehlermöglichkeit in diesem kritischen Zustand wird somit zum Teil ausgeschaltet. Ganz lässt sich diese erst vermeiden, wenn innerhalb dieser Methode auch der komplette Lebenszyklus kontrolliert wird (vereinheitlichte Initialisierungs-Methode).

Um die Möglichkeit des Poolings zu berücksichtigen, wird vor der Initialisierung der Instanz erst überprüft, ob das Pooling aktiviert ist und indirekt ob ein Objekt aus dem Pool entnommen werden kann. Für gewöhnlich erzeugen die Pools selbst neue Instanzen falls keine existierenden mehr vorhanden sind. In diesem Fall wurde darauf verzichtet um Quelltext-Verdoppelungen zu vermeiden. Der Pool müsste dieselbe Methode implementieren oder wiederum eine Methode des Managers aufrufen. Dadurch war es einfacher eine Überprüfung einzuführen und bei Bedarf die Instanz direkt im Manager zu erzeugen.

Die vorbereitenden Schritte, welche für den korrekten Ablauf des Lebenszyklus getroffen werden müssen, erfolgen direkt nach der Initialisierung. Zuerst wird im `GameObject` die Referenz auf den Manager gesetzt. Über den gesetzten Manager ist es möglich den Kern anzufordern (die Methode `getCore()` im `GameObject` dient dabei der Bequemlichkeit). Zusätzlich wird der Manager für die Erzeugung und vollständige Registrierung untergeordneter Spielobjekte benötigt.

Der nächste Schritt ist die automatische Erzeugung des Identifikators über den `GameObjectController` sowie dessen Vergabe an das Spielobjekt. Wird die Möglichkeit geboten, hier selbstdefinierte Identifikatoren zu verwenden, muss zusätzlich eine Überprüfung auf Eindeutigkeit erfolgen. Erhalten zwei Spielobjekte den gleichen Identifikator, würden sich diese bei der derzeitigen Implementierung ohne Überprüfung einfach überschreiben (in Bezug auf die Erreichbarkeit). Beide wären in der Liste für die korrekte Erzeugungs-Reihenfolge vorhanden und die Zerstörung des ersten Spielobjektes hätte die Löschung des zweiten zur Folge.

Programm 4.8: Die Methode zur Erzeugung einer Instanz eines Spielobjekt aus dem `GameObjectManager`. Vor der Erzeugung einer neuen Instanz wird auf aktiviertes Pooling überprüft. Danach wird der Identifikator und die Referenz auf den Manager gesetzt.

```
1 public GameObject createGameObject(Class<? extends GameObject> type) {
2     //try
3       GameObjectController con = getGameObjectController(type);
4       GameObject go = null;
5
6       if (con.isPoolingEnabled()) {
7         go = con.grabFromPool();
8       }
9
10      if (go == null) {
11        go = type.newInstance();
12      }
13
14      go.setManager(this);           //für den Zugriff auf den Kern
15      go.setId(con.getId());         //Verfügbarkeit der ID
16      objects.put(go.getId(), go);   //Erreichbarkeit über den Manager
17      creationOrder.add(go);         //für die korrekte Reihenfolge
18      return go;
19    //catch
20 }
```

Zuletzt wird das Spielobjekt beiden Sammlungen hinzugefügt. Erst ab diesem Zeitpunkt ist das Spielobjekt tatsächlich über den Manager erreichbar. Die Verwaltung der Objekte in zwei Sammlungen kann durchaus als aufwendig betrachtet werden. Eine bessere Alternative, welche weniger Aufwand bei gleicher Funktionalität bietet, konnte bisher jedoch noch nicht gefunden werden.

Abschließend noch einige Anmerkungen zur Erzeugung und Zerstörung von Spielobjekten über den Manager. Der Manager ist so implementiert, dass die Erzeugung eines Spielobjektes sofort nach erfolgtem Aufruf der Init-Methode abgeschlossen hat. Das gleiche gilt für die Funktionalität beim Zerstören. Das Problem des *1-Frame-Delay* (Erzeugung erst im nächsten Frame) tritt in dieser Architektur nicht auf. Dieses Problem tritt bei Architekturen auf, in denen die Erzeugung der Spielobjekte durch Spielschleife gesteuert wird. Zum Fehlerfall kommt es dann, wenn eine Logik-Komponente zwei Spielobjekte erzeugt und diese verknüpfen will. Da die Spielobjekte noch nicht existieren, sondern erst im nächsten Frame erzeugt werden, wird ein Zugriff auf ein nicht existierendes Spielobjekt gemeldet. Daher garantiert die vorliegende Implementierung die sofortige Erzeugung des Spielobjekts beim Aufruf der Methode.

4.4 Kommunikation

Die in den folgenden Abschnitten näher erörterte Architektur ist die Implementierung des in Abschnitt 3.5 vorgestellten Modells. Ein UML-Diagramm der Architektur befindet sich im Anhang in Abbildung A.1.

4.4.1 Architektur

`Message<T>`

Die Grundlage für die Kommunikation über Nachrichten und Events bildet die Klasse `Message<T>`. Die Nachricht beinhaltet lediglich eine Membervariable vom Typ `String` für ihren Identifikator und eine vom jeweils gewählten generischen Typ für die Speicherung der zu übermittelnden Daten.

Sowohl der Identifikator als auch die Daten der Nachricht müssen während des Konstruktors übergeben werden. Da nur Methoden für den lesenden Zugriff öffentlich angeboten werden, ist die Nachricht an sich schreibgeschützt. Es kann jedoch keine Aussage darüber getroffen werden, ob die jeweilige Datenstruktur selbst veränderbar ist. Dieser Schreibschutz erschwert zum einen das ungewollte Verändern einer Nachricht während sie die Empfänger durchläuft und zum anderen eine fehlerhafte Anwendung (z. B. Austausch von Daten beim Versuch die Nachricht wiederzuverwenden).

Die Message beinhaltet keine Membervariable zur Speicherung des Empfängers. Dies ergibt sich daraus, dass sowohl bei der Kommunikation über Events als auch bei der Intra-Objekt-Kommunikation über Nachrichten keiner benötigt wird. Muss eine Nachricht an ein anderes Spielobjekt verschickt werden, so muss der Empfänger von der Komponente selbst verwaltet werden.

Der Identifikator für die Nachricht wurde als String gewählt um möglichst flexibel zu sein. Für jede Komponente die eine Nachricht verschickt bzw. erhält ist es zwingend notwendig, den Identifikator konfigurierbar zu machen. Dies ermöglicht eine reibungslose Konfigruation auch in komplexen Setups. Ein einfaches Beispiel dafür ist eine Komposition die zwei Komponenten vom Typ `TimedSlot` enthält. Jede der zwei Komponenten hört auf eine Nachricht um ihren internen Countdown zu starten. Wären die Identifikatoren der Nachrichten nicht konfigurierbar, wäre es nicht möglich beide getrennt voneinander zu starten.

Ein Nachteil des `String` als Typ ist die komplexe Vergleichbarkeit. Unter Berücksichtigung der Häufigkeit mit der Nachrichten verschickt und die Identifikatoren verglichen werden, ist dies durchaus als relevant für die Performance zu betrachten. Um diesen Nachteil auszugleichen wird auf Javas internen String-Pool zurückgegriffen. Sofern gewährleistet ist, dass die verglichenen Strings aus diesem Pool stammen, liefert ein Vergleich der Referenzen ein valides Ergebnis. Um diese Vergleiche zu ermöglichen steht die Methode `isId(String id)` zu Verfügung.

Ein weiterer Schritt um die Häufigkeit von verschickten Nachrichten zu kompensieren, ist das Pooling der Nachrichten. Da für die Nachrichten kein eigenes Modul zur Verfügung steht, beinhaltet die Klasse `Message` selbst einen statischen Pool sowie statische Methoden um sie aus diesem zu entnehmen oder wieder rückzuführen. Da keine öffentlichen Konstruktoren für die Nachrichten bereitstehen ist die einzige Bezugsmöglichkeit für Instanzen über diesen Pool. Die Rückführung in den Pool der verwendeten Instanzen obliegt dem Anwender. In der vorgestellten Implementierung wird dies durch die Einbettung in das Komponenten-System dem Anwender abgenommen.

`MessageHandler`

Der `MessageHandler` ist das Interface, welches implementiert werden muss, damit sich ein Objekt beim `EventManager` registrieren kann. Es definiert nur die Methode `handleMessage(Message<?> msg)`. Um die Fähigkeit zum Erhalt von Nachrichten und Events in der Engine einheitlich erkennbar zu machen, implementiert jedes Objekt das für den Erhalt von Nachrichten gedacht ist, dieses Interface.

`EventManager`

Der EventManager dient als Modul für den Versand von Nachrichten an mehrere dem Versender unbekannte Empfänger. Jeder Empfänger muss sich im Vorfeld für den Empfang beim Manager registrieren. Die Registrierung erfolgt unter Angabe des Identifikators der gewünschten Nachrichten. Für ein leichteres Verständnis, werden Nachrichten die über den EventManager versendet werden, als Events bezeichnet.

Um einen Event zu versenden benötigt der Sender lediglich Zugriff auf den EventManager. Der Manager ist gleich dem Modul für die Spielobjekt-Verwaltung so implementiert, dass der Versand der Events ohne Verzögerung erfolgt. Im Weiteren gilt, dass die Empfänger in der Reihenfolge ihrer Registrierung verständigt werden.

Ein weiteres praktisches Anwendungsbeispiel für die Problematik des *1-Frame-Delay* zeigt die Erweiterung der Problematik um ein zusätzliches verzögerndes Modul (genauer einem anders implementierten `EventManager`). Das Beispiel beinhaltet drei Logik-Komponenten (A,B,C). Komponente B ist beim `EventManager` für einen Event registriert und soll beim Erhalt ein Spielobjekt mit dem erhaltenen Identifikator erstellen. Komponente C wartet auf eine Nachricht mit dem Identifikator eines Spielobjektes um von diesem die Position auszulesen. Komponente A löst nun den Event aus und verschickt gleich im Anschluss die Nachricht zum Verknüpfen an Komponente C. Erzeugen beide Module je einen Frame Verzögerung schlägt dieser Versuch fehl da weder der Event verschickt noch das Spielobjekt erzeugt wurde. Daher ist es von Vorteil wenn die Module ihre Aufgaben gleich erledigen.

`Signal` und `Slot`

Für die Verwendung des *SnS*-Systems (*Signals and Slots*) bietet die Architektur die Klassen `Signal` und `Slot`. Dabei bezeichnen die Slots die Ausgänge und die Signals die Eingänge. Die Implementierung als Klasse von der abgeleitet werden muss, bietet gegenüber der Implementierung als reines Interface Vorteile.

Die Signals und Slots werden direkt über Referenzen miteinander verbunden. Dies bedeutet, dass eine Komponente die ein Signal anbietet, sich bei ihrer Zerstörung vom jeweiligen Slot entfernen muss. Grundsätzlich wäre es möglich, dass der Ersteller eines Spielobjektes nicht nur die Verknüpfung innerhalb der `link()`-Methode übernimmt, sondern auch wieder deren Trennung in der `destroy()`-Methode. Um diese Routine-Aufgabe dem Entwickler abzunehmen, wurde ein System gewählt, bei dem die Komponente, welche das Signal anbietet, dieses bei Bedarf auch wieder vom Slot entfernt. Zusätzlich wäre es über die Implementierung als Interface nicht möglich, dass eine Komponente mehrere Signals oder Slots anbietet.

Wird ein Signal nicht von ihrem Slot getrennt wenn die dazugehörende Komponente zerstört wird, kann der *Garbage Collector* von Java die Komponente nicht löschen. Wird nun der Slot aktiviert und das Signal verschickt, kann es dazu kommen, dass die Komponente weiter ihre Aufgabe ausführt (z. B. die Ausgabe einer Log-Nachricht). In der Regel wird es jedoch zu einem Fehler kommen und die Suche nach der Fehlerquelle gestaltet sich schwierig, da die Komponente in der Spielobjekt-Verwaltung nicht mehr angezeigt wird.

Für die Verwendung der Slots bietet die implementierte Klasse alle benötigte Funktionalität. Um einen bzw. mehrere Slots anzubieten muss eine Komponente lediglich mehrere Instanzen der `Slot`-Klasse beinhalten. Der Versand der Signale erfolgt über den Aufruf der Methode `send()`:

Die einfachste Möglichkeit zur Implementierung von Signals, der sich während der Entwicklung der Architektur gezeigt hat, war die Verwendung von anonymen Klassen. Für jedes angebotene Signal wird dazu eine Membervariable vom Typ `Signal` angelegt. Die Instanz der Klasse wird erst bei Bedarf in der Getter-Methode erzeugt (engl. lazy initialization). Die erzeugte Instanz ist eine anonyme Klasse, welche die Methode `signal()` überschreibt und in ihr die gewünschte Funktionalität bereit stellt.

Für die Trennung der Verbindung bei der Zerstörung der Komponente steht sowohl für das Signal als auch den Slot die Methode `unlink()` zur Verfügung. Jede Komponente welche diesen Kanal benutzt ist verpflichtet, die Trennung zu übernehmen.

4.4.2 Die Einbettung in das Komponenten-System

Die Einbettung der Kommunikation in das Komponenten-System bietet die Möglichkeit die benötigten Funktionen direkt in den Komponenten zur Verfügung zu stellen. Ebenso können allgemeine Aufgaben (z. B. in Verbindung mit dem Pooling der Nachrichten) zentral implementiert werden. Die Anfälligkeit auf Fehler des Systems sinkt.

Um den Erhalt von Nachrichten und Events zu ermöglichen implementieren sowohl die `AbstractComponent` als auch das `GameObject` das Interface `MessageHandler`. Die `AbstractComponent` stellt dabei lediglich eine leere Implementierung der Methode für den Empfang zur Verfügung, welche bei Bedarf überschrieben werden kann. Dies bietet auch den Vorteil, dass bei Bedarf die `AbstractComponent` selbst einmal Nachrichten empfangen kann.

Jede Komponente hat somit Zugriff auf Methoden zum Versand von Nachrichten und Events. Diesen Methoden wird nur der Identifikator und die Datenstruktur übergeben sowie bei Bedarf auch der Identifikator des Empfängers. Die Entnahme einer Instanz eine `Message` sowie deren Rückführung in den Pool wird dem Anwender abgenommen.

Zusätzlich stehen Methoden zur Registrierung für den Erhalt einer bestimmten Nachricht zur Verfügung. Dies verhindert, dass Nachrichten an Komponenten weitergeleitet werden, die für den Erhalt keine Funktionalität anbieten. Die Verteilung der Nachrichten an die für den Erhalt registrierten Komponenten übernimmt das `GameObject` in der dort implementierten `handleMessage(Message<?> msg)`-Methode. In der derzeitigen Implementierung sind keine *Broadcasts* vorgesehen, da bisher noch kein Anwendungsfall für diese gefunden werden konnte.

Ein Sonderfall tritt bei der Verwendung von untergeordneten Spielobjekten auf. Besteht ein Spielobjekt aus einer Komposition mehrerer untergeordneter Spielobjekte, kann jedes davon gleichwertig als Empfänger einer Nachricht angegeben werden. Die Spielobjekte leiten sowohl die Registrierungen für die Nachrichten als auch die Nachrichten selbst an das oberste Element der Hierarchie weiter. Erst das oberste Element übernimmt die tatsächliche Verteilung der Nachrichten. Auf diesem Weg kann sichergestellt werden, dass eine Nachricht, egal an welches Spielobjekt der Hierarchie sie geschickt wird, in der gesamten Komposition ausgeliefert wird. Bei der Erstellung von Komponenten die eine Nachricht an das eigene Spielobjekt versenden muss daher nicht berücksichtigt werden, ob es sich dabei um ein untergeordnetes Spielobjekt handelt oder nicht. Ohne diese Weiterleitung der Nachrichten müsste für jedes Spielobjekt die exakte Identifikator des Empfängers angegeben werden. Besonders in Ausblick auf eine Erweiterung, bei der nicht jedes untergeordnete Spielobjekt im Manager aufscheint, wäre dies von Nachteil.

4.4.3 Die Anwendung von Messages

In den folgenden Abschnitten wird die Anwendung der vorhandenen Arten für die Kommunikation an einem praktischen Beispiel erklärt. Dazu wird die Komponente `ForwardComponent` erstellt, deren Aufgabe es ist, jede eingehende Kommunikation auf demselben Kanal weiterzuleiten. Den vollständigen Quelltext zu den folgenden Anwendungsbeispielen zeigt Programm 4.9. Die Methoden zum Setzen der Identifikatoren wurden weggelassen.

Für den Erhalt von Nachrichten wird die `handleMessage`-Methode überschrieben (siehe Zeile 23). Im Beispiel werden nach einer Überprüfung um welche Nachricht es sich handelt die Daten extrahiert und mit einem neuen Identifikator versandt. Die erste ausgehende Nachricht erfolgt dabei ohne Angabe eines Empfängers und wird somit im eigenen Spielobjekt verteilt. Für die zweite wird vom `GameObjectManager` der Empfänger angefordert und diesem zugestellt. Die An- und Abmeldung von dem Erhalt der Nachricht erfolgt in der Init- und Destroy-Phase des Lebenszyklus.

4.4.4 Die Anwendung von Events

Die Handhabung der Events ist jener der Nachrichten ähnlich. Die An- und Abmeldung erfolgt ebenfalls während der Init- und der Destroy-Phase des Lebenszyklus. Anders als die Nachrichten erfolgt die Registrierung intern beim `EventManager`. Events und Nachrichten werden auf die gleiche Weise empfangen und auf ihren Identifikator überprüft. Unterschiedlich hingegen ist der Versand des Events. Dieser erfolgt stets ohne Angabe eines konkreten Empfängers, da die Weiterleitung ohnehin an alle registrierten Empfänger erfolgt.

4.4.5 Die Anwendung von Signals und Slots

Für die Anwendung von Signals und Slots muss eine Komponente je eine Instanz der gewünschten Klasse anbieten. Die Erzeugung des Slots erfolgt ab Zeile 33 und die des Signals ab Zeile 38. Während es für den Slot reicht, die vorhandene Klasse zu verwenden und dieser das zu verbindende Signal hinzuzufügen, ist die Erstellung eines Signals aufwendiger. Die Erstellung erfolgt, wie bereits erwähnt, über eine anonyme Klasse welche die Methode `signal()` mit der gewünschten Funktionalität überschreibt. Im Beispiel die Weiterleitung des Signals über den ausgehenden Slot. Sowohl das Signal als auch der Slot müssen in der Destroy-Phase korrekt zerstört werden (siehe Zeile 16 und Zeile 17.

Programm 4.9: Implementierung einer Komponente welche die eingehende Kommunikation auf demselben Kanal weiterleitet.

```
1 public class ForwardComponent extends AbstractComponent {
2
3   private Signal incoming;
4   private Slot outgoing;
5
6   private String inMsgID, outMsgID, receiverID;
7   private String inEventID, outEventID;
8
9   public void init() {
10    super.init();
11    registerForMessage(inMsgID);
12    registerForEvent(inEventID);
13  }
14
15  public void destroy() {
16    if (incoming != null) { incoming.unlink(); }
17    if (outgoing != null)  { outgoing.unlink(); }
18    unregisterFromMessage(inMsgID);
19    unregisterFromEvent(inEventID);
20    super.destroy();
21  }
22
23  public void handleMessage(Message<?> msg) {
24    super.handleMessage(msg);
25    if (msg.isId(inMsgID)) {
26      sendMessage(outMsgID, msg.getData());
27      sendMessage(outMsgID, msg.getData(), receiverID);
28    } else if (msg.isId(inEventID)) {
29      fireEvent(outEventID);
30    }
31  }
32
33  public void slotOutgoing(Signal signal) {
34    if (outgoing == null) { outgoing = new Slot(); }
35    outgoing.addSignal(signal);
36  }
37
38  public Signal signalIncoming() {
39    if (incoming == null) {
40      incoming = new Signal() {
41        public void signal() {
42          if (outgoing != null) { outgoing.send(); }
43        }
44      };
45    }
46    return incoming;
47  }
48
49 }
```

Kapitel 5

Schlussbemerkungen

In diesem Kapitel soll nun ausgewertet werden, inwieweit das beschriebene Modell und dessen Implementierung den an sie gestellten Anforderungen gerecht wird und ob die beschriebenen Probleme gelöst werden konnten. Abschließend soll auch noch ein Ausblick gegeben werden, über die Möglichkeiten zu Erweiterungen und Verbesserungen.

5.1 Ergebnisse

5.1.1 Modularität

Die gewünschte Modularität wird durch den Plug-In Mechanismus und den komponentenbasierten Ansatz der Spielobjekt-Verwaltung erreicht. Dadurch ist es möglich die in der Engine zur Verfügung stehende Funktionalität zu erweitern bzw. bei Bedarf zu reduzieren. Der damit verbundene Aufwand ist abhängig von der konkreten Modifikation und dem Zeitpunkt an dem diese durchgeführt wird.

Wird die Wahl der Module schon zu Beginn des Projektes festgelegt, kann die Zusammenstellung noch ohne großen Aufwand erfolgen. Ändert sich die Zusammenstellung jedoch zu einem späteren Zeitpunkt der Entwicklung, kann dies Auswirkungen auf die bereits erstellten Spielobjekte haben. Dieses Risiko wird durch die Verwendung einer komponentenbasierten Spielobjekt-Verwaltung gut abgefangen. Der Austausch von einzelnen Komponenten ist in der vorgestellten Engine einfach zu lösen.

Durch die Erweiterung um Abstraktionsschichten besteht die Möglichkeit, dass auch die Komposition der Objekte unverändert bleiben kann, sofern die Funktionalität gleich bleibt (siehe Abschnitt 3.3.7). Da jedoch jede Teil-Engine unterschiedliche Möglichkeiten bietet, sollte sorgsam abgewogen werden, welche Funktionalität die Abstraktionsschicht bieten soll.

5.1.2 Spielobjekt-Verwaltung

Die komponentenbasierte Spielobjekt-Verwaltung unterstützt nicht nur die Modularität der Engine. Die Wiederverwendbarkeit der einzelnen Komponenten und die unlimitierte Verwendung in einem einzelnen Spielobjekt (*Unity3D* erlaubt nur eine Komponente eines Typs in einem Spielobjekt) bietet auch die Möglichkeit zur raschen Entwicklung von komplexen Spielen. Die Möglichkeit Spielobjekte zu verschachteln steigert den Grad der Wiederverwendbarkeit zusätzlich. In einem Spiel oft auftretende Kompositionen von Spielobjekten können so einmal zusammengestellt und in mehreren anderen Spielobjekten verwendet werden. Dieser Teil der Architektur kann jedoch noch verbessert werden. So ist es nicht zwingend notwendig, dass die untergeordneten Spielobjekte über eine eigene ID beim Manager registriert sind. Dies erhöht die Menge der vom Manager verwalteten Objekte und kann in großen Welten mit zahlreichen Spielobjekten die Suche nach einem konkreten Objekt (z. B. für die Kommunikation) verzögern.

Die weiteren Funktionalitäten, wie die Fähigkeit Spielobjekte ineinander zu verschachteln, Pooling, Tagging von Komponenten oder die automatisierte Erzeugung der IDs für Spielobjekte, decken weitgehend den Bedarf für die Entwicklung ab. Eine Erweiterungsmöglichkeit wäre noch das Erstellen von Gruppen für Spielobjekte. Dadurch könnten mehrere Spielobjekte gleichzeitig angesprochen werden unabhängig davon, wie viele der Gruppe angehören.

Gut geplante Lebenszyklen der Spielobjekte sowie der Komponenten lösen bekannte Probleme (z. B. der Notwendigkeit der doppelten Datenspeicherung) teilweise oder ganz, da jederzeit Zugriff auf die gesamte Funktionalität der Engine besteht.

Kritisch zu betrachten bleibt der *Overhead* der durch die Komponenten entsteht. Bei einem PC mit mehreren Gigabyte an Arbeitsspeicher kein Problem darstellt, könnte auf mobilen Plattformen mit stark begrenzten Hardware-Ressourcen ins Gewicht fallen.

5.1.3 Kommunikation

Für die Kommunikation wurden bereits bewährte Konzepte wie Nachrichten und Events adaptiert und in die Architektur aufgenommen. Diese decken zum derzeitigen Stand alle Anwendungsfälle ab. Der Nachteil von Nachrichten ist bei der Verwendung im Quelltext jedoch der noch relativ hohe Konfigurationsaufwand. Versuche diesen teilweise zu reduzieren (z. B. durch fest vorgegebene Nachrichten-IDs) scheiterten dahingehend, dass die Kommunikation in komplexeren Kompositionen fehlschlug.

Für die reaktive Kommunikation wurden die Signals and Slots von *QT* adaptiert und bieten in der vorgestellten Architektur eine gute Alternative zu den Nachrichten. Die Stärke liegt in dem hohen Maß an Flexibilität, welche jedoch durch den datenlosen Charakter sowie einem relativ hohen Entwick-

lungsaufwand geschwächt wird. Im Gegensatz dazu ist dieser Kanal der direkteste, da für die Kommunikation keine Empfänger gesucht werden müssen. Am geeignetsten sind Signal and Slots für die Intra-Objekt-Kommunikation. Die Verwendung über die Grenzen von Spielobjekten hinaus ist zwar technisch gesehen möglich, birgt jedoch einige Risiken (z. B. zerstörte Spielobjekte welche noch immer verbunden sind) und Umwege für die Konfiguration.

5.1.4 Einfache Anwendbarkeit

Die einfache Anwendbarkeit der vorliegenden Implementierung (im Vergleich zu anderen Implementierungen) resultiert aus der geringen Menge an Quelltext die für die Lösung einer Aufgabe benötigt wird. Im Weiteren wurden besonders im Plug-In Mechanismus herkömmliche Aufgaben wie die Ermittlung der Startreihenfolge oder die Verwaltung der Abhängigkeiten soweit wie möglich automatisiert. Dadurch wird es möglich den Aufwand für den Bezug einer Abhängigkeit zu einem anderen Modul auf die Kennzeichnung mit einer Annotation und dem Bereitstellen einer Setter-Methode zu reduzieren. Immer wiederkehrende Fehlerabfragen sowie die Erstellung von Fehlermeldungen entfallen.

Auf eine Automatisierung für die Spielobjekt-Verwaltung (automatische Verknüpfung der Komponenten) wurde vorerst noch verzichtet, da diese eine Limitierung der Kompositionen zur Folge gehabt hätten. Bestehende Ansätze sowie eigene Versuche resultierten in Lösungen, welche jedoch stets einen Mehraufwand für die Erstellung eines Spielobjekts bewirkt haben.

5.2 Ausblick

Die vorgestellte Architektur sowie die dazugehörige Implementierung bilden eine gute Basis für weitere Projekte. Die nächsten Schritte, welche am vielversprechendsten sind, wurden in Abschnitt 3.3.7 und Abschitt 3.4.7 beschrieben. Darunter ist die Erweiterung zur *data driven* Game-Engine der beste Startpunkt. Sobald die Notwendigkeit, Quelltext zu schreiben, entfällt, kann die Engine ein breiteres Zielpublikum erreichen. Ebenso ist dies die Grundlage für die Entwicklung eines visuellen Editors, der die Erstellung der Daten erleichtert und den Produktionsprozess beschleunigt.

All diese Erweiterungen sind jedoch im Vergleich zu den Möglichkeiten der kommerziellen Engines für aktuelle AAA-Spiele noch Grundlagen. Die wahre Stärke einer Engine entsteht erst durch die Anzahl und Leistungsfähigkeit der vorhandenen Module, sowie speziell im komponentenbasierten Umfeld durch die vorhandenen Komponenten. Dadurch können auch die Bedürfnisse der großen Entwickler gedeckt und deren Interesse geweckt werden. Dabei ist es durch den modularen Aufbau der Engine nicht zwingend notwendig, alle Module intern zu entwickeln. Durch Kooperationen oder Vergabe von Aufträgen an spezialisierte Unternehmen, sowie durch die Anbindung

vorhandener Teil-Engines, kann die Produktion weiter erleichtert und beschleunigt werden.

Eine der wesentlichsten Fähigkeiten einer Game-Engine ist jedoch einmalige Entwicklung eines Spiels für mehrere Plattformen. Dies bietet dem Entwickler die Möglichkeit ein größeres Publikum mit seinem Spiel zu erreichen, was auch die Chancen auf eine positive Bilanz steigert. Die Stärke einer Architektur liegt somit in der Möglichkeit, diese auf mehreren Plattformen (und dadurch zwangsläufig in mehreren Sprachen) zur Verfügung zu stellen. Besonders die neuen Möglichkeiten von HTML 5 bieten hier ein interessantes Gebiet, da sich bis zu diesem Zeitpunkt noch keine Engine für diesen Bereich durchgesetzt hat.

Abschließend kann festgestellt werden, dass die Spieleentwicklung seit jeher spannende Herausforderungen geboten hat. Blickt man von den heutigen 3D-Grafiken zurück auf „Tennis for two“ bieten die kommenden Jahre und Jahrzehnte ein unglaubliches Potenzial von dem wir uns alle ein Stückchen abschneiden können.

Anhang A

UML-Diagramme

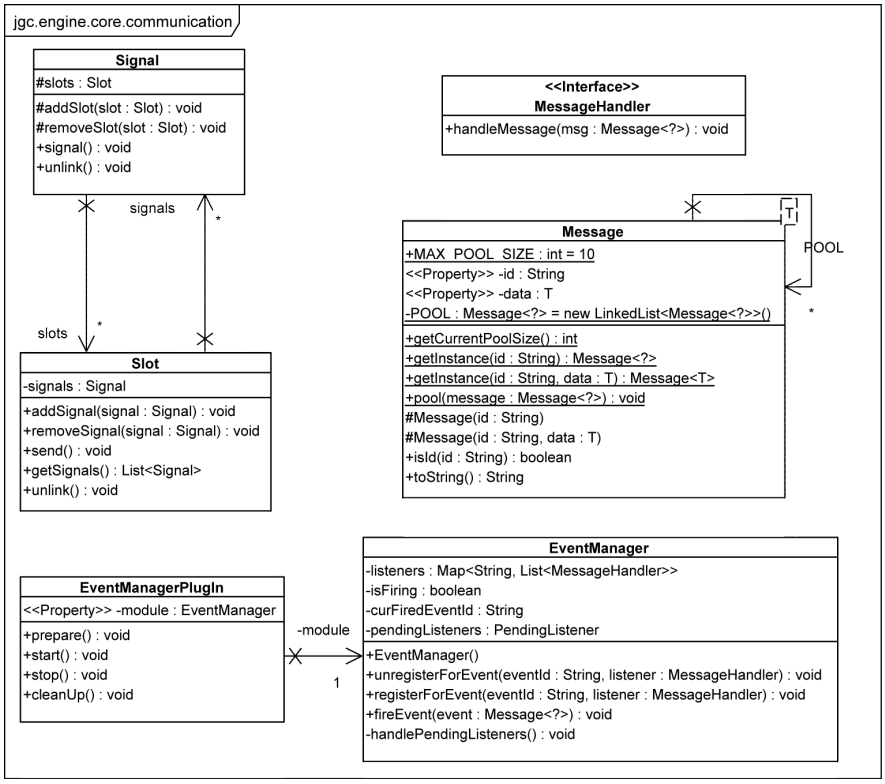


Abbildung A.1: UML-Diagramm der Klassen die innerhalb der Engine für die Kommunikation bereit stehen (siehe Abschnitt 4.4).

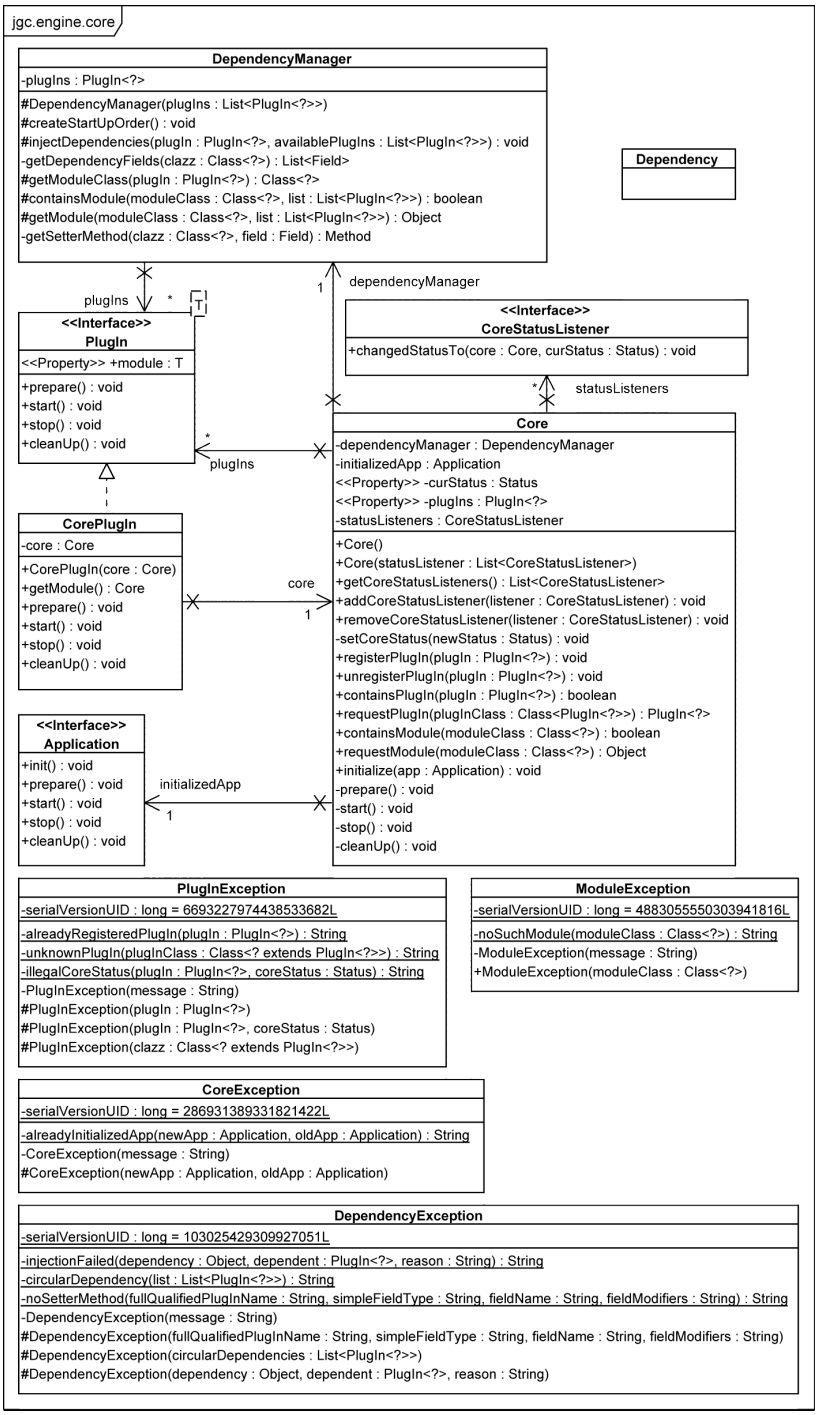


Abbildung A.2: UML-Diagramm der Implementierung des Plug-In Mechanismus aus Abschnitt 4.2

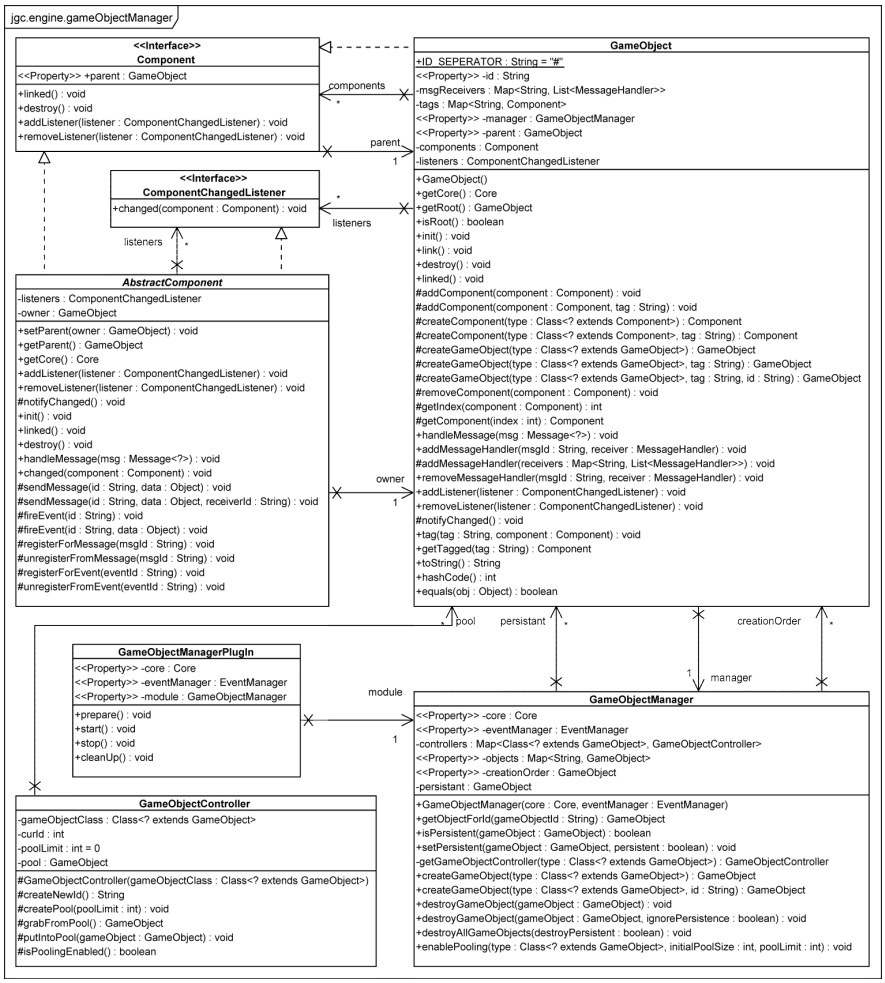


Abbildung A.3: UML-Diagramm der Implementierung der Spielobjekt-Verwaltung aus Abschnitt 3.4

Anhang B

Inhalt der CD-ROM

Format: CD-ROM, Single Layer, ISO9660-Format

B.1 Diplomarbeit

Pfad: /

dm08004_dammerer_alexander_da.pdf Diplomarbeit

B.2 Literatur

Pfad: /

Literatur/ . . . . . Kopien der Internetseiten und technischen Berichte

B.3 Projekt

Pfad: /Projekt/

bin/ . . . . . Jar-Archive der Implementierung
demo/ . . . . . Demospiel
doc/ . . . . . JavaDoc der Implementierung
junit/ . . . . . Testklassen für die Implementierung
src/ . . . . . Quelltext der Implementierung

Literaturverzeichnis

[1] Anderson, E. F., S. Engel, P. Comninos und L. Mcloughlin: *The case for research in game engine architecture.* In: *Future Play '08: Proceedings of the 2008 Conference on Future Play*, S. 228–231, 2008.

[2] Bradbury, S.: *Client-Spiel trifft Browsergame.* Making Games Magazine, 01(1):44–45, Feb. 2010.

[3] Chuang, T. R., Y. S. Kuo und C. M. Wang: *Non-intrusive object introspection in c++: Architecture and application.* In: *In Proc. Int. Conf on Software Engineering*, S. 312–321. Society Press, 1998.

[4] Deloura, M.: *Game engine showdown.* Game Developer Magazine, 16(5):7–12, Mai 2009.

[5] ECMA: *ECMAScript Language Specification, 5th Edition*, Dez. 2009. http://www.ecma-international.org/publications/standards/Ecma-262.htm.

[6] Forman, I. R. und N. Forman: *Java Reflection in Action.* Manning, Mai 2004.

[7] Gieselmann, H.: *Visionen für Millionen – Spieletrends auf der Game Developer Conference 2010.* C't, 8(1):20–24, Mai 2010.

[8] Gregory, J.: *Game Engine Architecture.* Transatlantic Publishers, Apr. 2009.

[9] Haller, M., W. Hartmann und J. Zauner: *A generic framework for game development.* In: *ACM SIGGRAPH and Eurographics Campfire*, 2002.

[10] Harmon, M.: *A system for managing game entities.* In: *Game Programming Gems 4*, S. 69–83. Charles River Media, 2004.

[11] Heineman, G. T. und W. T. Councill: *Component-Based Software Engineering — Putting the Pieces Together.* Addison-Wesley, 2001.

[12] Krüger, G.: *Handbuch der Java-Programmierung.* Addison-Wesley, 2006.

[13] Kuchana, P.: *Software Architecture Design Patterns in Java.* CRC Press, 2004.

[14] Lange, T.: *Das komponentenbasierte GameObject-System.* Making Games Magazine, 03(1):40–43, Mai 2010.

[15] Leach, P., M. Mealling und R. Salz: *A Universally Unique IDentifier (UUID) URN Namespace*, Juli 2005. http://www.ietf.org/rfc/rfc4122.txt, Kopie auf CD (rfc4122.pdf).

[16] Liang, S.: *Java Native Interface.* Addison-Wesley, Mai 1999.

[17] Linklater, M.: *Implementing Dataports*, Nov. 2006. http://www.gamasutra.com/view/feature/1779/implementing_dataports.php, Kopie auf CD (ImplementingDataports.pdf).

[18] Passos, E. B., J. W. S. Sousa, E. W. G. Clua, A. Montenegro und L. Murta: *Smart composition of game objects using dependency injection.* Computers in Entertainment, 7(4):53:1–53:15, 2009.

[19] Plank, M.: *Visuelle komponentenbasierte Spieleentwicklung.* Masterarbeit, Fachhochschule Hagenberg, Digitale Medien, Hagenberg, Austria, Juni 2008.

[20] Plummer, J.: *A flexible and expandable architecture for computer games.* Masterarbeit, Arizona State University, Nov. 2004.

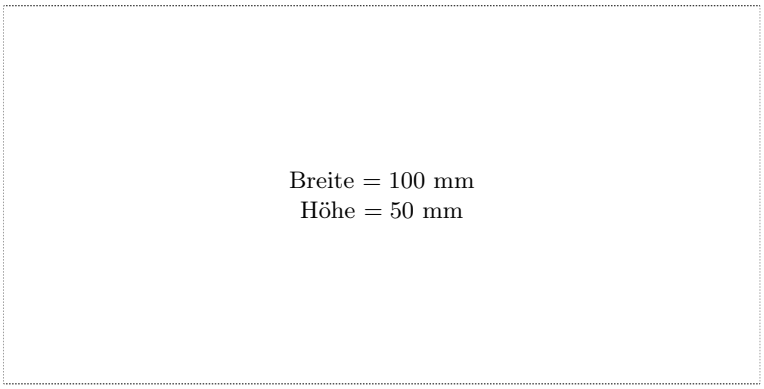
[21] Rollings, A. und D. Morris: *Game Architecture and Design.* Coriolis Group Books, 1999.

[22] Trolltech: *QT Reference Documentation, Signals and Slots*, 2010. http://doc.trolltech.com/4.7/signalsandslots.html, Kopie auf CD (SignalsSlots.pdf).

[23] West, M.: *Evolve Your Hierarchy — Refactoring Game Entities with Components*, 2007. http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/, Kopie auf CD (EvolveYourHierarchy.pdf).

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —