# Middleware zur Unterstützung architekturgetriebener Spielentwicklung

**Optimierung des Entwicklungsaufwandes für browserbasierte MMOGs**

Dissertation
zur Erlangung des akademischen Grades
Dr.-Ing.

vorgelegt dem Rat der Fakultät für Mathematik und Informatik
der Friedrich-Schiller-Universität Jena

von Herr Dipl.-Inf. Sebastian Apel
geboren am 03.10.1984 in Erfurt

Datum der Einreichung: 12.10.2016

**Gutachter**

1. Prof. Dr. Wilhelm R. Rossak, Friedrich-Schiller-Universität Jena
2. Prof. Dr. habil. Jörg Roth, Technische Hochschule Nürnberg
3. Prof. Dr. Christian Erfurth, Ernst-Abbe-Hochschule Jena

**Tag der öffentlichen Verteidigung:** 05.05.2017

# Abstract

Game development can be a massive task and is related to a broad range of disciplines. This provokes the question about redundancies especially within the domain of software development. The following approach takes a closer look onto browser-based massively multiplayer online games and identifies typical architectural characteristics within communication infrastructures. Based on the provided middleware, development can be reduced to conceptual elements without technological references. The middleware derives required information from game concepts, generates the necessary infrastructure and reduces development overheads. By using the COCOMO II cost model, seven different game concepts are analysed. The results of our analysis demonstrate an existing implementation overhead and its growth. Furthermore, this analysis demonstrates how to avoid this overhead by using the implemented middleware.

Softwarearchitecture, Massive Multiplayer Online Game, Game Development, Generative Code, Cost Analyses, Model Driven Development

# Zusammenfassung

Spielentwicklung ist ein umfangreiches Vorhaben und tangiert eine Vielzahl an Disziplinen. Dies provoziert die Frage nach Redundanzen, insbesondere im Bereich der Softwareentwicklung. Mit einem Fokus auf die Realisierung von browserbasierten Massively Multiplayer Online Games, werden im Rahmen dieser Arbeit architekturelle Charakteristiken zur Bereitstellung von typischen Kommunikationsinfrastrukturen identifiziert, welche sich in unterschiedlichen Ansätzen wiederfinden lassen. Mithilfe einer realisierten Middleware, zur Ableitung und generischen Bereitstellung einer Infrastruktur, kann der Aufwand zur Entwicklung einer Spielidee auf konzeptbezogene, technologieunabhängige Elemente reduziert werden. Unter Verwendung des COCOMO II-Kostenmodells werden sieben Spielkonzepte analysiert, Aussagen zur Aufwandsentwicklung dargelegt sowie die Möglichkeit zur Vermeidung des Aufwands in der realisierten Middleware demonstriert.

Softwarearchitektur, Massive Multiplayer Online Game, Spielentwicklung, Codegenerierung, Aufwandsanalyse, Modelgetriebene Entwicklung

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Abkürzungsverzeichnis

**AI** Artificial Intelligence

**AJAX** Asynchronous JavaScript and XML

**API** Application Programming Interface

**BNF** Backus-Naur Form

**BSON** Binary JSON

**CDI** Contexts and Dependency Injection

**DOM** Document Object Model

**EE** Enterprise Edition

**EGM** Enterprise Gaming Metamodel

**EJB** Enterprise Java Bean

**ES5** Electroserver 5

**EUP** Electrotank Universe Platform

**HTML** HyperText Markup Language

**HTTP** Hypertext Transfer Protocol

**IP** Internet Protocol

**JAAS** Java Authentication and Authorization Service

**JACC** Java Authorization Contract for Containers

**JAX-RS** Java API for RESTful Web Services

**JAX-WS** Java API for XML Web Services

**JAXB** Java Architecture for XML Binding

**JAXP** Java API for XML Processing

**JCP** Java Community Process

**JDBC** Java Database Connectivity

**JMX** Java Management Extension

**JNDI** Java Naming and -Directory Interface

**JOS** Java Object Serialization

**JPA** Java Persistence API

**JSF** Java Server Faces

**JSON** JavaScript Object Notation

**JSP** Java Server Pages

**JSR** Java Specification Request

**JTA** Java Transaction API

**LAN** Local Area Network

**LLoC** Logical Lines of Code

**LoC** Lines of Code

**MBean** Managed Bean

**MMOG** Massively Multiplayer Online Game

**MVC** Model-View-Controller

**ORM** Object-Relational Mapping

**OSI** Open System Interconnection

**PHP** PHP: Hypertext Preprocessor

**ProtoBuf** Protocol Buffers

**REST** Representational State Transfer

**RFC** Request for Comments

**RMI** Remote Method Invocation

**RPG** Role Playing Game

**SE** Standard Edition

**SmartFoxServer** SmartFoxServer 2X

**SOAP** Simple Object Access Protocol

**SQL** Structured Query Language

**TCP** Transmisson Control Protocol

**UCC** Unified Code Counter

**UDP** User Datagram Protocol

**URI** Uniform Resource Identifier

**URL** Uniform Resource Locator

**VM** Virtual Machine

**W3C** World Wide Web Consortium

**WWW** World Wide Web

**XML** Extensible Markup Language

**XPath** XML Path Language

# 1. Einleitung

Eine unberührte endlose Natur mit Wüsten, Wäldern, Seen, Taiga, Tundras und Meeren, welche frei von Humanoiden ist, wird jäh gestört durch einen lauten Knall und spontan vom Himmel herabstürzenden Würmern. Den Aufschlagkrater verlassend, beginnen die Würmer zu fressen, Hölzer zu sammeln und sich zu reproduzieren. Erste Individuen entwickeln Beine, um an höher gelegene Nahrung zu gelangen. Andere Individuen entdecken die Vorzüge eines gemütlichen und feuchten Lochs unter der Erde und wieder andere Viecher gewinnen Geschmack an dem Verzehr ihrer Kontrahenten. Mit zunehmend verstrichener Zeit, gestaltet sich die kontinuierliche Fortpflanzung der entstehenden Fauna und der damit verbundenen Spezialisierung von Körper und Geist zu einem Katz und Mausspiel zwischen gefressen und gefressen werden. Einem sich entwickelnden Wechselspiel, bei dem jedes Mittel zur Erhaltung der eigenen Herde und der Expansion in Pangäa gerechtfertigt scheint.

Die Idee zu Viechers [67] beschreibt ein Spielkonzept, bei dem in einem chaotischen Umfeld ein durch den Spieler gesteuerter Evolutionsprozess dazu genutzt wird, um eine dominante Spezies auf dem Urkontinent Pangäa durch deren Weiterentwicklung zu erschaffen. Konzeptioniert als browserbasiertes Massively Multiplayer Online Game (MMOG), welches Spiele mit mehreren tausend Spielern in einer persistenten Umgebung beschreibt [46, S. 23], soll auf einer gemeinsam genutzten Landmasse um begrenzte Ressourcen, Fläche und das Recht des Stärkeren konkurriert werden, um alleine oder als Team das Ziel des Überlebens zu realisieren.

Ein Spiel, bzw. ein MMOG, zu entwickeln ist nach Blow [14, S. 35] „Harder than you think“ und umfasst nach ihm für eine 3D MMOG-Entwicklung z. B. Bereiche wie Grafiken, Audio und Musik, 3D Animationen und Modelle, Rendering, Physik und Kollisionen, Persistenz, Authentifizierung, Updateprozesse, Werkzeuge, Datenübertragung, Datenformate, Simulation des Modells, Analy-

sen, Artificial Intelligence (AI), Server Gameplay Code und statische Inhalte. Diese Elemente zur Entwicklung finden sich in der Aufführung von Disziplinen in der Spielentwicklung von Bethke [11, S. 63] und Gregory [46, S. 5], welche z. B. Audio Designer, Grafiker, Animatoren, 3D Modellierer, sowie Software-, AI-, Netzwerk-, Spiel-Mechanik- und Script-Programmierer umfasst. Die Realisierung eines Spiels ist, neben den kreativen Elementen zur Konzeptionierung, sowie auditiven und visuellen Ausgestaltung, eine Softwareentwicklung [11, S. 4]. Für die Entwicklung eines Spiels im Sinne einer Softwareentwicklung sind damit Anforderungs- und Anwendungsfallanalysen [11, S. 136-144], Modellierung der Entitätstypen [11, S. 145-147], sowie der Architekturerstellung [11, S. 149] ebenso notwendig.

Viechers ist als browserbasiertes MMOGs zu realisieren. Browserbasierte MMOGs werden vom Spieler direkt im Browser ausgeführt und genutzt [7, 77]. Mit dieser Zielstellung vereinfacht Viechers die Wahl der Werkzeuge zur Darstellung auf jene im Browser ausführbaren. Darüber hinaus wirft das Konzept jedoch die Frage nach der technischen Realisierung auf. Dies betrifft insbesondere den Austausch von Informationen zwischen Visualisierung im Browser und einer zentralen Instanz, die das Spielkonzept persistent bereitstellt. Eine Vielfalt an Spiel-bezogenen Architekturen [6, 12, 15, 37, 56, 68] und Erfahrungsberichten [14, 79] mit dem Fokus auf Spiele, Mehrspieler-Spiele und MMOG, sowie die Referenzarchitektur in [21], skizzieren einen klassischen Aufbau, der sich in der Regel auf eine Client-Server-Architektur reduzieren lässt. Insbesondere greifen diese Spiel-bezogenen Vorlagen die Notwendigkeit von Nachrichtenverarbeitung, Netzwerkschichten, Abbildungsformaten und Persistenz, sowie Simulation des Spiels ohne direkte Nutzerinteraktion auf. Bei der Realisierung jener Elemente, bzw. die Etablierung einer Architektur, welche in der Lage ist, Informationen zwischen den verschiedenen Spielern über eine zentrale Instanz zu verteilen, muss in jeder Spielentwicklung eine Entscheidung nach den zu verwendenen Protokollen und Werkzeugen getroffen werden. Ein Ad-Hoc Ansatz wäre die eigenständige Konstruktion eines Transmisson Control Protocol (TCP)-Servers und -Clients wie z. B. von Mulholland und Hakala [56, S. 218] beschrieben. Alternativ sind Ansätze möglich: die Realisierung mittels Hypertext Transfer Protocol (HTTP) – z. B. über Werkzeuge wie Express.js [70] für Node.js [57] – oder mittels spezialisierter Lösungen, die für

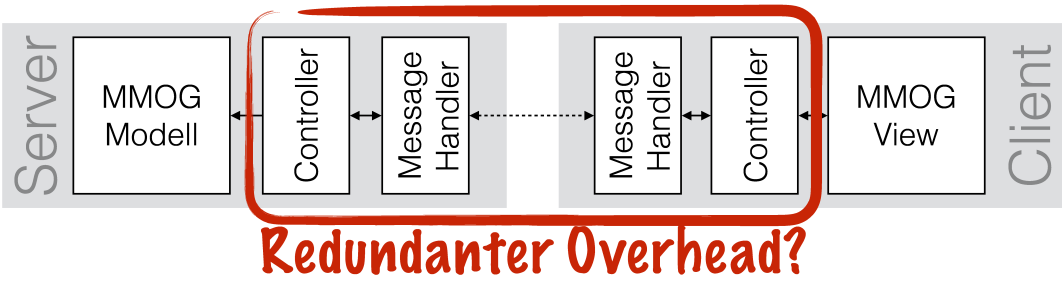



**Abbildung 1.1: Zielstellung für einen alternativen Ansatz.**

die Realisierung von browserbasierten MMOGs zur Verfügung stehen, z. B. der SmartFoxServer [42] und der Photon Server [34]. Jeder Ansatz lässt einen Overhead vermuten, dargestellt in Abbildung 1.1, der zur Realisierung der eigenen Spielidee in der jeweiligen Umgebung entsteht. Dies wirft die Frage auf, welche Elemente dieser Architektur wiederkehrend, bzw. redundant und somit vermeidbar sind.

Im Rahmen dieser Arbeit ist der Overhead, welcher bei der Anwendung der verschiedenen Architekturansätze für ein browserbasiertes MMOG entsteht, zu identifizieren und charakterisieren. Zusätzlich ist der Aufwand der Realisierung bezüglich den Charakteristiken zu quantifizieren, um einen Eindruck über die Verteilung zwischen den verschiedenen Merkmalen zu erhalten. Mithilfe der Herleitung eines minimalen Spielkonzepts als „Beispiel“ werden im Folgenden unterschiedliche Implementierungen von Architekturen für die Betrachtung herangezogen. In Ergänzung zur Betrachtung des Overheads werden Anforderungsanalysen und der Stand der Technik dazu genutzt um eine Architektur und deren zu verwendenden Protokolle und Abbildungsformate als alternativen Ansatz abzuleiten. Der neue, alternative Ansatz adressiert den identifizierten Overhead und berücksichtigt die analysierten Anforderungen an eine Ausführungsumgebung für browserbasierte MMOGs.

Die in Java entwickelte Ausführungsumgebung, welche im Rahmen des EXIST Programms des Bundesministeriums für Wirtschaft und Energie [17] begonnen und anschließend mithilfe der Thüringer Gründerfonds weiter finanziert wurde, dient zur Minimierung des Overheads bei der Realisierung eines MMOG-Konzeptes. Im Rahmen dieser Arbeit wurde die Ausführungsumgebung weiter spezialisiert, um die Entwicklung auf die Realisierung der Mechanismen einer Spielidee zu reduzieren, den Kontakt mit technologischen

Randbedingungen zu vermeiden und das Werkzeug in Entwicklungsprozesse von Spielprojekten einzubetten und zu erproben. Die Bewertung der erreichten Ziele wird mittels Daten- und Modellanalysen durchgeführt. Hierfür wird die Ausführungsumgebung zur Bewertung in unterschiedlichen Spielprojekten angewendet und basierend auf der initialen Betrachtung des Overheads verglichen. Im Ergebnis kann eine Aussage über die erreichte Minimierung formuliert werden. Zusätzlich werden Randbetrachtungen der entwickelten Lösungen durchgeführt, um zum Beispiel die Eigenschaften des verwendeten Abbildungsformates zur Übertragung von Informationen zwischen Spieler und zentraler Instanz im Vergleich zu alternativen Ansätzen zu betrachten.

Für die Erprobung und Anwendung der Ausführungsumgebung werden neben eigenen Implementierungen, wie Viechers und einem Demonstrator Stickman, die Projekte Kokosnussbikini, Terra-Life-Evolution, Hack'N'Slay und MMO-Meischta, welche in Zusammenarbeit mit Studenten entstanden sind, betrachtet. Der Versuchsaufbau bei der Durchführung der Studentenprojekte adaptiert im Rahmen von Lehrveranstaltungen den Entwicklungsprozess für Spiele [23][11, S. 39]. Dieser Prozess umfasst Methodiken zur Skizzierung einer Spielidee, Anwendungsfall- und Anforderungsanalysen, sowie Designdokumente zur Beschreibung von Entitätstypen, Generalisierungen und Assoziationen der intendierten Spielideen zum Praktizieren der klassischen Softwareentwicklung. Abgeschlossen werden die Projekte mit der Realisierung der Idee unter Verwendung der Ausführungsumgebung in einen ausführbaren, vertikalen Prototypen.

Für die Analyse der realisierten Spielprojekte wird mittels „Reverse Engineering“ das verfügbare Spielmodell dokumentiert, mit dem zuvor durch das Team dokumentierten Spielmodell verglichen und hinsichtlich erreichter Qualitäten untersucht. Darüber hinaus findet die Bewertung des Aufwands mittels des COCOMO II Kostenmodells [76] und der darin enthaltenen Bewertung des Aufwands der Implementierung auf Basis von sogenannten Logical Lines of Code (LLoC) statt. Mithilfe eines im Rahmen der Arbeit entwickelten Messwerkzeugs wird jede Codezeile hinsichtlich den herausgearbeiteten Charakteristik einer MMOG-Architektur bewertet, um die gewünschte Aussage über die Aufwandsentwicklung und den Overhead bezüglich dem Spielmodell zu erhalten.

Die final resultierende Ausführungsumgebung kann als „Network Engine“ zur Entwicklung von Spielen klassifiziert werden und grenzt sich damit zu prominenteren Entwicklungsumgebungen wie der Unity- [74] und Unreal Engine [33] ab. Werkzeuge dieser Art unterstützen die Entwicklung von Mehrspieler-Spielen, fokussieren jedoch auf die effiziente Realisierung des Clients und auf die Erstellung von 3D Landschaften [46, S. 26].

Die folgende Arbeit beginnt in Kapitel 2 (S. 7) mit der Differenzierung der Begriffe „Spiel“, „Videospiel“ und „Online Spiel“, um diese zum Begriff browserbasiertes MMOG zu erweitert und folgend Anforderungen abzuleiten und ein verallgemeinertes Spielmodell zu schaffen, welches als Basis eines minimalen Beispiels genutzt werden kann. Im Kapitel 3 (S. 31) werden verschiedene Technologien betrachtet, die im Rahmen eines browserbasierten MMOGs zum Einsatz kommen können. Die betrachteten Technologien werden in Implementierungen des minimalen Beispiels aufgegriffen und zur Identifikation der Charakteristiken betrachtet und zur Bewertung des Aufwands analysiert. Mithilfe der daraus abgeleiteten Thesen in Kapitel 4 (S. 89) entsteht anschließend in Kapitel 5 (S. 93) die Architektur als Grundlage für die in Kapitel 6 (S. 123) beschriebene Implementierung der Ausführungsumgebung. Abschließend betrachtet und bewertet Kapitel 7 (S. 179) die resultierende Ausführungsumgebung in unterschiedlichen Anwendungsbeispielen. Kapitel 8 (S. 221) greift die Ergebnisse auf und zieht Vergleiche zu den initialen Betrachtungen, um diese abschließend zu bewerten.

# 2. Massively Multiplayer Online Games

Spielentwicklung, insbesondere der Prozess zur Entwicklung eines MMOGs, erscheint nach der Auflistung von Komponenten und Disziplinen in [14], [11, S. 86] sowie [46, S. 5] aufwendig und umfangreich. Im ersten Schritt wird erarbeitet, was genau Spiele sind, was Online Spiele sind, was browserbasierten MMOGs sind und wie Spiele im Allgemeinen entwickelt werden. Als Ergebnis entsteht ein abstraktes Modell zur Beschreibung eines Spieles, welches für die Entwicklung verwendet werden kann. Darauf aufbauend werden Anforderungen an die Ausführung von browserbasierten MMOGs formuliert, sowie ein einfaches Beispiel abgeleitet, welches die Grundelemente dieses Modells verwendet. Die Ergebnisse werden im folgenden Kapitel 3 (S. 31) verwendet, um den Aufwand, sowie Techniken und Werkzeuge zur Realisierung eines browserbasierten MMOG zu skizzieren.

## 2.1 Spiele

„Spiel ist eine freiwillige Handlung oder Beschäftigung, die innerhalb gewisser festgesetzter Grenzen von Zeit und Raum nach freiwillig angenommenen, aber unbedingt bindenden Regeln verrichtet wird, ihr Ziel in sich selber hat und begleitet wird von einem Gefühl der Spannung und Freude und einem Bewusstsein des ‚Andersseins' als das ‚gewöhnliche Leben'." [48, S. 37] Spiele entstehen durch das menschliche Begehren zu spielen und dessen Möglichkeit zum Vorstellen. Spielen kann als unwesentlich und übliche Freizeitaktivität angesehen werden, oder wie Mark Twain „Arbeiten" bezüglich „Spielen" vergleicht: „Work consists of whatever a body is OBLIGED to do, and that Play consists of whatever a body is not obliged to do." [73, S. 32] Spielen bringt somit die Freiheit zu agieren sowie die Freiheit zu wählen, wie agiert wird. Spielen ermöglicht ebenso sich etwas vorzustellen. Das Vorstellen (vgl. engl.

pretend) innerhalb eines Spiels ist die Fähigkeit eine fiktive Realität zu schaffen. Diese unterscheidet sich für den Vorstellenden bewusst von der realen Welt und kann von ihm erzeugt, beendet oder geändert werden – der sogenannten „pretended reality“ [3, S. 2].

Zur Definition des Handlungsspielraums innerhalb der „pretended reality“ haben Spiele Regeln. Diese existieren, selbst wenn sie nicht niedergeschrieben oder angewendet werden. Regeln sind Anweisungen, wie das Spiel zu spielen ist und werden vom Spieler für die Dauer des Spielens akzeptiert. Zur Differenzierung können diese, nach Adams in [3, S. 8], in sechs Gruppen von Regeln unterschieden werden:

- **Semiotics of the game**
  Diese bezeichnen die Bedeutung und die Beziehungen der verschiedenen Symbolik des Spiels. Diese können abstrakt sein oder Parallelen zur realen Welt haben. Sie enthalten Objekte eines Spiels, mit denen ein Spieler interagieren kann.

- **Gameplay**
  Beschreibt die Herausforderungen (challenges) und Aktionen (actions), die ein Spiel dem Spieler zur Verfügung stellt.

- **Sequence of play**
  Bezeichnet Weiterentwicklung von Aktivitäten, die das Spiel bildet.

- **Goal of the game**
  Definiert den Zweck des Spiels, welchen es gilt zu erreichen. Jedes Spiel muss ein Ziel haben, wobei dieses beliebig sein kann, da es durch den Spieldesigner erdacht wurde.

- **Termination condition**
  Dies sind Regeln, die ein Spiel beenden.

- **Metarules**
  Bezeichnet Regeln, die beschreiben, unter welche Bedingungen sich andere Regeln ändern können oder wann Ausnahmen erlaubt sind.

Aus diesen drei Regeln entscheiden „Semiotics of the game“ über die im Spiel definierten Symboliken sowie deren Beziehungen und „Gameplays“ über Herausforderungen und Aktionen. Während Herausforderungen dem Spieler gestellt werden, definieren Aktionen, wie ein Spieler sich dieser Herausforderung entgegenstellen kann. Sie bestimmen seine Handlungsmöglichkeiten, damit der Spieler die ihm gestellten Herausforderungen bewältigen und das Ziel des Spiels erreichen kann. Sid Meier sagte „A game is a series of interesting choices.“ [3, S. 9], und meint damit, dass ein Spiel lohnenswert ist, wenn Entscheidungen in einem Spiel nicht trivialer Natur sind. Die möglichen Aktionen zur Bewältigung von Herausforderungen stellen damit einen wichtigen Bestandteil in Spielen dar. Gutes Gameplay erfordert zwei weitere Faktoren: Risiko und Belohnung. Wieso sollte sich ein Spieler einer Herausforderung stellen, wenn er dadurch keine Belohnung erhält? Je höher das Risiko ist, desto umfangreicher muss die Belohnung ausfallen. [3, S. 11ff]

Die Regeln, das Spielen an sich und die vorgestellte Realität ergeben die Grundelemente eines Spiels und lassen sich in der folgenden Definition für ein Spiel wiederfinden:

**Definition 1** *A game is a type of play activity, conducted in the context of a pretended reality, in which the participant(s) try to achieve at least one arbitrary, nontrivial goal by acting in accordance with rules. [3, S. 3]*

## 2.2 Videospiele

Wie unterscheiden sich Videospiele von klassischen Spielen? Ein Videospiel greift dieselben Grundelemente eines konventionellen Spieles auf. Dieses unterscheidet sich jedoch bei deren Zugang zu Regeln und Einfluss auf das Tempo. Videospiele benötigen keine niedergeschriebenen Regeln. Regeln werden durch den Entwickler implementiert und der Spieler wird zur Einhaltung bei der Interaktion mit dem Spiel gezwungen. Das führt letztlich dazu, dass Spieler zum Spielen die Regeln nicht kennen müssen – sie können diese ausprobieren. Dies birgt den Nachteil, dass es häufig schwierig ist, das Spielergebnis zu optimieren, wenn die Regeln nicht bekannt sind, die zu einem bestimmten Ergebnis führten. Ähnlich verhält es sich mit der Zeit. Konventionelle Spiele erhalten ihr

Tempo durch den Spieler oder einem unabhängigen Gutachter. In Videospielen definiert der Computer das Tempo. [3, S. 15]

Durch versteckte Regeln und vorgegebenem Tempo wird Gameplay zur entscheidenden Quelle des Unterhaltungswertes bei Videospielen und damit erneut entscheidend für den Erfolg eines Spieles. Es ist wichtig, welche Herausforderungen dem Spieler gestellt werden und mit welchen Aktionen er diesen Herausforderungen entgegen treten kann. [3, S. 19]

Die von einem Spieldesigner definierten Regeln sind im Fall eines Videospiels in eine passende Symbolik und dessen mathematisches Model zu überführen, welches folgend implementiert werden kann. Dieses mathematische Regelmodell – im Folgenden „Spielmodell“ genannt – ist die Kernmechanik eines Videospiels. Es beschreibt die Symbolik, Relationen, sowie die möglichen Aktionen und bildet Herausforderungen ab, die einem Spieler ein interessantes Gameplay bieten.

## 2.3 Online Spiele

Bisher betrachtet wurden klassische Spiele, die von einem oder mehreren Spielern gespielt werden können, sowie Videospiele, die sich in erster Linie einem Spieler widmen. Im Folgenden wird herausgearbeitet, wie sich dieses Videospiel zu einem Online Spiel spezialisieren lässt, bzw. was diese von einem Videospiel unterscheidet.

Die ersten Echtzeit-Multiplayer-Spiele boten für zwei Spieler ein miteinander verbundenes Spiel über ein Modem an, z. B. Falcon A.T., und wurden mit der Verfügbarkeit von Local Area Network (LAN) und Internet, z. B. mit Quake und Bartle, für mehr Spieler zugänglich. [68, S. 7]

Ein Spiel wird als Online bezeichnet, wenn deren Spieler und die ausführenden physischen Maschinen über ein Netzwerk miteinander verbunden sind. Die Verbindungsart setzt nicht zwingend das Internet (Wide-Area-Network) voraus. Ein in einem LAN agierendes Spiel ist ebenso als Online Spiel zu verstehen. [3, S. 591]

Online Spiele sind durch eine Interaktion zwischen den verschiedenen Spieler charakterisiert. Spieler spielen nicht ausschließlich mit bzw. gegen künstliche Intelligenzen. Häufig agieren Spieler gemeinsam oder agieren in Konkurrenz

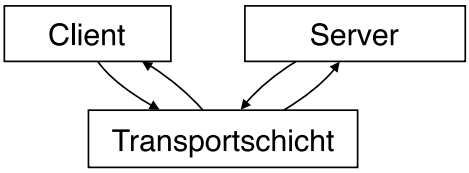



**Abbildung 2.1: Darstellung einer vereinfachten Sicht auf ein MMOG.**

um begrenzte Ressourcen. [68, S. 7]

Differenziert man Online Spiele weiter, so lassen sich zwei Variationen finden. Die erste bezeichnet Spiele, die in einer kleinen weitestgehend festen Gruppierung gespielt werden, z.B. lokale LAN Mehrspielermodi. Die zweite bezeichnet Spiele, die in einer sich häufig verändernden Spielerschaft mit ggf. vielen tausend Spielern auf einer gemeinschaftlichen virtuellen Spielwelt stattfinden, sogenannte MMOGs.

## 2.4 Massively Multiplayer Online Games

Während lokale LAN Mehrspielermodi hier nicht von vertiefenden Interesse sind, liegen die MMOGs im Fokus der Betrachtung. Spiele dieser Art verbinden über das Internet tausende Spieler zu einer virtuellen persistenten Gemeinschaft. Eine zentrale Instanz dient zur ständigen Synchronisation und Persistenz der Spielerdaten und muss Änderungen an potentielle Interessenten verteilen [46, S. 23]. Ebenso muss diese zentrale Instanz die Integrität der Spieldaten garantieren und beugt durch zumeist zentralisierte Verwaltung der Bevorzugung einzelner Spielindividuen vor („cheating“) [68, S. 214]. Die zentrale Instanz organisiert die Weiterentwicklung abhängig von Einflussfaktoren, steuert die Zugriffsrechte auf die jeweiligen Symboliken im Spiel die innerhalb der vorgestellten Realität existieren und verwaltet die sich ständig ändernde Menge an aktiven Spielern.

Als Gegenstück zur zentralen Instanz, dem sogenannten Server, dient in einer typischen MMOG-Architektur wie in [9, S. 16] der Client des Spielers. Er kann als visuelle und auditive Präsentationsfläche verwendet werden. Der Client bereitet die Spieldaten auf und präsentiert sie dem Individuum, welches das Spiel auf einer für das Spiel geeigneten Systemplattform benutzt. Server, Client und der Austausch von Informationen über die sogenannte Transport-

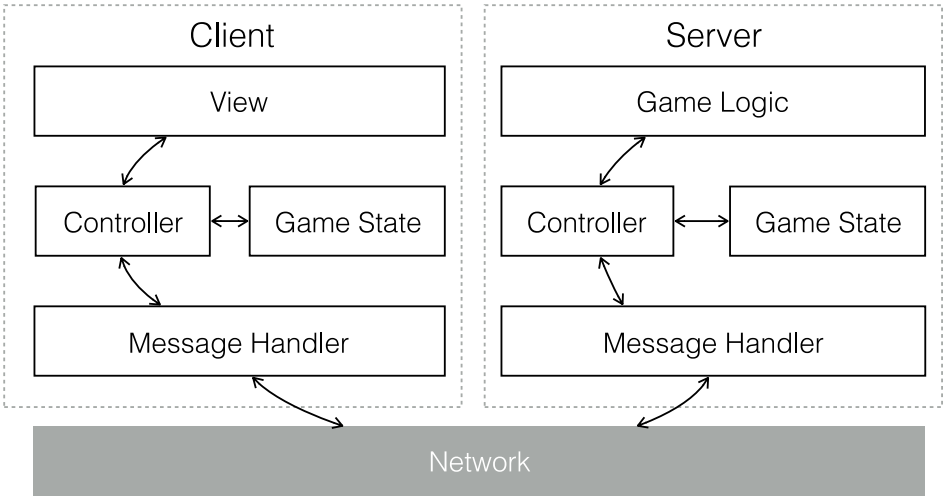



**Abbildung 2.2: Darstellung einer Architektur für MMOGs. Der Entwurf basiert auf einer Referenzarchitektur für MMOGs ignoriet an dieser Stelle jedoch AI und Grafik, Sound und Hardware. Pfeile stellen im Diagramm den Datenfluss dar [21].**

schicht ergeben die in Abbildung 2.1 gezeigte vereinfachte Architektur.

Basierend auf einem Entwurf für eine MMOG-Referenzarchitektur in [21] können Client, Server und Transportschicht aus Abbildung 2.1 um typische Komponenten erweitert werden, die für die Umsetzung des MMOGs notwendig sind. Abbildung 2.2 zeigt die erweiterte Architektur grafisch. Client und Server werden wie in Abbildung 2.1 über eine *Transportschicht* verbunden. Darauf aufbauend existiert auf der Seite des Servers und Clients ein *Message Handler*, der eingehende und ausgehende Nachrichten verarbeitet, interpretiert, auf lokale Kommunikationsobjekte abbildet. Während auf Clientseite die Interpretation auf einen Nutzer bezogen ist, muss die Serverseite die Interpretation mehrerer parallel agierender Nutzer verarbeiten. Die dem *Message Handler* nachgelagerte Komponente *Controller* erfasst die durch den *Message Handler* produzierten Ereignisse in Form von Kommunikationsobjekten und wendet sie auf die „Game Logic“ an und persistiert ggf. entstandene Änderungen am Zustand des Modells. Im Client wird, im Gegensatz zum Server, ein Teilabbild des Spielzustandes gepflegt. Dieses ist notwendig, um die aktuelle Spielumgebung zu visualisieren. [21]

Für die Realisierung eines Spiels müssen die verschiedenen Komponenten dieser Architektur implementiert werden. Zur Vereinfachung dieser Entwick-

lung ist es möglich *View*, *Message Handler* und Peristenz mit Hilfe von Frameworks, bzw. Middleware zu realisieren. Eine Anwendung dieser wird in Kapitel 3 (S. 31) dargestellt.

## 2.5 Browserbasierte Massively Multiplayer Online Games

Abschließend die Betrachtung der browserbasierten MMOGs. Browserbasierte MMOGs erweitern das Konzept des MMOGs um die Zugänglichkeit des Spiels bzw. die Ausführung des Clients über einen herkömmlichen Browser. Ein Spieler kann die Anwendung ohne zusätzliche Hilfsmittel oder Installationen direkt in diesem Nutzen. Wird die Anwendung über das Internet bereitgestellt, kann ein browserbasiertes MMOG unabhängig von physischen Positionen des Spielers genutzt werden und jeder internetfähige Computer als Ausführungsumgebung dienen. [7][77]

Populäre Vertreter dieser Spiele sind zum Beispiel Travian [72] und DarkOrbit [13]. Travian ist ein Aufbaustrategiespiel, welches vom Spieler die Verwaltung eine Menge an Dörfern fordert. Der Spieler soll seine Dörfer Ausbauen, Handeln, Allianzen schließen und vor Konkurrenten verteidigen. DarkOrbit ist ein Action-Spiel, bei dem der Spieler ein Raumschiff durch eine virtuelle Raumkarte steuert und gegen außerirdische Lebensformen und andere Spieler kämpft.

Browserbasierte MMOGs sind zumeist kostenfrei zugänglich und verfolgen alternative Modelle zur Monetarisierung. Dies kann zum Beispiel ein Abonnement basiertes Modell mit regelmäßigen Zahlungen oder der Verkauf von In-Game Währung sein. Durch den kostenfreien Zugang sprechen Spiele dieser Art große Nutzergruppen an. Nach eignen Aussagen hat DarkOrbit mehr als 90.000.000 Nutzer. [77]

Eine spezielle Untergruppe dieser Spiele fordert vom Spieler den Browser um ein oder mehrere Plug-ins zu erweitern. Diese Plug-in-basierten Browserspiele werden häufig Plug-in-Spiele genannt, können aber im Folgenden der Kategorie der Browserspiele zugeordnet werden. Eine Differenzierung ist hier nicht notwendig. Der Flash Player [5] und der Unity Web Player [74] sind Beispiele für Plug-ins, welche im Browser installiert werden können, um diesen als

Ausführungsumgebung zu erweitern.

## 2.6 Abstraktes Spielmodell

Mit dem Wissen über die Arten und Bestandteile eines Spiels ist der Versuch möglich ein Model zu beschreiben, welches Spiele abstrahiert und verallgemeinert, sowie als Basis für Realisierungen von Spielen genutzt werden kann. Für die Herleitung dieses abstrakten Models dienen die bereits genannte Definition und enthaltenen Elemente. Spiele finden in einer „pretended reality“ statt und sie enthalten Regeln (siehe Kapitel 2.1, S. 7). Für ein Modell beschreibbar sollen die Teile des Regelwerkes sein, welche sich mit den „Semiotics“ befassen. Dies umfasst Symboliken wie ein Avatar des Spielers, Symboliken von Szenarien, Gegenstände und Non-Personal-Characters sowie deren Beziehungen zueinander. Ebenso beschreibbar in diesem Modell sollen die Aktionen des Gameplays sein, als Möglichkeit zur Interaktion mit dem Spiel. Beschreibbar sollen weiterhin Ereignisse sein, die im Fall von Zustandsänderungen am Modell verarbeitet werden. Herausforderungen im Spiel können durch Ereignissen, die im Fall ihres Eintretens einen bestimmten Prozess anstoßen, beschrieben werden. Das Erreichen eines Zieles im Spiel kann zumeist aus dem Spielzustand ermittelt werden und ist damit, ebenso wie eine Herausforderung, als Ereignisverarbeitung realisierbar. Daraus definiert sich ein Spielmodell als:

**Definition 2** *Ein Spielmodell ist die Menge aus Symboliken, Ereignissen und Aktionen innerhalb einer abgegrenzten Realität.*

Für ein Spielmodell werden damit Symboliken, Ereignisse und Aktionen als gemeinsame Basis verwendet. Diese existieren in einem abgeschlossenen Definitionsraum, der Spielwelt. Bildet man diese Faktoren in einem abstrakten Spielmodell ab ergeben sich folgende Elemente:

- **Spielwelt**
  Definiert eine feste Menge an möglichen Spielobjekten, eine variable Menge an Instanzen von Spielobjekten und eine feste Menge an Spielaktionen. Die Spielwelt gibt vor, wann ein Objekt existiert, wie es entsteht und wie es verschwindet (Lifecycle Management).

- **Spielobjekt**
  Jede Symbolik einer Spielwelt ist ein Spielobjekt. Dies können Lebewesen, Pflanzen, leblose Objekte oder der Spieler, auch in dritter Person, sein.

- **Spielereignisse**
  Definiert ein Ereignis, was zu einem bestimmten Zustand des Spielmodells ausgelöst wird.

- **Spielereignisempfänger**
  Definieren, was passiert, wenn ein bestimmter Zustand des Spieles erreicht wird.

- **Spielaktion**
  Eine Spielaktion wird von einem Objekt auf sich selbst, die Welt oder andere Objekte ausgeführt. Sie können eine Veränderung des Spielzustandes bewirken - müssen aber nicht.

Diese Elemente bilden die Schnittmenge, mit der Spiele unabhängig von ihrem Genre beschrieben werden könnten. Mithilfe dieser Elemente kann die Definition des Spielmodells wie folgt erweitert werden:

**Definition 3** *Ein Spielmodell ist die Menge aus Symboliken, Ereignissen und Aktionen innerhalb einer abgegrenzten Realität. Es ist beschrieben durch eine Entität Spielwelt sowie mehreren Entitäten zu verfügbaren Spielobjekten, -ereignissen und -aktionen. Änderungen am Zustand können über Spielereignisempfänger verarbeitet werden.*

Folgend zwei Beispiele, die zeigen, wie die genannten Elemente des Spielmodells auf ein konventionelles Spiel und ein Videospiel abgebildet werden könnten.

**Beispiel 1** *Das Kartenspiel Doppelkopf ist ein Spiel bestehend aus den Spielobjekten „Karte“ und „Spieler“ und der Spielaktion „Karte legen“, welche von Spielern auf eine „Karte“ ausgeführt werden kann, die er selbst besitzt. Das Tempo des Spiels wird durch die Spieler bestimmt. Durch die Spielaktion „Karte legen“, die durch Spieler ausgeführt werden, kann die Spielwelt ihren Zustand*

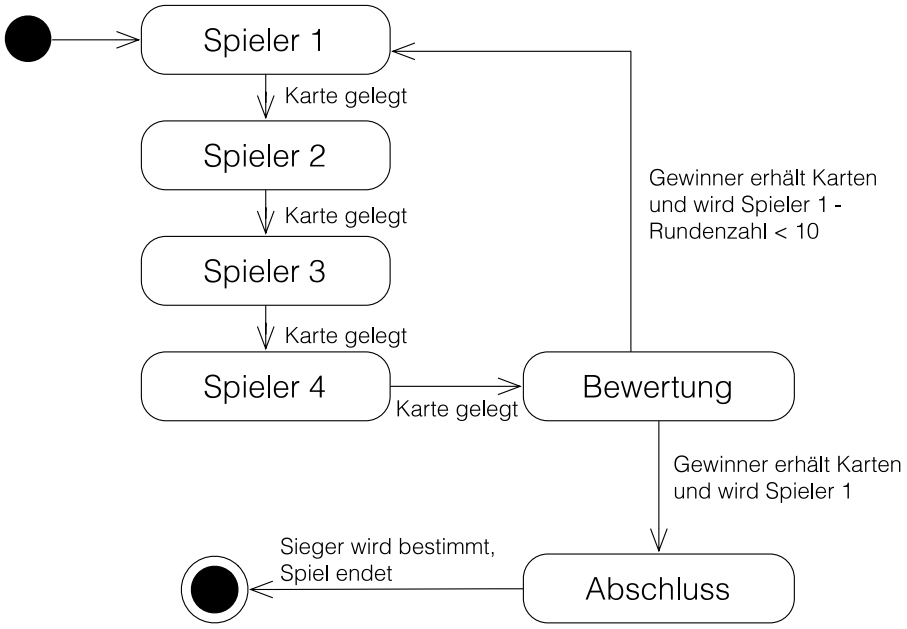



**Abbildung 2.3: Ein einfaches Zustandsdiagramm eines Doppelkopfspiels.**

*ändern: Spieler 1 legt Karte, dann Spieler 2, dann Spieler 3 und dann Spieler 4. Sind 4 Karten gelegt wird ein Ereignis ausgelöst. Der Empfänger bewertet anhand der ausgelegten Karten, wer diese als Punkte (Stich) erhält. Der Gewinner der Runde beginnt in der folgenden Runde als Spieler 1. Dieser Vorgang wird 10-mal wiederholt. Abschließend erfolgt ein Ereignis über das Ende der Runden und der Empfänger des Ereignisses bewertet die erhaltenen Punkte und bestimmt den Gesamtsieger. Abbildung 2.3 zeigt diesen Verlauf in einem Aktivitätsdiagramm grafisch.*

**Beispiel 2** *Das Computerspiel „Age of Empire II“ [52] enthält verschiedene Spielobjekte wie Häuser, Bäume, Tiere Soldaten und Arbeiter, die auf in einem kartesischen Koordinatensystem angeordnet werden. Jedes Objekt erhält eine Teilmenge an Spielaktionen und interagiert mit seiner Umwelt. Einige Aktionen kann der Spieler starten (z. B. Angreifen), andere Aktionen werden als Reaktion ausgeführt (z. B. Verteidigen). Das Spiel entwickelt sich in einem zeitlichen Kontext konstant fort. Der Spieler kann die Fortentwicklung optimieren. Veränderungen am Modell (z. B. das Hinzukommen von Rohstoffen) kann als Ereignis abgebildet werden. Die Möglichkeit zum Bauen neuer Häuser*

*besteht, wenn ausreichend Rohstoffe verfügbar sind (Ereignisempfänger). Ziel des Spiels ist es eine stabilen Zivilisationszustand zu erreichen (wirtschaftlich und militärisch) und die künstlichen oder realen Gegenspieler zu besiegen.*

Wie bereits in Kapitel 2.1 (S. 7) gezeigt, haben Spiele einen zeitlichen Kontext, der im Falle von Videospielen durch den Computer, bzw. die Anwendung vorgegeben wird. Es wäre möglich eine zeitbasierte Modellmodifikation auf eine ereignisbasierte Modifikation zurückzuführen – da der Zeitfaktor als Bedingung für ein Ereignis aufgefasst werden kann. Technisch setzt die Bearbeitung von Zustandsveränderungen auf Basis von Zeitfaktoren eine spielerlosgelöste Aktionsquelle voraus, einen unabhängig agierenden Dritten: einen Zeitgeber. Dieser kann wiederum als abstrakter Spieler aufgefasst werden, wird jedoch im Folgenden als Spezialfall behandelt. Der Zeitgeber etabliert sich damit als optionales Element in dem Spielmodell und modifiziert die Definition:

**Definition 4** *Ein Spielmodell ist die Menge aus Symboliken, Ereignissen und Aktionen innerhalb einer abgegrenzten Realität. Es ist beschrieben durch eine Entität Spielwelt sowie mehreren Entitäten zu verfügbaren Spielobjekten, -ereignissen und -aktionen. Ein Spielmodell kann einen Zeitgeber enthalten. Änderungen am Zustand können über Spielereignisempfänger verarbeitet werden.*

Eine aktionsbedingte Zustandsveränderung kann mit einem klassischen Observer-Pattern abgebildet werden [40, S. 326]. Wird dieses Muster der Softwaretechnik als Generalisierung von Spielwelt und -objekten verwendet, kann jedes Element des Spielmodells ereignisbasierte Veränderungen verarbeiten. Ergänzt wird zusätzlich der genannte Zeitgeber. Dieser kann in einem zeitlichen Kontext das Spielmodell verändert und vom Spieler unabhängig arbeiten. Abbildung 2.4 zeigt die Abhängigkeiten der einzelnen Grundelemente erweitert um das Observer-Pattern, sowie einen Zeitgeber in einem Klassendiagramm grafisch.

Eine Middleware zur Abbildung von Spielmodellen könnte eine Basisimplementierung für Spielwelt, -aktionen, -ereignisse, -empfänger und -objekte bieten, sowie ein System zur Ereignisverarbeitung und einen optionalen Zeitgeber, um eine Simulation des Spielmodells, ohne Spielerinteraktion, zu ermöglichen.

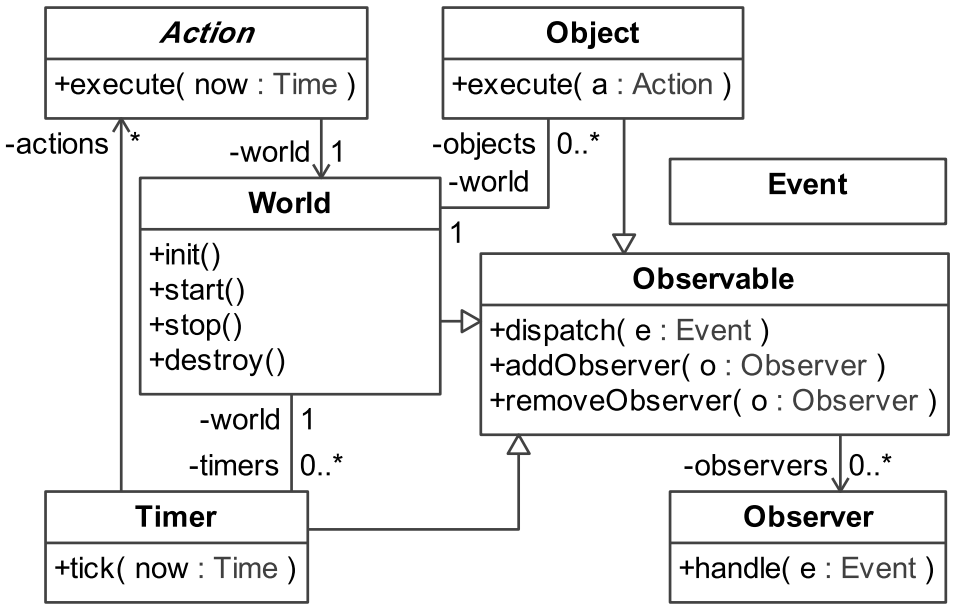



**Abbildung 2.4: Abstraktes Spielmodell für Spiele. Ereignisse werden mittels Ableitungen der Klasse *Event* abgebildet, Ereignisempfänger werden mithilfe des Interfaces Observerable abgebildet.**

## 2.7 Entwicklungsprozess

In Ergänzung zur Betrachtung des Aufbaus und der Funktionsweise eines Online Spiels, wird im Folgenden betrachtet, wie diese Spiele entstehen, bzw. welche Schritte in deren Realisierungsprozess vorkommen. Das Wissen um diesen Prozess und die dabei im Zentrum stehenden Schritte können im Rahmen eines Werkzeuges, welches die Entwicklung vereinfachen soll, Anwendung finden.

Ein Videospiel ist im eigentlichen Sinne eine Software [11, S. 4]. Diese kann mit bekannten Methoden der Softwareentwicklung, wie z. B. Anforderungsanalysen, Anwendungsfallbetrachtungen, Architekturen und Klassendiagrammen [11, S. 86], betrachtet und realisiert werden. Im Unterschied zu einer klassischen Entwicklung kann der Prozess um viele Disziplinen, wie z. B. Sound- und Grafik-Design, sowie Konzept und Skripterstellung, erweitert werden [11, S. 63]. Dadurch erweitert sich der Entwicklungsprozess um die Erschaffung des selbigen. Wie dieser Vorgang verallgemeinert stattfindet, ist in Abbildung 2.5 dargestellt.

Der Vorgang in Abbildung 2.5 beginnt mit einer Vorproduktionsphase, welche dazu dient die eigentliche Spielidee zu umreißen, zu festigen, zu doku-

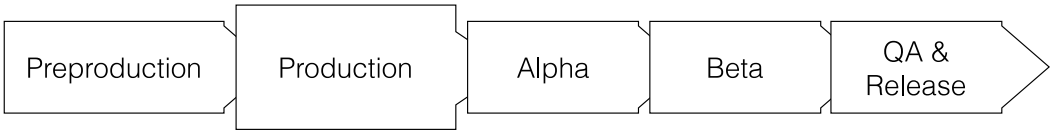



**Abbildung 2.5: Darstellung des Spielentwicklungsprozesses nach [23].**

mentieren und in Prototypen zu realisieren. Mithilfe dieser Prototypen kann anschließend das Projekt vor Investoren präsentiert, sowie in Teilprojekten differenziert und anschließend spezifiziert werden. Teilprojekte im Sinne des Spiels umfassen zum Beispiel Werkzeuge und Ausführungsumgebungen. [23]

Der Vorproduktion in Abbildung 2.5 folgend kommt die eigentliche Produktionsphase. Diese vertieft die Erschaffung des Spiels. Im Detail betrifft dies die Implementierung der Werkzeuge, der Architektur und der Komposition der einzelnen Elemente in eine Gesamtlösung. Als Ergebnis steht an dieser Stelle ein inhaltsloses Spiel, dass alle Anforderungen der Idee berücksichtigt, jedoch noch keine Inhalte abbildet. [23]

Inhalte, im Allgemeinen das was das Spiel dem Spieler abverlangt und präsentiert, wird mit Beginn der Alphaphase umgesetzt. Die aus der Produktionsphase stammenden Werkzeuge können genutzt werden, um in dieser Phase die eigentlichen Level und Herausforderungen im Spiel zu realisieren. Die Alphaphase liefert eine spielbare Anwendung, welche noch ungetestet und unbalanciert ist. [23]

Abgeschlossen wird die Entwicklung mit der Betaphase. Diese dient zum Balanzieren und Testen des Spiels im Gesamten. Tests beschreiben hier nicht Softwaretests im Sinne der Funktionalität, sondern Tests im Sinne dessen was das Spiel dem Spieler abverlangt und präsentiert. Es soll getestet werden, ob das Spiel spielbar ist. Die Inhalte schlüssig und die Herausforderungen interessant sind. Es wird getestet, ob die einzelnen Spielobjekte in einem sinnvollen Verhältnis stehen. Das Ergebnis dieser Phase ist eine spielbare Anwendung, das ein vollständiges Softwareprodukt ist, welches die initiale Idee realisiert. [23]

Am Ende der Spielentwicklung steht der Release. Dieser dominiert durch Qualitätskontrolle, Abnahme der Software und Veröffentlichung über verschiedene Distributionskanäle. Bezogen auf ein browserbasiertes MMOG wäre dies die Veröffentlichung im Internet und die Verknüpfung mit diversen Plattfor-

men, damit das Spiel für den Spieler angeboten werden kann. [23]

## 2.8 Anforderungen

Wird ein Spiel im Sinne des beschriebenen Entwicklungsprozesses erstellt, stellt sich die Frage nach den Anforderungen an diese Entwicklung: Was sind die minimalen und notwendigen Voraussetzungen zur Realisierung und Bereitstellung der Spielidee auf einer Plattform, über die das Spiel angeboten wird.

Für browserbasierte MMOGs kommen, basierend auf den bisherigen Betrachtungen, sechs Schwerpunkte zum Tragen: ein Spielmodell (1), Kommunikation zwischen Spieler und zentraler Instanz (2), die Ausführung des Spielmodells (3), die Persistenz (4) des Spielmodellzustandes, die visuelle, funktionelle und auditive Ausgestaltung des Clients (5) und die Dienste (6), die dem Spieler zum Spiel angeboten werden. Diese sechs Schwerpunkte leiten sich aus der Betrachtung in Kapitel 2.3 (S. 10) und in Kapitel 2.4 (S. 11) ab. Beispiel 3 wendet diese Schwerpunkte auf ein existierendes browserbasiertes MMOG an.

**Beispiel 3** *Die Plattform des Spiels Travian [72] der Firma Travian Games GmbH, zum Beispiel, besteht aus dem eigentlichen Spiel, welches als Webseite mit interaktiven Elementen genutzt wird, einem Forum zum gegenseitigen Austausch für Spieler, einer Bezahlschnittstelle für erweiterte Funktionalitäten im Spiel und einem Supportdienst für Probleme zum Spiel. Client und Server kommunizieren über HTTP in Form von Webseitenaufrufen oder mit asynchronen Techniken innerhalb einer Webseite, wie Asynchronous JavaScript and XML (AJAX), um den Inhalt zu aktualisieren. Der Server ist zum großen Teil in PHP: Hypertext Preprocessor (PHP) implementiert und speichert Daten in eine Datenbank. Alles zusammen, die Spielplattform, bezeichnet die Menge an Diensten, Schnittstellen und Funktionalitäten, die dem Spieler für das Spiel zur Verfügung stehen.*

Während visuelle, funktionelle und auditive Ausgestaltung der Clientanwendung und das Spielmodell individuell je Spielidee zu finden sind, werden im Folgenden auf die Kommunikation, die Persistenz, die Dienste und die Ausführung von Spielmodellen eingegangen. Diese erscheinen bei der Betrachtung verschiedenen Spielen als notwendige und konstante Elemente.

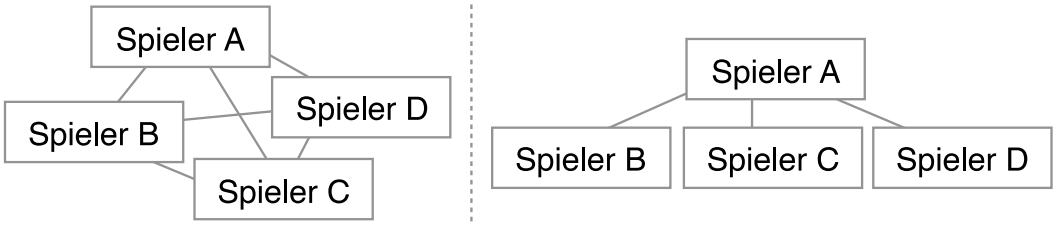



**Abbildung 2.6: Vgl. Peer-To-Peer (links) und Client-Server (rechts) Organisation von Spielteilnehmern. [9, S. 16]**

### 2.8.1 Kommunikation

Für browserbasierte MMOGs ist der Austausch von Informationen von zentraler Bedeutung. Wie wird in Spielen zwischen Clients und Server kommuniziert, was wird kommuniziert und womit wird kommuniziert? Welche verschiedenen Kommunikationsarten gibt es und welche spezifischen Merkmale von Spielen in verschiedenen Genre müssen beachtet werden?

In den ersten Spielen, in denen mehrere Teilnehmer gegeneinander antraten, gab es keine Frage nach der Kommunikation – die Spiele wurden an einer physischen Maschine ausgetragen und jedem Teilnehmer wurde ein gewisser Teil der verfügbaren Peripheriegeräte zugeordnet. Zum Beispiel erhält Spieler A die Pfeiltasten zur Steuerung seines Avatars und Spieler B die Tasten A, W, S und D. In den folgenden Evolutionsstufen kamen zu diesen sogenannten Hot-Seat-Spielen erste Multiplayer Lösungen mit mehreren physischen Maschinen. Diese konnten in zwei grundlegenden Architekturansätzen organisiert werden. Entweder jeder Teilnehmer ist gleichgestellt (1) und kennt jeden anderen Teilnehmer (Peer-To-Peer) oder es gibt einen Hauptknoten (2), der das Spielmodell pflegt und an alle anderen Teilnehmer Informationen weiterleitet (Client-Server). Abbildung 2.6 stellt die Peer-To-Peer und Client-Server Ansätze gegenüber. [9, S. 15ff]

Während eine Peer-To-Peer-Lösung die Ausfallsicherheit erhöht, ist sie im Fall einer Realisierung der aufwendigere Ansatz. Jeder Teilnehmer führt für sich eine unabhängige Instanz der Spielanwendung. Nichtdeterministische Änderungen, wie Spielerinteraktionen, müssen allen anderen Teilnehmern bekannt gemacht werden, damit deren lokale Anwendung zum globalen Zustand synchron bleibt. Quellcode 2.1 zeigt in einem Beispiel die steigenden Nutzerzahlen und die dabei entstehenden Nachrichtenmengen sowie das Datenvolumen, wel-

| *Spieler* | *Nachrichten / Sekunde* | *Datenvolumen / Sekunde* |
|---:|---:|---:|
| 2 | 10 | 600 |
| 4 | 60 | 3.600 |
| 16 | 1.200 | 72.000 |
| 32 | 4.960 | 297.600 |
| 64 | 20.160 | 1.209.600 |
| 128 | 81.280 | 4.876.800 |

**Tabelle 2.1: Betrachtung steigender Spielerzahlen in einer Peer-To-Peer Anwendung bei konstanten 5 Nachrichten je Spieler je Sekunde, welche mit minimalem TCP [63, S. 15] und IPv4 [64, S. 11] Header und einer fixen Größe von 20 Byte versendet werden.**

ches an die Teilnehmer gesendet wird. Wenn in diesem Beispiel die Anzahl an agierenden Spieler linear steigt, kann das Datenvolumen schnell nicht mehr ohne zusätzliche Maßnahmen bewältigt werden. Peer-To-Peer-Multiplayerspiele kommen nach eigenen Erfahrungen häufig bei geringen Nutzerzahlen zum Einsatz, bei denen ein nicht zentrischer Ansatz sinnvoll ist und Kommunikationsschwierigkeiten durch Redundanz ausgeglichen werden sollen. [9, S. 15ff]

Für Anwendungen mit Nutzerzahlen, die die Größenordnungen einer Peer-To-Peer-Anwendung übersteigen, sind Architekturen von Nöten, die Informationen gefiltert und an interessierte Empfänger überstellt. Diese Client-Server-Architekturen sind in ihrer Belastbarkeit durch die physischen Merkmale der Rechenmaschine des Servers begrenzt. Der zentrale Server könnte in dieser Architektur ab einer gewissen Schwelle das Gesamtsystem ausbremsen. Diese Grenzen lassen sich durch ein Hybridmodell, bei der auf der Serverseite eine Peer-To-Peer- oder eine Client-Server-ähnliche Struktur zum Einsatz kommt, erweitern. Diese Erweiterung betrifft im Allgemeinen das Verteilen der Aufgaben auf mehrere physische Maschinen und die Etablierung einer Kommunikation zwischen diesen Maschinen. [9, S. 15ff]

Neben der Struktur des Kommunikationsnetzes sind zwei unterschiedliche Strategien bei der Kommunikation anzutreffen. Die erste Variante ist eine Kommunikation, bei der der Client Anfragen an einen Server stellen kann, um spielbezogene Informationen zu beziehen. Diese Variante kann als unidirektionalen Nachrichtenaustausch bezeichnet werden. Möchte der Client wissen, wo Gegenspieler zu finden sind, stellt er die Frage an den Server und erhält eine Antwort mit den zugehörigen Details, z. B. wo Gegenspieler zu finden

sind. Die zweite Variante erlaubt, zuzüglich zu den clientseitigen Anfragen, Informationen ohne Anfrage an Clients zu übermitteln. Diese Variante kann als bidrektionaler Nachrichtenaustausch bezeichnet werden. Nähert sich zum Beispiel ein Gegenspieler, erhält er die notwendige Information unaufgefordert und kann die Visualisierung entsprechend anpassen.

Wird ein Spiel „Online“ gespielt, jedoch nicht zusammen oder lediglich indirekt mit anderen Spielern, genügt in den meisten Fällen eine Kommunikation, bei der nur clientseitig Anfragen gesendet werden. Alle spielbezogenen Ereignisse treten durch Aktionen des Spielers auf und können zum Beispiel zur Persistierung an einen Server übermittelt werden. Sobald mehrere Spieler an einem gemeinsamen Geschehen beteiligt sind und dieses beobachten, ist es notwendig, Zustandsänderungen an alle Spieler zu verteilen. Dies kann mit regelmäßigen Anfragen des Clients (Polling genannt) simuliert werden, erzeugt jedoch mit steigender Nutzerschaft eine stetig zunehmende Last am Server. Eine Kommunikation, bei der der Server eigenständig Informationen an Clients versenden kann, erscheint nach eigener Betrachtung für Spiele mit mehrere tausend Nutzern und einem sich nicht deterministisch ändernden Spielmodell sinnvoll.

Im Umfeld der browserbasierten MMOGs sind erst seit vollendeter Spezifikation von HTML5 und den damit verbundenen WebSockets eine Plug-in-lose Kommunikation möglich, bei der bidirektional Anfragen und Antworten zwischen Client und Server versendet werden können. Zuvor wurde dies über Browsererweiterungen wie Flash und Unity realisiert.

Aus den vorangegangenen Betrachtungen resultieren folgende Anforderungen zur Ausführung eines browserbasierten MMOGs bezüglich der Kommunikation in Spielen:

- Etablierung einer Kommunikationsinfrastruktur zur Kommunikation zwischen Spieler und Spielwelt.
- Unidirektionalen oder bidirektionalen Nachrichtenaustausch zum Versenden von Anfragen und Zustandsänderungen.
- Parallele Verarbeitung von (tausenden) Spielern in einer geeigneten Architektur.

### 2.8.2 Persistenz

MMOGs sind persistente Spielwelten, die vom Spieler betreten und verlassen werden können. Die Persistenz bezieht sich dabei auf die Sicherung des Zustands, der im Modell verwalteten Entitäten. Eine Strategie ist, dass dieser Zustand in einer Datenbank verwaltet wird und bei Bedarf in die Anwendung geladen wird, um Anfragen von Spielern zu bearbeiten. Eine alternative Strategie ist, dass dieser Zustands innerhalb der Anwendung vorgehalten wird, um den Overhead für das Beziehen des Zustands aus der Datenbank zu vermeiden und um Anfragen schneller bearbeiten zu können. Wird der Zustand in der Anwendung gehalten, entsteht die Notwendigkeit jenen zusätzlich in einem persistenten Speicher abzugleichen.

Der Einsatz von relationalen Datenbanken ist eine verbreitete Strategie im Bereich der browserbasierten MMOGs, wie Gespräche mit Entwicklern auf Konferenzen wie dem Browsergame Forum zeigen. Im Fall von relationalen Datenbanken kann diese Persistierung von Entitäten durch das sogenanntes Object-Relational Mapping (ORM) unterstützt werden. Das ORM hilft dabei Entitäten des Spielmodells auf relationale Datenbanken abzubilden und ermittelt eigenständig, welche Modifikationen notwendig sind, um den Zustand zu synchronisieren. Der Entwickler muss für den Einsatz dieser Werkzeuge zusätzliche Metainformationen in Entitäten hinterlegen, um die Beziehungen aus Sicht der relationalen Datenbank für das ORM abbildbar zu bekommen. Dies bedeutet, dass der Entwickler markiert, welche Entitäten persistiert werden, welches Attribut als ID dient, welche Attribute in welcher Kardinalität auf andere Entitäten verweisen und welche Attribute wie in einem Datenstrom überführt werden müssen. ORM Implementierungen, wie Hibernate und EclipseLink, realisieren das Pradigma „Convention over Configuration“ [20]. Ziel ist es, lediglich den Entitätstyp zu markieren, sowie das zugehörige ID-Attribut bekannt zu geben. Falls es im Speziellen notwendig ist, müssen zusätzlich Beziehungen zu anderen Entitäten markiert werden. Arbeiten darüber hinaus sind nur in Sonderfällen notwendig, was ein ORM-Werkzeug zu einem effizienten Hilfsmittel zur Kopplung eines objektorientierten Modells an eine Datenbank macht. Das folgende Beispiel 4 zeigt jedoch, dass das ORM nicht immer die beste Wahl sein muss.

**Beispiel 4** *In der Implementierung die im Rahmen der Arbeit „Browserbasierte Echtzeit Simulation ‚Evolution'" [8] entstand, wurde für die Persistenz des Spielmodells auf ein ORM-Werkzeug gesetzt. Im Rahmen der Arbeit war die Betrachtung der Ausführungszeit des Modells nicht von Bedeutung und konnte daher vernachlässigt werden. In anschließenden Tests konnte gezeigt werden, dass die Belastungsgrenze für Nutzer je Server bei 10 bis 20 Spielern erreicht war. Dessen Ursache lies sich im Spielmodell und dem zugrunde liegenden ORM-Werkzeug finden. Aufgrund einer modellgetriebenen Entwicklung und durch eine hohe Anzahl an Assoziationen zwischen den verschiedenen Entitätstypen wurden Structured Query Language (SQL) Statements generiert, die mehrere DIN A4 Seiten überspannten. Optimierung in der Anwendung konnte zwar geringfügig die Verarbeitungskapazität erhöhen, jedoch tendierten die Ansätze zur Optimierung das Modell zur besseren Verwendung des ORM-Werkzeuges.*

Das Ergebnis der Lasttests des Anwendungsfalls in Beispiel 4 war nicht überraschend und auf Schwierigkeiten im korrekten Umgang mit dem ORM-Werkzeug, bzw. dem komplexen Spielmodell zurückzuführen. Einfache Anfragen erforderten häufig das große Mengen des Modells aus der Datenbank zu laden waren, um Zusammenhänge korrekt wiederherzustellen. Optimierungen erforderten Modifikationen am Modell selbst, um das ORM-Werkzeug optimal einzusetzen. Ein Spielmodell, das für den Einsatz eines speziellen ORM-Werkzeug angepasst ist, wiederspricht der Idee, dass das Modell unabhängig vom Technologiestack entwickelt wird.

Eine Alternative zum Einsatz eines Datenbankverwaltungssystems und dem ORM wäre das Pflegen von Dateien und deren Ablage auf einem persistenten Speicher. Dabei wird der aktuelle Zustand des Modells in ein zu definierendes Format überführt und das Ergebnis auf dem Speicher hinterlegt.

Aus den vorangegangenen Betrachtungen resultieren folgende Anforderungen zur Ausführung eines browserbasierten MMOGs bezüglich der Persistenz von Spielen:

- Persistenz des Spielmodells, insbesondere das Persistieren des Zustands über Ausführung des Spielmodells hinaus.

- Performante Bereitstellung des Zustandes zur Simulation des Spielmodells oder zu Bearbeitung von Anfragen.

- Synchronisation des Zustandes in eine Datenbank oder ein Dateiformat.

### 2.8.3 Dienste

Zusätzlich zum eigentlichen Spiel existieren in Spielplattformen ergänzende Dienste, die zum Spiel bereitgestellt werden. Dies können z. B. Dienste wie Authentifizierung, Foren, Wikis, Bezahldienste, Nutzerkontenverwaltung, Nachrichten, Statistiken und Blogs sein. Teilweise hängen diese Dienste voneinander ab, sodass zum Beispiel ein Wiki den Authentifizierungsdienst nutzen könnte. Bei der Umsetzung der Spielplattform sind die Entwicklungsstudios meist darauf bedacht, möglichst wenig dieser Dienste selbst zu entwickeln, womit der Bedarf entsteht, dass diese effizient zusammenarbeiten.

Dienste dieser Art werden entweder als Teil des Spieles angeboten oder parallel ausgeführt und über Schnittstellen miteinander verbunden. Dienste fordern demnach die Möglichkeit zur gegenseitigen Interaktion und offenen Schnittstellen. Es ist denkbar, aber nicht zwingend notwendig, dass die Dienste in derselben Umgebung ausgeführt werden, um den Zugriff auf gemeinsame Ressourcen zu optimieren.

Aus der vorangegangenen Betrachtungen resultiert die folgende Liste an Anforderungen zur Ausführung eines browserbasierten MMOGs bezüglich den Diensten zum Spiel:

- Möglichkeit zur Ergänzung der ausgeführten Spielmodelle, um zusätzliche Dienste (z. B. Authentifizierung, Foren, Wikis, Bezahldienste, Nutzerkontenverwaltung, Nachrichten, Statistiken und Blogs).

- Bereitstellung von offenen Schnittstellen zum Dienst für die Möglichkeit zur Interaktion zwischen Spiel und Dienst sowie Dritte und Dienst.

### 2.8.4 Ausführung des Spielmodells

Ein Spielmodell kann eine Orchestrierung von Entitätstypen im Rahmen einer modellgetriebenen Entwicklung sein, bevor es zu einer Konkretisierung der

anzuwendenden Plattformtechnologien kommt. Eigene Beobachtungen zeigen jedoch, dass häufig Serverimplementierung von Spielprojekten einer Ansammlung von Proxies zur Verarbeitung von Kommunikationsanfragen und dessen Anwendung auf eine Datenbank gleicht. Jene Proxies interpretieren Befehle, verarbeitet diese mit Informationen aus einer Datenbank und beantwortet damit die Anfrage. Abschließend werden Änderungen am Zustand in der Datenbank gesichert.

Davon ausgehend, dass ein Entwicklungsprozess das Spielmodell im Zentrum hat, spielt die Verarbeitung von Anfragen eine nachgelagerte Rolle. Das Spielmodell definiert die Spielwelt und die Spielobjekte, das Regelwerk und die Ziele. Die Verarbeitung von Anfragen wird folgend dazu genutzt, um jenes Spielmodell für einen Client nutzbar zu machen. In diesem Fall stellt sich die Frage, wie ein solches Modell ausgeführt werden kann und sich innerhalb einer Plattform integriert. Es stellt sich ebenso die Frage, wie stark dessen Kopplung an die spezifischen Merkmale dieser Umgebung zum Ausführen ist und wie leicht diese Umgebung wechselbar wäre.

Im Gegensatz zu einem klassischen Webdienst kann ein Spielmodell, insbesondere im Fall von MMOGs, ein sich stetig simulierendes Model sein. Einmal gestartet erfordern Spielkonzepte häufig eine fortwährende Entwicklung des Zustands, welche losgelöst von Nutzerinteraktionen stattfinden (z. B. das agieren von Non-Personal-Characters). Diese Elemente sind im gezeigten abstrakten Spielmodell durch den Zeitgeber und die Ereignisverarbeitung verankert und müssen im Fall der Ausführung des Spielmodells berücksichtigt werden.

Aus den vorangegangenen Betrachtungen resultieren folgende Anforderungen zur Ausführung eines Spielmodellen eines browserbasierten MMOGs:

- Ausführbarkeit eines Spielmodells, insbesondere dessen Spielobjekten innerhalb einer dauerhaft aktiven Spielwelt.

- Verarbeitung von zeitbasierten und bedingten Ereignissen.

- Verarbeitung von nutzerbasierten Ereignissen.

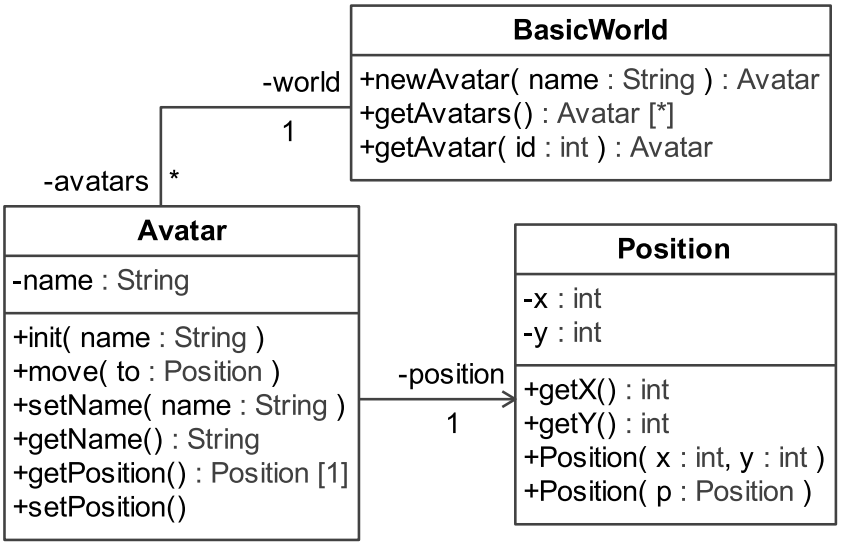



**Abbildung 2.7: Minimales Spielmodell reduziert auf eine Spielwelt und Spieler die agieren können.**

## 2.9 Minimales Beispiel eines Spielmodells

Für die folgenden Betrachtungen, insbesondere der Architektur und Implementierung, stellt sich die Frage, wie die Anwendung des abstrakten Spielmodells aussieht. Bezüglich der grundlegenden Elemente, die in Kapitel 2.6 (S. 14) vorgestellt wurden, wäre das Minimum die mindestens einmalige Verwendung der definierten Elemente Spielwelt und Spielobjekt. Die Spielwelt und Spielobjekte könnten um Spielereignis, Spielereignisempfänger und Spielaktion erweitert werden; das folgende Beispiel verzichtet jedoch zugunsten einer verständlicheren Implementierung auf die Realisierung dieser Elemente. Die Abbildung 2.7 zeigt dieses minimale Spielmodell in einem Klassendiagramm. Dieses enthält die *BasicWorld* als Spezialisierung der Spielwelt und den *Avatar* als Spezialisierung eines Spielobjektes sowie ein Entitätstyp zur Beschreibung einer Position.

Die primären Entitätstypen des Modells sind *BasicWorld* und der *Avatar*, welcher sich in dieser Welt bewegt. Das Spielobjekt *Avatar*, der erste Entitätstyp, enthält dafür einen Namen, eine zweidimensionale Position, sowie die Möglichkeit diese Position in einer Bewegung zu ändern. Die Spielwelt *BasicWorld*, der zweite Entitätstyp, enthält eine Liste von existierenden Entitäten des Typs *Avatar* und die Möglichkeit neue *Avatar*-Entitäten zu erzeugen.

Dieses minimale Spielmodell in Abbildung 2.7 folgt keinem sinnvollen Kon-

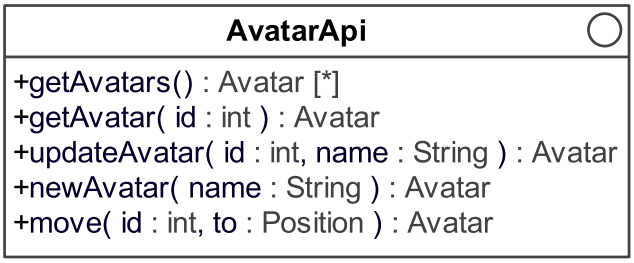



**Abbildung 2.8: Beschreibung der Schnittstelle zur Interaktion zwischen Client und Server basierend auf dem minimalen Modell in Abbildung 2.7.**

zept, es reduziert das Spielmodell lediglich auf ein Mindestmaß an Entitätstypen im Sinne der definierten Elemente in Kapitel 2.6 (S. 14). Der Spieler kann der Spielwelt beitreten und sich in ihr bewegen, Aktionen darüber hinaus sind für dieses Modell nicht notwendig. Die hierfür notwendige Schnittstelle, die der Server anbieten muss, ist in Abbildung 2.8 dargestellt.

# 3. Entwicklung von Massively Multiplayer Online Games

In der browserbasierten MMOG-Entwicklung dominiert der Einsatz von Java, .Net oder PHP sowie seit kurzer Zeit JavaScript als Technologie zum Umsetzen von webbasierten Diensten. Im Folgenden wird betrachtet, wie Daten zwischen den Spielern und den Spielwelten – bzw. Client und Server – ausgetauscht werden. Dabei stehen Technologien für die Überführung von Objektinstanzen aus einer Anwendung in transportierbare Datenströme, Technologien für den Austausch der Daten zwischen Client und Server sowie mögliche Ausführungsumgebungen für Spiele im Fokus.

## 3.1 Abbildungsformate für Anwendungsdaten

Damit Informationen in heterogenen Umgebungen ausgetauscht werden können, sind für den Transport von Daten aus Anwendungen Abbildungsformate notwendig. Der Prozess zur Umwandlung von Daten einer Anwendung (z. B. Objektinstanzen) in ein transportierbares Format wird Marshalling, bzw. Serialisierung genannt [24, S. 172]. Die Rückumwandlung nennt sich Unmarshalling, bzw. Deserialisierung [24, S. 172]. Die im Umfeld von webbasierten Systemen häufig verwendeten Abbildungsformate sind Extensible Markup Language (XML) und JavaScript Object Notation (JSON). Diese zeichnen sich durch ihre gute Integration in verschiedene Programmiersprachen und ihre hohe Verbreitung aus.

Die Anwendung eines Verfahrens zur Abbildung von Anwendungsdaten auf ein transportierbares Format bedeutet einen zusätzlichen Aufwand für die Entwicklung einer Anwendung. Die in der eigenen Anwendung vorhandenen Objekte, sowie deren Attribute müssen, wenn sie transportiert werden, manuell

```
1 ws ::= (#x20 | #x9 | #xD | #xA)+
2 string ::= #x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#xFFFD]
      | [#x10000-#x10FFFF]
3 name ::= ([A-Z] | [a-z] | [0-9] | '_' ) ([A-Z] | [a-z] |
     [0-9] | '.' | '-' | '_' )*
4
5 document ::= prolog element
6 prolog ::= '<?xml version="1.0" encoding="UTF-8"?>'
7 element ::= (tag | string | commend) ws
8 commend ::= '<!--' ws string ws '-->'
9 tag ::= startTag element endTag
10 startTag ::= '<' name (ws attribute)* '>'
11 endTag ::= '</' name '>'
12 attribute ::= name '=' value
13 value ::= '"' (string) '"'
```

**Quellcode 3.1: Gekürzte BNF eines XML Dokumentes. Gekürzt wurden zusätzliche Deklarationen und verschiedene Regelungen wann welches Zeichen erlaubt ist. Eine ausführliche Variante kann in der Quelle nachgelesen werden. [16]**

abgebildet werden. Abhilfe schaffen Werkzeuge, die aus vorhandenen Informationen einer Anwendung die Abbildung automatisieren – bzw. abstrahieren. Im Folgenden wird eine Einleitung zum Thema XML und JSON und der Spezifikation Java Architecture for XML Binding (JAXB), ein Java Werkzeug zur abstrakten Abbildung von Klassen in XML und JSON, gezeigt. Abschließend folgt ein Einblick in die in Java integrierte Serialisierung. Diese ist durch die teilweise native Implementierung performant, jedoch auf Java Anwendungen begrenzt. Zusätzlich wird das Abbildungsformat Protocol Buffers (ProtoBuf) betrachtet, welches aktuell als ein performantes und kompaktes plattformunabhängiges Format zur Verfügung steht.

### 3.1.1 Extensible Markup Language

XML, das erste betrachtete Abbildungsformat, definiert eine Menge an Regeln für semantische Tags (z. B. `<xyz>...</xyz>`), die ein Dokument in Teile zerbricht und hilft die verschiedenen Teile eines Dokuments zu verstehen. XML ist eine Meta-Markup Sprache und definiert einen Syntax zur Definition von domänenspezifischen, semantischen und strukturierten Markup Sprachen [47, S. 3]. Der Syntax von XML wurde in Quellcode 3.1 verkürzt in einer BNF

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <haus>
3   <adresse>
4     <strasse>Beispielweg</strasse>
5     <nr>3</nr>
6     <plz>12334</plz>
7     <stadt>Jena</stadt>
8   </adresse>
9   <raum nummer="3242">
10    <tuer id="123" />
11    <fenster id="42" />
12  </raum>
13  <!-- Abstellkammer -->
14  <raum nummer="1234" />
15 </haus>
```

**Quellcode 3.2: Beispiel XML Dokument.**

beschrieben. Dieser beschreibt, neben Spezialfällen, die sogenannten Tags, wie jeder von ihnen Key-Value-Pairs (z. B. `key="‘value"’`) enthalten kann, sowie die Möglichkeit einem Tag Kinder zuzuordnen [47, S. 53ff]. Die vollständige Version kann in [16] nachgelesen werden und ergänzt detailliertere Angaben für Zeichenrestriktionen in speziellen Kontexten. Zum Beispiel müssen in einem Textknoten verschiedene Sonderzeichen maskiert werden.

In Webanwendungen wird XML verwendet, um den Datenaustausch zwischen Client und Server zu realisieren. Die hierarchische Struktur ermöglicht eine Strukturierung der Daten und erhält während des Transports die Lesbarkeit der Pakete.

Quellcode 3.2 zeigt ein Beispiel, in dem im Dokument ein Haus beschrieben wird. Angegeben werden Informationen zur Adresse und den verfügbaren Räumen. Diese Dokumente sind eine Ansammlung verschiedener Start- und End-Tags. Start-Tags können Attribute haben. Hat ein Element keine ihm untergeordneten Elemente, kann eine Kurzform verwendet werden (z. B. `<xyz></xyz>` wird zu `<xyz/>`). Darüber hinaus zeigt das Beispiel die häufigsten Elemente eines Dokumentes:

- In Zeile 1 ist eine Verarbeitungsinstruktion, diese gibt dem Compiler an, wie das Folgende zu lesen ist. Dies bedeutet jedoch nicht, dass das Dokument korrekt codiert ist, Sonderzeichen muss der Autor eigenständig im korrekten Charset eingeben [47, S. 65].

- In Zeile 2 ist das Root-Element. Ein XML Dokumente sollten immer mit einem Hauptknoten beginnen. Dieses Element enthält alle anderen Elemente des Dokuments [47, S. 66].

- In Zeile 3 wird das Kind-Element *adresse* vom Root-Element *haus* definiert. Dieses enthält weitere Kind-Elemente wie *strasse*, *nr*, *plz* und *stadt*. Die Information, z. B. zur Stadt, wird als Kind-Element vom Typ Text am Element definiert, z. B. `<stadt>` mit dem Text-Kind „Jena“.

- In Zeile 9 wird das Kind-Element *raum* in dem Root-Element *haus* definiert. Dieses Element *raum* definiert ein Attribut *nummer* mit dem Wert „3242“.

- In Zeile 13 wird das Kommentar-Element im Root-Element *haus* definiert.

XML kann für die Verarbeitung als Baum repräsentiert werden. Jeder Knoten stellt ein Element in dem XML-Dokument dar. Die Wurzel des Baums repräsentiert das Dokument selbst. Attribute, Kommentare, Text und Verarbeitungsinstruktionen werden als Knoten dargestellt. Diese Baumdarstellung dient als Basis. Document Object Model (DOM) greift diese Form der Repräsentation auf und erlaubt die Navigation durch das XML Dokument auf Basis von Kind-, Vater- und Geschwisterbeziehungen. Die XML Path Language (XPath) bietet eine Abfragesprache auf dem DOM und erlaubt das Auswählen von Knoten – basierend auf einer Pfadbeschreibung die – zur Aufnahme derer in eine Ergebnismenge. Abbildung 3.1 zeigt das Beispiel aus Quellcode 3.2 überführt in eine Baum-Darstellung.

Ein Vor- und zugleich Nachteil von XML ist, dass Daten „humanreadable“ abgebildet werden. Elementare Datentypen wie Ganzzahlen, Fließkommazahlen und Wahrheitswerte werden als Zeichenketten codiert. Das führt dazu, dass das Datenpaket mit einem beliebigen Editor erzeugt und betrachtet werden kann. Dies führt jedoch auch dazu, dass z. B. eine 8 Bit Ganzzahl mit bis zu 24 Bit codiert wird. Eine 64 Bit Ganzzahl wird mit bis zu 160 Bit codiert. Zusätzlich zu der zeichenkettenbasierten Repräsentation der Zahlenwerte kommt durch den eigentlichen Syntax von XML einen Overhead zur Strukturierung

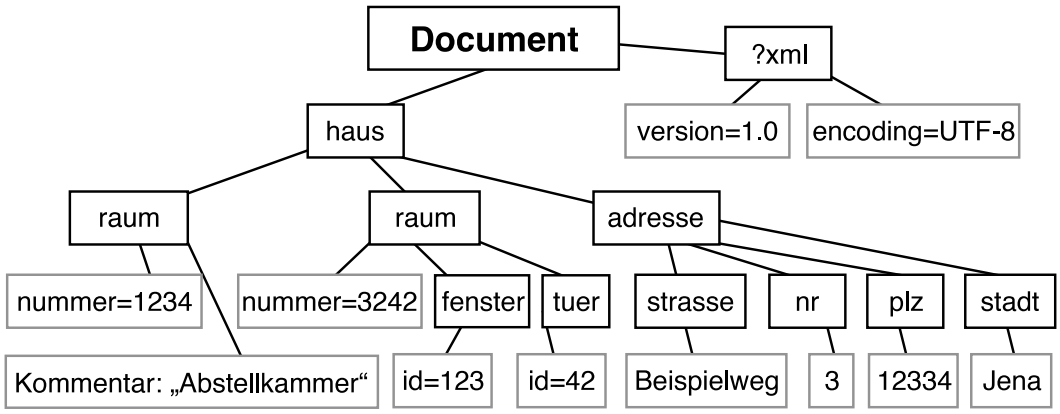



**Abbildung 3.1: Darstellung des XML Dokuments in Quellcode 3.2 als Baum. Jedes Element (Tag, Attribute, Kommentar, Verarbeitungsinstruktion, Text, usw.) wird als Knoten dargestellt. Hervorgehoben sind Tags und die Wurzel.**

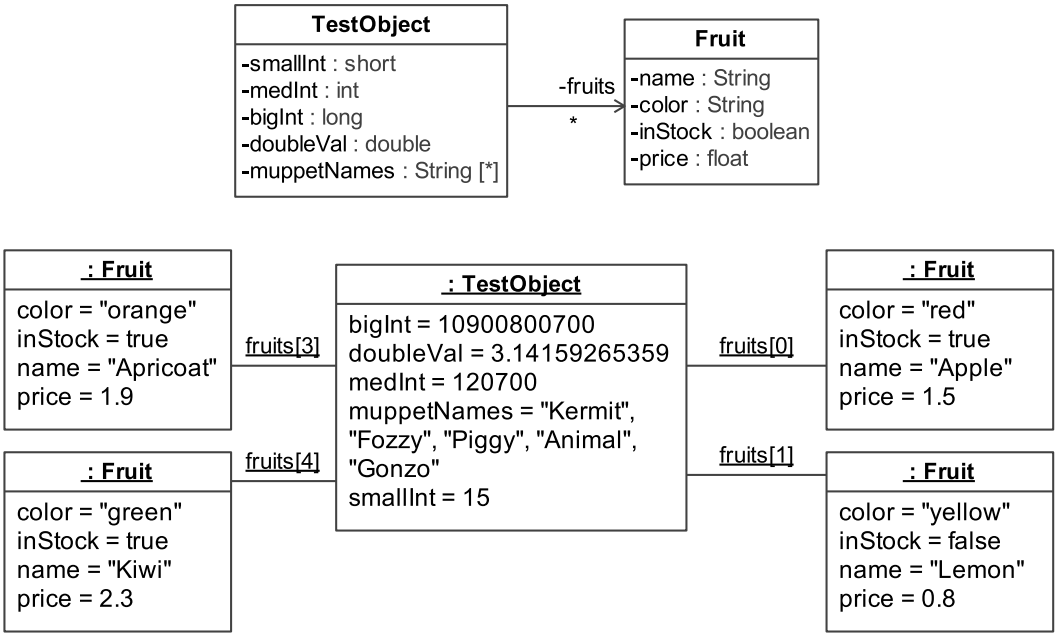



**Abbildung 3.2: Beispiel Kommunikationsobjekte *TestObject* und *Fruit* [45].**

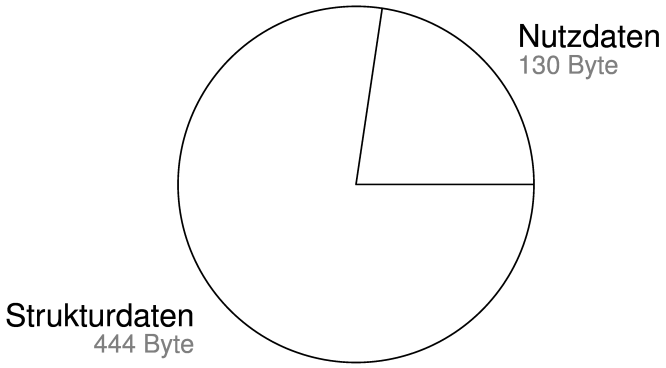



**Abbildung 3.3: Verhältnis der Bytes im XML-Datenstrom bezüglich Struktur- und Nutzdaten im Beispiel aus Abbildung 3.2.**

des Dokuments hinzu. Dieser Overhead kann ein einfaches Datenpaket schnell aufblähen.

Abbildung 3.2 zeigte ein Beispiel mit Instanzen der Klasse *TestObject* und *Fruit*. Werden diese Instanzen in XML überführt, entsteht ein Ergebnis ähnlich dem in Anhang C (S. 239). Bei der Betrachtung wird zwischen Nutz- und Strukturdaten unterschieden. Nutzdaten sind variable Inhalte, welche abhängig von der abgebildeten Instanz sind, z. B. Werte von Attributen. Strukturdaten beschreiben fixe Inhalte, welche auf die Klasse zurückzuführen sind, z. B. Attributnamen und Datentypen. Die Analyse des Datenpakets hinsichtlich Nutz- und Strukturdaten – bei dem alle Leerzeichen, Tabulatoren und Zeilenumbrüche ignoriert wurden – ergibt, dass die Nutzdaten im Vergleich in Abbildung 3.3 ca. 23 % bzw. 130 Byte des Paketes einnehmen. Die verbleibenden 77 % werden durch Regeln im Syntax und Attributinformationen, z. B. Namen von Attributen, gefüllt. Das gesamte Dokument zur Beschreibung der Instanzen in Abbildung 3.2 nimmt 574 Byte ein.

### 3.1.2 JavaScript Object Notation

JSON ist ein leichtgewichtiges, textbasiertes, sprachunabhängiges Datenaustauschformat. Der Syntax stammt aus dem Standard für ECMAScript Programmiersprachen und definiert eine Menge an Regeln für den einfachen Aus-

```
1 begin-array ::= ws %x5B ws ; [
2 begin-object ::= ws %x7B ws ; {
3 end-array ::= ws %x5D ws  ; ]
4 end-object  ::= ws %x7D ws  ; }
5 name-separator ::= ws %x3A ws  ; :
6 value-separator ::= ws %x2C ws  ; ,
7 ws ::= (#x20 | #x9 | #xD | #xA)+
8 string ::= '"' (#x9 | #xA | #xD | [#x20-#xD7FF] | [#xE000-#
      xFFFD] | [#x10000-#x10FFFF]) '"'
9
10 document ::= object | array
11 object ::= begin-object (member (value-separator member)*)?
      end-object
12 member ::= string name-separator value
13 value ::= false | null | true | object | array | number |
      string
14 array ::= begin-array (value (value-separator value)*)? end-
      array
```

**Quellcode 3.3: Vereinfachte BNF des JSON Syntax [26].**

tausch von Daten in strukturierter Form. Quellcode 3.3 zeigt die BNF eines JSON Dokumentes. Dieser ist gekürzt und verzichtet auf eine hohe Detailtiefe, z. B. bei speziellen Zeichenrestriktionen. Die vollständige Version kann im JSON-Standard Request for Comments (RFC) 4627 eingesehen werde. [26]

Da JSON eine Ableitung aus dem Syntax zur Definition von Objekten im ECMAScript Standard ist, erklären sich die Ähnlichkeiten zum JavaScript Sprachsyntax. JSON wird in der Art verwendet, wie in JavaScript Objekte definiert werden können.

Ein Schwachpunkt in JSON ist der Mangel an Möglichkeiten zur Datenpaketidentifikation. Während bei XML Dokumente mit dem Root-Element und dessen Name eine eindeutige Identifikation möglich ist, beginnt dieses Dokument mit einer `{` oder `[` und fährt anschließend direkt mit Inhalten fort. Entwickler behelfen sich hier, indem Sie eine Zwischenstufe einfügen oder ein Attribut zur Identifikation ergänzen.

Ebenso wie XML liefert JSON Daten als Zeichenkette aus und unterliegt damit einer ähnlichen Expansion des Datenvolumens. Binary JSON (BSON) [54], eine Spezifikation, die den JSON-Standard anpasst, adressiert diesen Umstand und bildet Datentypen ohne den Umweg über Zeichenketten direkt ab. BSON ist nicht Teil der JSON-Spezifikation und hat daher eine begrenzte Reichweite.

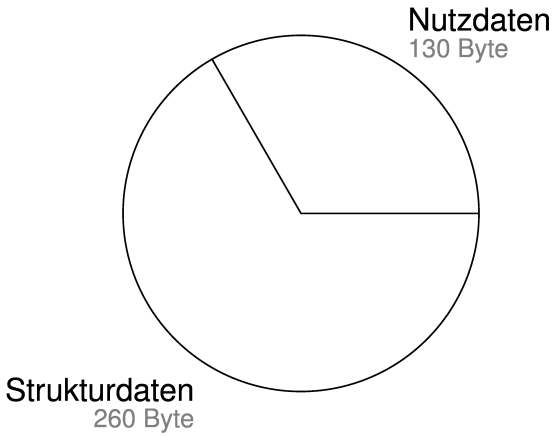



**Abbildung 3.4: Verhältnis der Bytes im JSON-Datenstrom bezüglich Nutz- und Strukturdaten im Beispiel aus Abbildung 3.2.**

Das Beispiel aus Abbildung 3.2 erzeugt im Fall der Serialiserung der Objektinstanzen mit JSON ein Ergebnis wie in Anhang D (S. 241). Dessen Analyse in Abbildung 3.4 bezüglich Nutz- und Strukturdaten zeigt, dass im Datenpaket ca. 33 % Nutzdaten enthalten sind. Dessen 130 Byte Nutzdaten entsprechen derselben Menge wie in XML. Das Beispiel verdeutlich die kompaktere Notation, den geringeren Overhead und damit ein insgesamt kleineres Datenpaket, im Vergleich zu XML, von 390 Byte.

### 3.1.3 Java Architecture for XML Binding

Will ein Entwickler Daten aus seiner Anwendung in JSON oder XML codieren, kann er dies tun, indem er die Zeichenketten manuell generiert oder auf Abbildungswerkzeuge zurückgreift. Low-Level-Tools wie die Java API for XML Processing (JAXP), bietet einen grundlegenden Zugriff zum Verarbeiten von XML Dokumenten in Java. Ähnliche Werkzeuge lassen sich in anderen Programmiersprachen wiederfinden und bieten den beschriebenen Zugriff auf DOM-Basis. Ein einfaches Datenpaket, wie in Abbildung 3.2, kann mit Werkzeugen dieser Art schnell viele Zeilen Code in Anspruch nehmen, was bei einem komplexen Anwendungsfall, insbesondere in einem Spiel mit ausreichend umfangreichen Inhalt, schnell schwer gewartet und aufwendig werden kann.

Die nächste Abstraktionsstufe nach dem JAXP-Werkzeug ist JAXB. Dieses

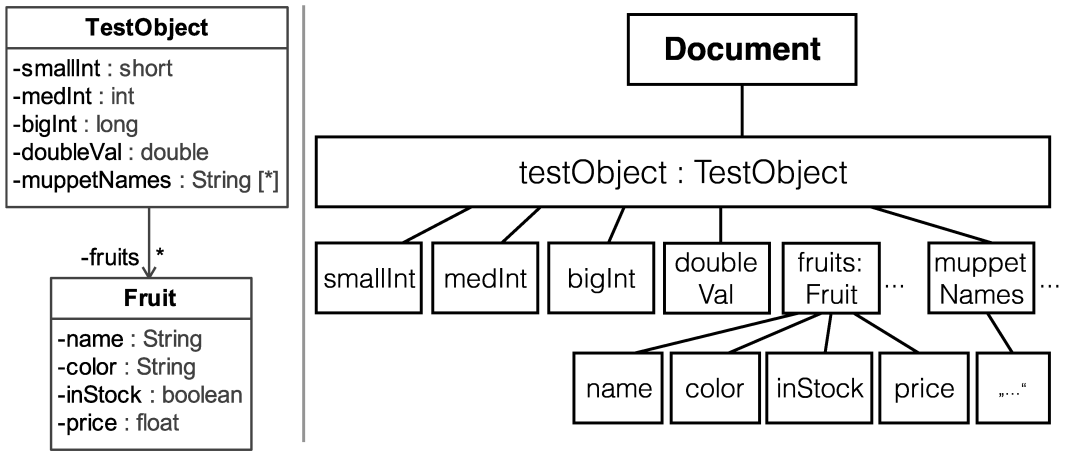



**Abbildung 3.5: Überführen der Klassen aus Abbildung 3.2 auf ein DOM.**

Werkzeug bietet einen Mechanismus, mit dem Objekte in Java in eine XML Struktur überführt werden können, ohne die Abbildung manuell formulieren zu müssen. Für die Application Programming Interface (API) von JAXB ist es erforderlich, dass die in Java implementierten Klassen mit zusätzlichen Metainformationen ausgestattet werden. In diesem Fall kommen Annotationen zum Einsatz. Diese Annotationen beschreiben, wie eine Klasse im XML-Dokument abzubilden ist. Während viele Informationen nur optional angegeben werden müssen, zeigt die Praxis, dass es besser ist, jedes Attribut (inkl. Namen) zu definieren, um spätere Fehler vorzubeugen.

Abbildung 3.5 zeigt, die zwei einfachen Klassen – *TestObject* und *Fruit* – aus Abbildung 3.2 mit einer gerichteten Assoziation und verschiedenen Attributen, sowie deren Überführung in ein XML-Dokument. Die Klasse, welche serialisiert werden soll, wird im XML-Dokument als Root-Element abgebildet und erhält als Namen einen durch Annotationen definierten oder den Klassennamen. Als Kinder erhält dieses Root-Element alle Attribute der Klasse. Diese Kinder können als Elemente oder – falls es sich bei dem Attributtyp um einen Basisdatentypen handelt – als Attributknoten realisiert werden. Kind-Elemente werden nach dem Attributnamen benannt. Im Fall von Assoziationen mit Multiplizitäten ungleich 1 werden mehrere Kind-Elemente desselben Namens und dem entsprechenden Zustand erzeugt.

### 3.1.4 Java Object Serialization

Neben der Abbildung von Instanzen in Datenströme auf Basis von Standards wie XML und JSON, liefern Entwicklungsumgebungen eigene Verfahren mit. Im Fall von Java kann eine Instanz mittels Java Object Serialization (JOS) serialisiert werden [58]. Über den sogenannten *ObjectOutputStream* lassen sich Datenströme erzeugen, welche ohne zusätzliche Hilfsmittel – im Vergleich zu XML und JSON – schwer lesbar sind.

Damit eine Klasse in Java serialisierbar ist, muss diese die Schnittstelle *Serializable* (oder *Externalizable*) realisieren. Die Schnittstelle selbst hat keine Methodensignaturen und dient in diesem Fall zur Markierung der Klasse als „Sicher zur Abbildung im Datenstrom“ [58]. Serialisiert werden dabei alle Informationen zur Klasse. Die Namen und Typen, der nicht transienten Attributen dieser Informationen, werden im Folgenden als Strukturdaten klassifiziert. Verwendet ein Attribut einen komplexen Datentyp, der nicht zum Java Sprachumfang gehört, so wird dessen Datentypbeschreibung im Paket eingebettet. JOS stellt im Abbildungsverfahren sicher, dass jeder Datentyp nur einmal beschrieben wird, um das Volumen nicht unnötig aufzublähen. Werden durch Assoziationen Instanzen einbezogen, dessen Datentyp nicht die Schnittstelle *Serializable* realisiert, wird der Prozess mit einem Fehler beendet. Auf diese Weise wird verhindert, dass Daten serialisiert werden, die nicht hätten die Grenzen der Anwendung verlassen sollen.

Dem Beispiel aus Abbildung 3.2 folgend, wird in Anhang E (S. 243) das binäre Ergebnis der Serialisierung mit Java als Hex-Block-Schreibweise dargestellt. Der Anhang zeigt einen Datenstrom, der die Beschreibung für die Klasse *TestObject* und *Fruit*, dessen Attribute, sowie deren Namen und Datentypen in voll qualifizierter Schreibweise enthält. Die Nutzdaten der Instanzen können an den markierten Stellen gefunden werden. Abbildung 3.6 zeigt, dass die Nutzdaten einen Anteil von 22 % bzw. 111 Byte im Datenstrom ausmachen. Im Gegensatz zum Datenstrom von XML und JSON sind je Attribut pro Instanz keine Namen zugeordnet. Diese sind implizit durch die Reihenfolge der Attribute in den Datentypenbeschreibung vorgegeben. Trotz des vergleichbaren Datenumfangs ist das Paket insgesamt mit 502 Byte voluminös. Dies begründet sich in den vollständigen Datentypbeschreibungen und relativiert sich, wenn

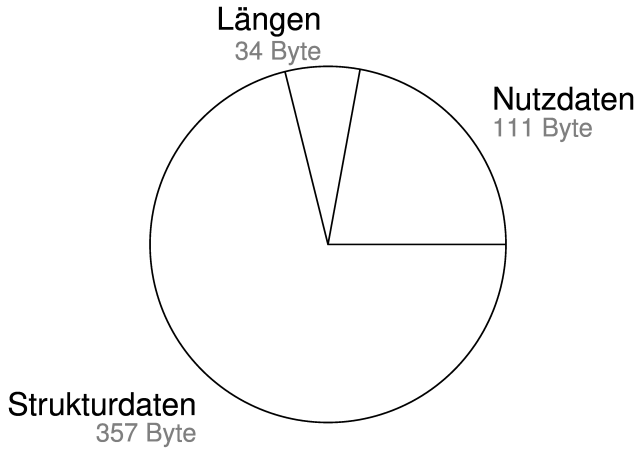



**Abbildung 3.6: Vergleich des Anteils an Nutz- und Strukturdaten bei Verwendung des Abbildungsformates JOS.**

ein Datenpaket viele Instanzen derselben Klasse enthält.

### 3.1.5 Protocol Buffers

Neben den Standards XML und JSON und z. B. Serialisierungsverfahren in Java sind weitere Formate verfügbar, die den Austausch vereinfachen sollen. Ein aktuelles Abbildungsformat, welches auf einen minimalen Overhead und schnelle Abbildung fokussiert, ist ProtoBuf von Google. Dieses Format, z. B. eingesetzt im Spiel Pokémon Go und in [1] per Reverse Engineering dokumentiert, erlaubt mittels Formulierung einer allgemeinen Beschreibung von Nachrichten, der sogenannten Proto-Datei, die Erzeugung des notwendigen Quellcodes in Java, Python, Go, C++ und C#. [41]

Das Format ist auf das wesentliche reduziert und verglichen mit XML drei- bis zehn-mal kompakter. Der Aufbau gleicht Strukturen, wie sie in XML und JSON zu finden sind: Eine Nachricht beginnt mit einer optionalen Längeninformation und enthält folgend Key-Value-Pairs. Die Keys, bzw. Attribute, werden mithilfe einer ID abgebildet und die Values, bzw. Werte, werden basierend auf ihrem deklarierten Typ abgebildet. [41]

Ein einfaches Beispiel kann in Quellcode 3.4 betrachtet werden. Dieses Beispiel definiert eine Nachricht *TesterObject* mit einem Attribut *value* vom Typ *Int32*. Übersetzt man diese Nachricht und bildet das *TesterObject* mit $2^{14}$ für

```
1 package de.fsu.stream;
2 option java_package = "de.fsu.test";
3 option java_outer_classname = "TestProtos";
4 message TesterObject {
5   required int32 value = 1;
6 }
```

**Quellcode 3.4: Beispiel Proto-Datei zur Konfiguration einer Nachricht.**

*value* ab, erhält man einen Datenstrom, der Binär dargestellt $0x0408808001$ entspricht. Das erste Byte $0x04$ beschreit die Länge der Nachricht, das zweite Byte $0x08$ den Typ und die folgenden drei Bytes den Wert $2^{14}$ in einem dynamisch wachsenden Format für Ganzzahlen. Dabei ist der höchstwertigste Bit eines Bytes reserviert zur Markierung, ob das aktuelle Byte das Ende der Kette zur Abbildung der Zahl darstellt. Zur Errechnung des ursprünglichen Wertes muss der höchstwertigste Bit gelöscht werden und die Kette in umgekehrter Reihenfolge zusammengesetzt werden[1]: $((0x1 \wedge 0x7F) \ll 14) \vee ((0x80 \wedge 0x7F) \ll 7) \vee (0x80 \wedge 0x7F) = 2^{14}$. Dieses Verfahren wird für ganzzahlige Attribute und Längen verwendet. Attribute mit Typen wie Fließkommazahlen und Zeichenketten werden wie in JOS abgebildet, z. B. ein Double als 64 Bit Fließkommazahl. Reihungen der genannten Ketten werden auf eine Dimension begrenzt und benötigen im Ausgleich dafür keinen Längenidentifikator. Zusätzliche Ebenen lassen sich durch Nachrichten erzeugen, welche selbst Reihungen enthalten und gereiht verwendet werden. [41]

Anhang F (S. 245) zeigt den Objekt-Datenstrom, der durch die Serialisierung der Instanzen aus Abbildung 3.2 entsteht. Das spezielle Verfahren zur Abbildung von Ganzzahlen ist in dieser Darstellung nicht berücksichtigt, dabei gilt, dass je Byte für Ganzzahlen ein Bit Strukturinformation enthält. Abbildung 3.7 stellt die Auswertung des Nachrichtenpaketes bzgl. Nutz- und Strukturdaten dar. Das mit 154 Byte große Datenpaket enthält im Beispiel 69 % Nutzdaten.

Nachteilig erscheint, dass die Kommunikationsobjekte nicht im Sinn einer objektorientierten, sondern nachrichtenorientierten Entwicklung entstehen. Der Referenz folgend wird empfohlen, dass hier eine zusätzliche Ebene

[1] Der Operator $\ll$ bezeichnet die Schiebeoperation.

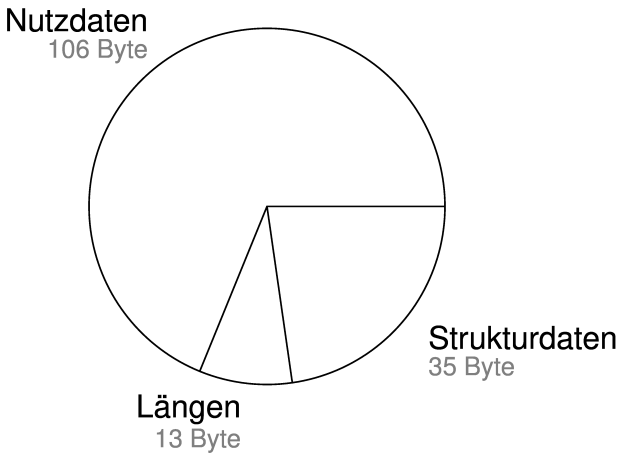



**Abbildung 3.7: Vergleich des Anteils an Nutz- und Strukturdaten bei Verwendung des Abbildungsformates ProtoBuf.**

zu konstruieren ist, welche die Nachrichtenobjekte mittels Wrapper, im Sinn des Adapter-Patterns [40, S. 157], kapselt. [41]

Zusätzlich fällt auf, dass mangels Nachrichtenidentifikator Schwierigkeiten bei der Interpretation von kontinuierlichen Datenflüssen entstehen. Während im Fall von XML und JOS der primäre Typ der Nachricht im Datenpaket verankert ist, verzichtet ProtoBuf, ähnliche wie JSON, auf diese Information. Hierfür ist es notwendig zusätzliche Informationen in den abzubildenden Objekten zu verankern, die eine Wiederherstellung von beliebigen Kommunikationsobjekten ermöglicht.

## 3.2 Schnittstellentechnologien

Der Serialisierung folgt im Fall eines Client-Server-Anwendungsfalls die Datenübermittlung. Diese wird, im Fall von MMOGs, auf Basis des Internet Protocol (IP) [64] basierten Transports durchgeführt. Dabei kommen, neben den direkten Ansätzen auf Basis von TCP [63] oder User Datagram Protocol (UDP) [62], wie z. B. in [56] für ein MMOG realisiert, häufig anwendungsnahe Protokolle wie HTTP oder WebSockets zum Einsatz. HTTP, z. B. im Spiel Travian [72], und WebSockets, z. B. im Spiel Browserquest [51], sind im Fall von browserbasierten MMOGs beliebt, da diese nativ im Browser unterstützt werden. Der folgende Überblick liefert einen Einstieg in die Datentransportschicht TCP

und UDP sowie der Anwendungsschicht HTTP und WebSocket.

### 3.2.1 Datentransportschicht

Wenn nicht mit dem IP Protokoll begonnen wird, dann bietet sich der Einstieg über die Transportprotokolle TCP und UDP des Open System Interconnection (OSI) [49] Modells an. Während das IP-Protokoll die Zustellung eines Datenpaketes an eine sogenannte IP-Adresse sicherstellt, können mit TCP und UDP Daten an Anwendungen an dieser Adresse zugestellt werden – z. B. Spielanwendungen. Beide Protokolle verwenden dafür einen 2 Byte großen Identifikator für Anwendungen – den sogenannten Port. [62] [64]

UDP ist ein kompaktes Transportprotokoll, das auf eine effiziente Datenübertragung ausgelegt ist. Der Header hat ein Volumen von 64 Bit und enthält Informationen wie Quell-Port, Ziel-Port, Länge des Datenpakets (inkl. Header in Oktetts, demnach mindestens 8 Byte und höchstens 65.535 Byte), sowie eine Prüfsumme [62, S. 1]. Dem anschließend folgen, in dem im Header angegebenen Volumen, die transportierten Daten. Für die Initiierung der Verbindung ist kein zusätzlicher Aufwand notwendig. Ist die IP-Adresse des Empfängers und der Port der Anwendung bekannt, kann das Datenpaket versendet werden. Eine Prüfung des Zustellungserfolges wird bei dem Versand mittels UDP nicht sichergestellt. [62]

Eine UDP Kommunikation ist in erster Linie unidirektional. Da das UDP und das übergeordnete IP Paket jedoch Absender IP und Port enthält, ist prinzipiell eine Antwort so lange möglich, wie der Rückweg existiert. Eine bidirektionale UDP-Server-Implementierung besteht aus einer endlosen Verarbeitungsschleife von ankommenden Daten. Diese wird zumeist um eine Sitzungsverwaltung erweitert, die bei Eingang eines Pakets eine neue Sitzung erstellt und diese so lange erhält, bis zu einer bestimmten Zeitüberschreitung keine weiteren Daten empfangen werden konnten. Die Sitzung speichert die IP-Adresse und Port des Absenders und ermöglicht das Versenden von Antworten.

TCP ist, im Gegensatz zu UDP, ein schwergewichtiges Transportprotokoll mit einem Fokus auf die Sicherstellung des Datentransports. Der Header ist durchschnittlich 3-mal umfangreicher als der des UDP Protokolls. Während UDP konstante 64 Bit benötigt, hat der Header von TCP eine variable Größe

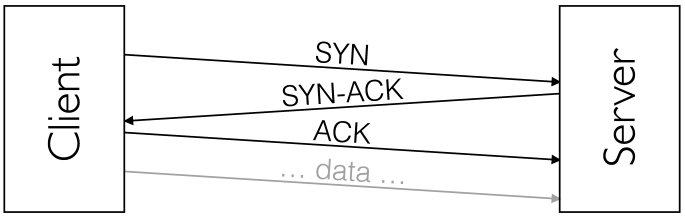



**Abbildung 3.8: Initiierung des TCP basierten Datentransports nach [63, S. 23].**

und ist abhängig von der Anzahl an Optionen. Die Größe des Headers ist trotz Variabilität immer ein Vielfaches von 32, was am Ende des Headers durch das Auffüllen mit 0'en sichergestellt wird [63, S. 19]. Wie der UDP-Header, enthält dieser ebenso Quell-Port, Ziel-Port und Prüfsumme. Ergänzt wird der Header um zusätzliche Optionen und Kontrollbits (z. B. ACK und SYN) [63, S. 15]. Das Volumen der transportieren Daten ist mittels Sequenz-Nummerierung realisiert. Diese positioniert das Paket eindeutig innerhalb einer Sequenz. Die Nummerierung von Paketen innerhalb von Sequenzen zeigt die Vorteile von TCP: das Protokoll erlaubt die Überführung der eintreffenden Daten in die ursprüngliche Reihenfolge, sowie die Prüfung ob diese vollständig zugestellt wurden sind. Fehlen Elemente in der Sequenz erlaubt TCP das erneute Anfragen. [63]

Den zusätzlichen Aufwand, den TCP im Vergleich zu UDP betreibt, wird ebenso beim Verbindungsaufbau sichtbar. Während das UDP-Protokoll keine initialen Prozeduren vorsieht und direkt mit der Zustellung von Daten an den Empfänger beginnt, stellt TCP den korrekten Aufbau der Verbindung über einen Handshake sicher [63, S. 31]. Dieser Handshake, siehe Abbildung 3.8, beginnt mit einem SYN-Paket (markiert mittels Kontrollbit) vom Client zum Server. Der Server beantwortet dieses Paket mit einem SYN-ACK-Paket und der Client erwidert mit einem ACK-Paket. Dem letzten Paket folgend kann der Client direkt mit dem Senden von Daten beginnen.

Eine TCP-Server-Implementierung, wie in [56, S. 216], ist bereits bidirektional und besteht zumeist aus einer Verarbeitungsschleife für neue Verbindungsanfragen zur Annahme des Handshakes, sowie ein oder mehreren Verarbeitungsschleifen für ein- und ausgehende Daten. In einem einfachen Aufbau kann z. B. jeder Verbindung ein eigener Prozess mit entsprechender Verar-

beitungsschleife zugewiesen werden. Dieser Prozess entspricht der aktuellen Client-Sitzung. [56, S. 193ff]

Die Gefahr besteht an dieser Stelle, dass bei hohem Aufkommen zu viele parallele Prozesse entstehen, weswegen sich in Anwendungsfällen mit hohem Verbindungsaufkommen eine aufwendigere Implementierung anbietet. [8]

### 3.2.2 Beispiel-Server mit Java und TCP

Mit dem Wissen um die TCP-Server-Implementierung kann ein Beispiel auf Basis des minimalen Spielmodells aus Abbildung 2.7 und der notwendigen Schnittstelle aus Abbildung 2.8 konstruiert werden. Der folgende Ansatz zeigt eine Realisierung und die notwendigen Elemente.

Für den in Abbildung 3.9 dargestellte Ansatz wurde serverseitig der beschriebene Architekturaufbau aus Abbildung 2.2 umgesetzt. *Server* und *Connection* stellen die notwendigen Klassen für die Sitzungsverwaltung des TCP-Server dar. Während der *Server* in seiner Verarbeitungsschleife (Methode *run*) neue Verbindungen akzeptiert, kann die *Connection* in ihrer Verarbeitungsschleife Datenpakete empfangen und die Verarbeitung im *MessageHandler* anstoßen. Das Spielmodell enthält die in Abbildung 2.7 genannten Entitätstypen; in der Abbildung 3.9 wird jedoch zugunsten der Übersichtlichkeit auf die komplette Darstellung verzichtet. Der *MessageHandler* wird im Beispiel mittels *JsonMessageHandler* als ein JSON-basierter Interpreter von Nachrichten implementiert. Bedingt durch die Typensicherheit in Java ist es notwendig, dass jede Signatur dieser Schnittstelle eine Request- und eine Response-Klasse benötigt. Die Request-Klasse enthält die Argumente und die Response-Klasse die Antwort der Signatur. Während die Request-Klasse im Beispiel für jede Signatur individuell ist, reichen insgesammt zwei Response-Klassen für alle Signaturen. Die Erste beschreibt die öffentlichen Informationen eines Avatars (ID, Name und Position) und die Zweite eine Liste von Avataren. Die Implementierung der Schnittstelle aus Abbildung 2.8 und die Behandlung der empfangenen und interpretierten Objekte wird in der Schnittstelle *AvatarApi* umgesetzt. Diese verwendet die im Request enthaltenen Attribute und den Typ des Request-Objekts um die jeweilige Methode des Spielmodells aufzurufen.

Der Ablauf zur Verarbeitung einer Anfrage wird in Abbildung 3.10 darge-

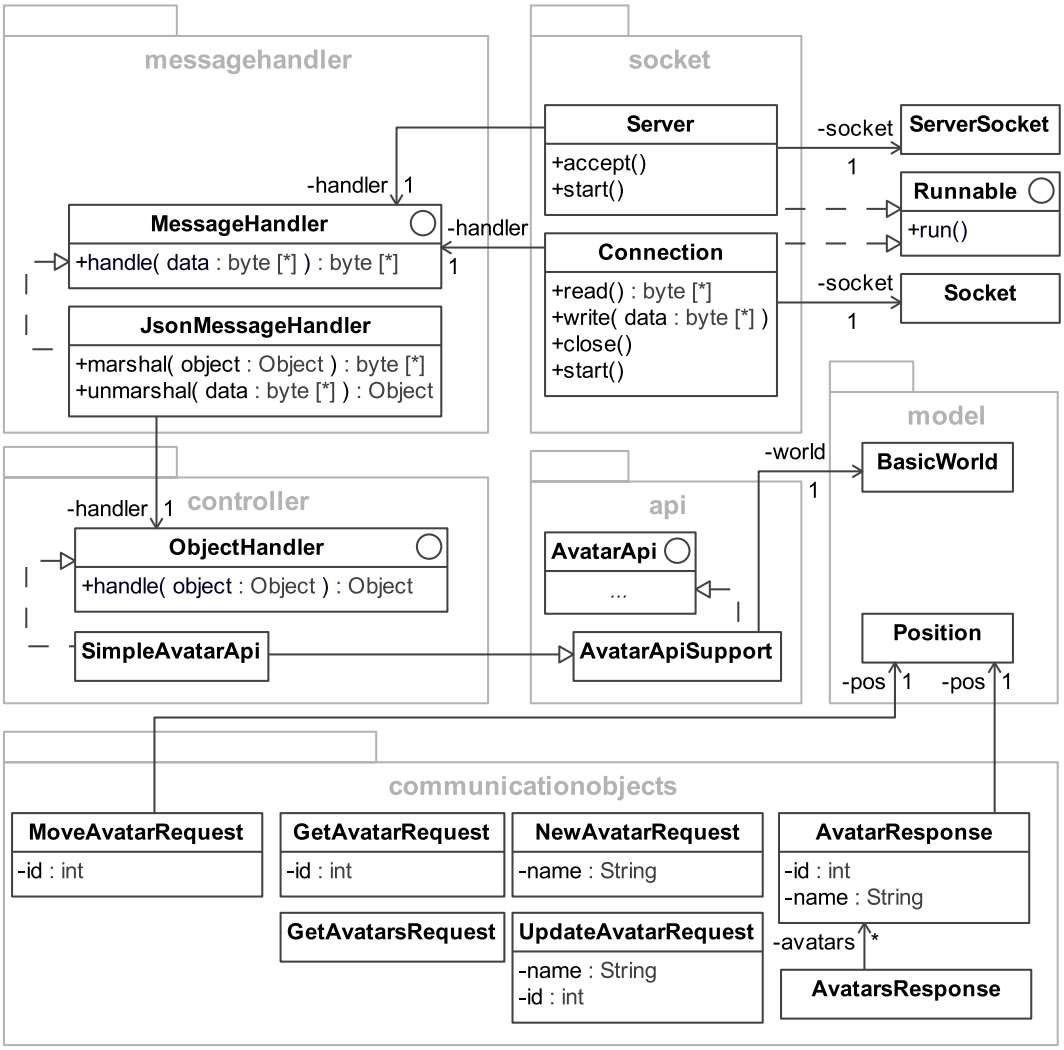



**Abbildung 3.9: Klassendiagramm der Realisierung des TCP-Servers in Java.**

stellt. Beginnend mit dem Eintreffen eines Datenpakets in der Verarbeitungsschleife der *Connection* Instanz über die Methode *read* wird das Paket an den konfigurierten Message Handler übergeben. Im Fall des Beispiels wird dafür der genannte *JsonMessageHandler* genutzt. Dieser deserialisiert das Datenpaket in eine Instanz der zugehörigen Request-Klasse und gibt diese dem assoziierten *ObjectHandler* weiter. Im Beispiel ist dies die *SimpleAvatarApi*, welche die Attribute des deserialisierte Request-Objekts an die korrekte API-Methode übergibt. Abschließend wird der Rückgabewert des Aufrufs in eine Instanz der korrespondierenden Response-Klasse zurückgegeben, durch den Message Handler serialisiert und durch die *Connection* an den Client gesendet.

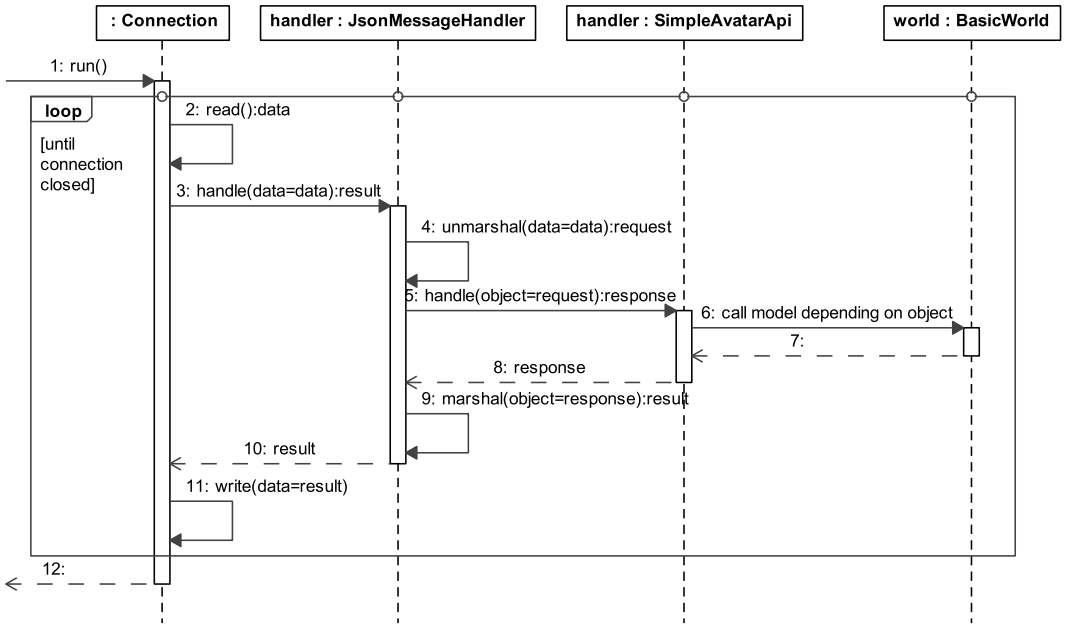



**Abbildung 3.10: Verarbeitungsablauf des TCP-Servers in Java.**

### 3.2.3 Anwendungsschicht

Die Anwendungsschicht setzt auf ein Protokoll der Datentransportschicht auf. Im Fall der MMOGs sind Protokolle wie HTTP und WebSockets auf Basis von TCP von Interesse, falls diese nicht proprietär wie im Beispiel in Kapitel 3.2.2 (S. 46) umgesetzt werden oder UDP zum Einsatz kommt.

HTTP dient für verteilte, gemeinschaftliche, quer verweisende Informationssysteme. Haupteinsatzzweck ist das World Wide Web (WWW) und wird seit 1990 zum Beziehen von Daten genutzt [38]. Während das IP-Protokoll den Empfänger mittels IP benennt und TCP mittels Port einer Anwendung zuordnet, kann mit HTTP eine Ressource innerhalb der Anwendung adressiert werden.

HTTP ist ein zustandsloses Protokoll zum Adressieren einer Ressource, welches dem Request-Response Ansatz folgt. Wenn ein Client einen HTTP-Request senden möchte, führen Client und Server den TCP-Handshake durch, das Paket wird überstellt und der Server verarbeitet dieses. Abschließend wird die Anfrage mit einem HTTP-Response beantwortet und die Verbindung in der Regel geschlossen. Der Standard berücksichtigt für Requests verschiedene Arten (Methods) von Anfragen. Häufige Verwendung finden dabei die GET-Requests, welche die einfachste Art der Anfrage nach einer Ressource darstel-

```
<scheme>://<userinfo>@<hostname>:<port><path>?<query>
http://example.com/index.html
http://max@example.com:8080/authenticate
http://example.com/list/search.php?name=test
```

**Abbildung 3.11: Elemente einer URL und deren Verwendung in Beispielen [10, S. 11].**

len, die POST-Requests, welche die Möglichkeit bieten die angefragte Ressource um den Inhalt der Anfrage zu ergänzen, die PUT-Requests, welche die Möglichkeit bieten die angefragten Ressourcen mit dem Inhalt der Anfrage zu ersetzen, sowie die DELETE-Requests, welche die Möglichkeit bieten die angefragte Ressource zu entfernen. [38, S. 36]

Ein wichtiges Element im Umgang mit der Ressourcen Adressierung in HTTP sind sogenannte Uniform Resource Identifiers (URIs) [38, S. 18]. Hier am bekanntesten sind die Uniform Resource Locators (URLs), welche eine Teilmenge der URIs sind. Sie dienen der Identifizierung von Ressourcen über eine primäre Zugriffstechnik, in diesem Fall HTTP [10, S. 3]. Abbildung 3.11 zeigt den vereinfachten Aufbau einer URL. Das *scheme* ist im Fall von HTTP *http*. Die *userinfo* sind optional und selten notwendig, können jedoch verwenden werden, um den anfragenden Nutzer zu benennen. Der *hostname* bezeichnet die Zielmaschine. Dieser kann als vollqualifizierter Name oder als IP-Adresse angegeben werden. Dem *hostname* folgend kommt der *port*. Ist dieser nicht angegeben (inkl. dem führenden Doppelpunkt), wird der definierte Default-Port für das verwendete *scheme* angewendet. Im Fall von HTTP ist 80 dieser Default-Port. Anschließend folgt der *path*, welcher einen eindeutigen Pfad zur Ressource darstellt, und abschließende eine Menge von Key-Value-Pairs im *query*.

Auf der Basis von HTTP lassen sich weitere Protokolle und Programmierparadigmen aufsetzen, die als Basis zur Konstruktion von verteilten Anwendungen dienen. Zum Beispiel ist Simple Object Access Protocol (SOAP) ein Standard, der (unter anderem) auf Basis von HTTP arbeitet. Der durch das World Wide Web Consortium (W3C) spezifizierte Standard verwendet XML-basierter Deskriptoren, welche es ermöglichen in heterogenen Systemlandschaften den Austausch von Daten zu strukturieren [78]. Auf eine detaillierte Erklärung wird an dieser Stelle verzichtet, da SOAP aus eigener Erfahrung für

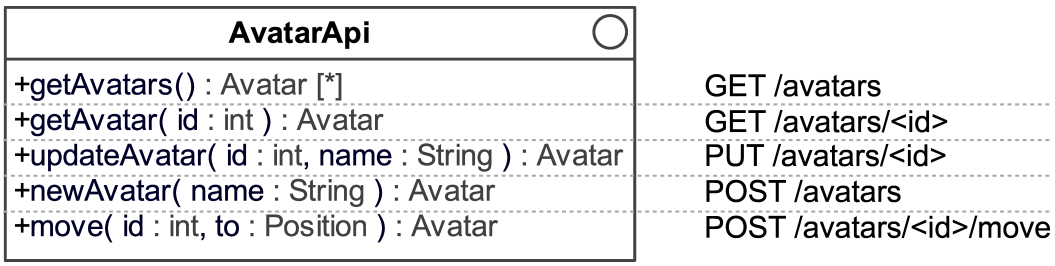



**Abbildung 3.12: Abbildung einer Schnittstelle auf HTTP-Anfragen in REST.**

den Einsatz in den meisten MMOG Spielen ungeeignet erscheint.

Representational State Transfer (REST) ist ein Programmierparadigma, welches ebenso (unter anderem) auf Basis von HTTP arbeitet und weniger schwergewichtig als SOAP ist. Dieses Programmierparadigma beschreibt die Interaktion mit einer Schnittstelle z. B. über das zustandslose HTTP Protokoll. In REST werden dem Pfad und den Arten der Anfrage wohl-definierte Bedeutungen zugeordnet, welche sich auf die Schnittstelle abbilden lassen, und Daten als XML, JSON oder beliebig definiertes Format transportiert [39]. Abbildung 3.12 zeigt, wie die Beispiel-Schnittstelle aus Abbildung 2.8 zur Interaktion mit dem minimalen Spielmodell aus Abbildung 2.7 mittels REST in HTTP realisiert werden kann.

Dem Umstand einer HTTP-basierten Lösung geschuldet ist bei dem Einsatz von REST-Schnittstellen nur eine clientseitige Interaktion mit dem Server möglich. Nach erfolgter Anfrage wird, HTTP bedingt, die Verbindung zumeist geschlossen und der Server kann den Client nicht aktiv informieren, wenn sich am Spielmodell etwas ändert.

Eine Möglichkeit dieses Problem zu adressieren wird mit dem Ansatz Comet in [25] beschrieben. Dabei baut der Client mithilfe einer Anfrage eine Verbindung zum Server auf, dieser hält diese so lange wie möglich offen, indem keine Antwort formuliert wird, und sendet über diese im Fall von Ereignissen das Ergebnis. Wird die maximale Zeitspanne für die offene Verbindung erreicht oder ein Ergebnis zugestellt, sendet der Client eine neue Anfrage, die anschließend erneut vom Server so lange wie möglich gehalten wird. Auf diese Art wird ein Rückkanal zum Client simuliert, der es ermöglicht, Daten zeitnah zuzustellen. [25]

Neben REST ist der Standard für WebSockets [36] im Bereich der browserbasierten MMOGs interessant. Dieser ist seit 2011 festgehalten im RFC Standard 6455 und beschreibt eine Protokoll-Schicht, die für den Einsatz in Webanwendungen auf Basis von TCP gedacht ist. Als Beispiel kann hierfür das von Mozilla beauftrage Spiel Browserquest [51] verwendet werden, welches mittels WebSockets realisiert ist und ohne zusätzlichen Plug-ins in einem Browser arbeiten kann.

Der WebSocket Standard verwendet, in Ergänzung zum TCP-Handschake, einen HTTP basierten Handshake. Inhalt dieses Handshakes über einen HTTP-Request ist der Wunsch des Clients an den Server um ein Upgrade der Verbindung. Bestätigt der Server die Anfrage positiv, beendet der Server die Verbindung zum Client nicht wie üblich nach dem Request-Response-Austausch, sondern hält sie offen für den folgenden Austausch von Datenpaketen. Diese werden im Fall von WebSockets-Dataframes genannt und folgen einer Paketstruktur, die die Nutzdaten maskiert überträgt. [36]

Die mit HTML5 eingeführte Kommunikationsmöglichkeit erlaubt einen sicheren, bidirektionalen Austausch von Daten zwischen einer JavaScript-basierten Anwendung im Browser und einem Server, sowie die Zustellung von Informationen ohne vorherige Anfrage.

### 3.2.4 Beispiel-Server mit JavaScript und REST

REST Schnittstellen sind in den meisten Programmiersprachen schnell umsetzbar. Im Fall von komplexen Anwendungen entwickeln sie sich schnell zu aufwendigen Implementierungen, in denen leicht die Übersichtlichkeit verloren geht.

Die folgende mit JavaScript realisierte Architektur fokussiert auf einen einfachen Server, der das REST-Paradigma verwendet und auf Comet verzichtet. Eine Ausführungsumgebung zur Realisierung von Anwendung auf Basis von JavaScript ist „node.js“ [57]. Die Tatsache, dass dieser Server in JavaScript entwickelt wird, bietet für Entwickler die Möglichkeit Client- und Serverseite in einer Sprache zu implementieren und senkt damit den Entwicklungsaufwand.

HTTP-Server und REST-Paradigma müssen nicht vollständig selbst implementiert werden. Im Umfeld von node.js, hat sich ein umfangreiches Spektrum an Modulen, z. B. bereitgestellt über die Plattform „npm“, gebildet und mit

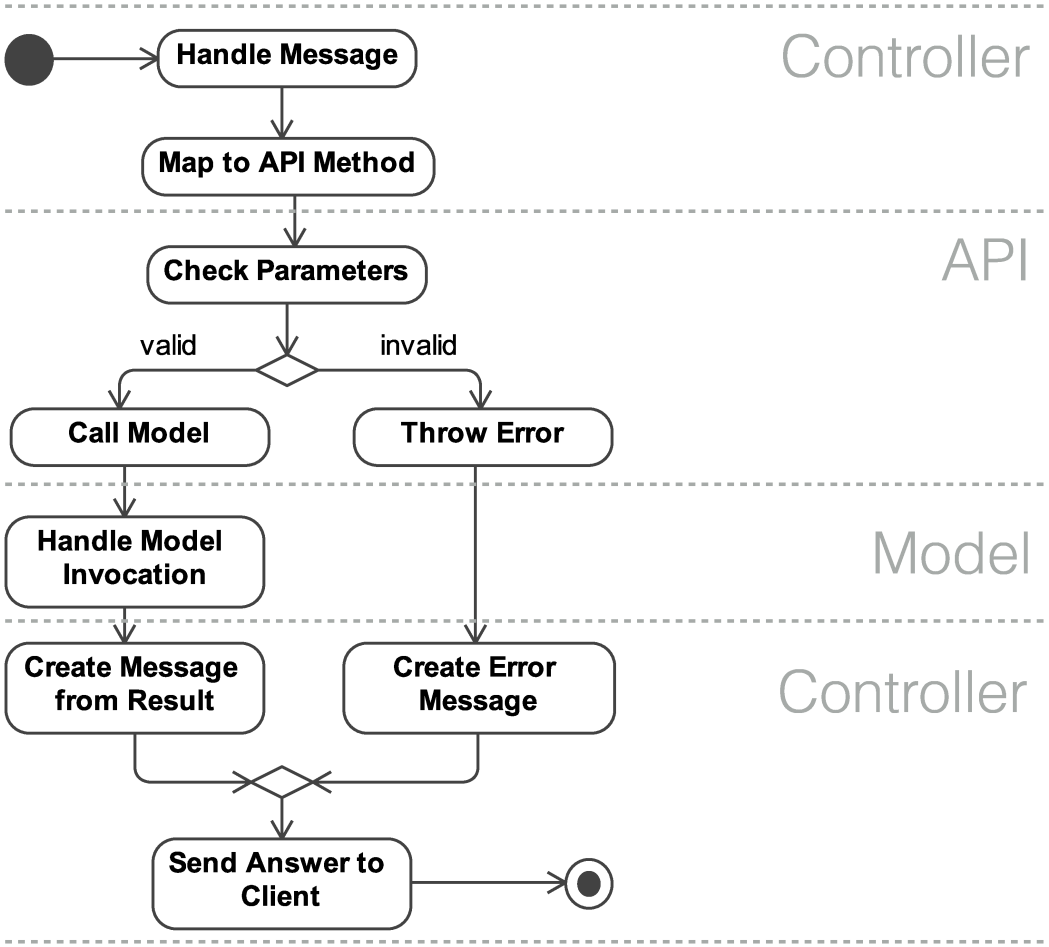



**Abbildung 3.13: Ablauf der Nachrichtenverarbeitung der Anwendung aus Anhang A (S. 231).**

dessen Hilfe sich REST Schnittstellen konstruieren lassen.

Das Beispiel aus Abbildung 3.12 aufgreifend, könnte eine Implementierung wie im Anhang A (S. 231) realisiert werden. Die Implementierung zeigt auf Basis von node.js und REST mithilfe des Moduls „express“ die fünf Schnittstellenmethoden auf Serverseite unter Berücksichtigung des gezeigten Architekturaufbaus für MMOGs. Die Anwendung teilt sich in Controller, Spielmodell und API für den Modellzugriff. Die Nachrichtenverarbeitung folgt dabei den in Abbildung 3.13 dargestellten Aktivitäten. Je Methode, z. B. im Fall der Bewegung des Spielers, wird die Prüfung der eingehenden Parameter, die entsprechenden Modellmanipulationen und das Senden einer positiven oder negativen Antwort durchgeführt. Vorteilhaft gegenüber dem TCP-Beispiel in Java ist die Nichtnotwendigkeit der Implementierung des Kommunikationspro-

tokolls und der Sitzungsverwaltung. Durch die Art der freien Definition von Attributen und Objekten ohne vorherige Deklaration ist zudem keine Definition von Request- und Response-Klassen notwendig. Diese entstehen automatisch durch die Anwendung der in JavaScript eingebetteten JSON-Verarbeitung.

### 3.2.5 Beispiel-Server mit JavaScript und WebSockets

Die Realisierung in Kapitel 3.2.4 (S. 51) zeigt in Anlehnung an die Architektur in Abbildung 2.2 eine Implementierung bei der der Controller, das Spielmodell und die API für den Modellzugriff als eigenständige Komponenten existieren. Der Ansatz in Anhang B (S. 237) ersetzt für ein WebSocket-Beispiel den Controller durch eine WebSocket-Implementierung. Die im Kapitel 3.2.4 (S. 51) genutzte API und Model Implementierung werden ohne Modifikationen übernommen.

Ein Unterschied zu reinen HTTP Anfragen und resultierenden Antwort, sowie REST ist, dass nach erfolgreicher Etablierung der Verbindung Daten ausgetauscht werden können. Während im Fall von REST die Identifikation der Anfrage über die URI realisiert wird, ist dies im Fall von WebSockets nach erfolgreichem Aufbau der Verbindung nicht mehr möglich. Daher muss das Datenpaket, welches ausgetauscht wird, eine zusätzliche Möglichkeit zur Identifikation bzw. Zuweisung auf eine Anfrage, enthalten. Aus diesem Grund sendet der Client im Beispiel ein Datenobjekt, das aus einem Kommando und dem eigentlichen Datenpaket besteht. Unabhängig davon bietet diese Implementierung dieselben Vorteile wie die JavaScript-REST Implementierung. Es sind keine zusätzlichen Kommunikationsobjekte notwendig und durch die Verwendung des WebSocket-Servers ist der TCP-Socket, die Sitzungsverwaltung und das Kommunikationsprotokoll bereits vorhanden.

## 3.3 Java Standard- und Enterprise Edition

Die beiden Versuche für eine Implementierung der Beispiele in Kapitel 3.2.4 (S. 51) und Kapitel 3.2.5 (S. 53) nutzten JavaScript und wurden mit node.js ausgeführt. Java bietet vergleichbare Werkzeuge und erlaubt einen feingranularen sowie standardisierteren Umgang mit Model und Controller. Um die Programmiersprache davon nicht zu überladen, wurde Java in zwei Versionen dif-

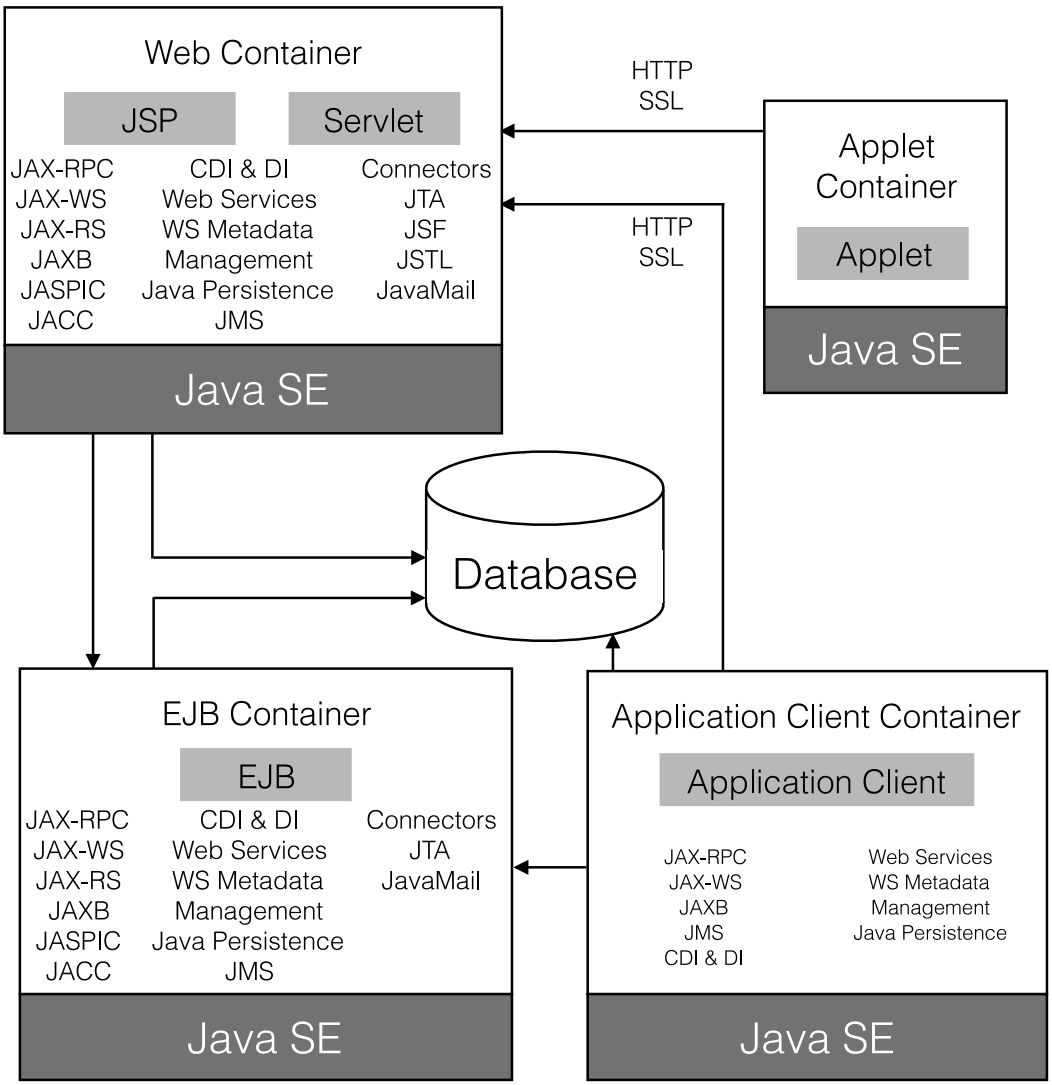



**Abbildung 3.14: Java EE Architektur Diagramm [71, S. 5].**

ferenziert: Java Standard und Enterprise Edition. Während die Java Standard Edition (SE) der Basisversion entspricht und den grundlegenden Sprachumfang liefert, fasst die Java EE zusätzliche Bibliotheken die für Entwicklung, Deployment sowie Verwaltung von multi-tier und serverzentrischen Anwendungen zusammen. Mit einem speziellen Fokus auf Webanwendungen baut die Java EE 7 auf Spezifikationen aus den Java Community Processs (JCPs) auf, die standardisierte Schnittstellen definieren. Diese Schnittstellen werden durch Werkzeug-Entwickler als Ausführungsumgebungen von Java EE realisiert und bereitgestellt. Implementierung auf Basis von Java EE sind damit vereinheitlicht und können theoretisch unabhängig in einer beliebigen der aktuell 16

verfügbaren Lösungen ausgeführt werden. [59]

Abbildung 3.14 liefert einen Überblick über die verschiedenen Bereiche der Java EE Spezifikationen. Diese unterteilt sich in Client- und Server-Bereiche und weist jedem Bereich anzuwendende Spezifikationen zu. Der Client kann entweder als einfaches Applet – eine spezielle Java Anwendungen geeignet zur Ausführung in einem Browser auf Basis eines Plug-ins[2] – mit dem Web Container über HTTP kommunizieren oder als eigenständige Java Application über verschiedene Protokolle mit allen beteiligten Serverkomponenten kommunizieren. Der Server teilt sich in einen Geschäftslogikbereich (Enterprise Java Bean (EJB) Container) und einen Präsentationsbereich (Web Container). [71, S. 5ff]

Eine Ausführungsumgebung muss nicht alle Spezifikationen umsetzen. Um jedoch das Zertifizierungssiegel für ein spezielles Profil zu erhalten, z. B. „Java EE Full Profile Certificated“, muss die in der Java EE Spezifikation verankerten Profil-Definition umgesetzt werden. [71, S. 200ff]

Im Folgenden ein kurzer Einblick in Spezifikationen, welche für Java EE Anwendungen häufig zum Einsatz kommen. Diese befassen sich mit der Modell-, Präsentations- und Persistenzschicht. Dies umfasst insbesondere die folgenden in [71] zusammengefassten Spezifikationen:

- **EJBs**
  Werden für die Realisierung des Zugriffs auf Geschäftslogik verwendet.

- **Java Servlets, Java Server Pages (JSP) und Java Server Faces (JSF)**
  Werden für Web-Frontends und technische Schnittstellen verwendet.

- **Java Naming and -Directory Interface (JNDI)**
  Wird für die Interaktion zwischen Komponenten verwendet.

- **Java Database Connectivity (JDBC) und Java Persistence API (JPA)**
  Wird für die Kommunikation zu Datenbanken und Synchronisation von Objektinstanzen mit der Datenbank verwendet, vgl. ORM.

[2] Vergleichbar mit Flash und Unity Browser-Plug-ins.

- **Java Management Extension (JMX)**
  Wird als Grundgerüst zur Realisierung von Services für die Ausführungsumgebung verwendet.

- **Java API for RESTful Web Services (JAX-RS) und Java API for XML Web Services (JAX-WS)**
  Wird zur Realisierung von REST- und SOAP-Schnittstellen genutzt.

- **Contexts and Dependency Injection (CDI)**
  Wird zum Bereitstellung von Abhängigkeiten in Komponenten verwendet.

- **Java Transaction API (JTA)**
  Wird für die Transaktionsverwaltung verwendet.

- **Java Authentication and Authorization Service (JAAS) und Java Authorization Contract for Containers (JACC)**
  Wird für Sicherheitsaspekte verwendet.

### 3.3.1 Java SE und EE Spezifikationen

Die **EJBs** dienen zur Umsetzung des Zugriffs auf Geschäftslogik. Die Beans sind nicht-visuelle Komponenten und werden in einem speziellen Container ausgeführt, der den Lebenszyklus vorgibt. Die aktuelle Spezifikation EJB 3 im Java Specification Request (JSR) 318 beschreibt die drei Grundtypen: Session [28, S. 23], Entity [28, S. 27] und Message Driven Bean [28, S. 33]. Beans sind Java Klassen die eine Annotation erhalten, die definiert, welcher Art diese EJB ist. Wird die Klasse in einem Applikationsserver veröffentlicht, erkennt dieser die Klasse und führt sie anhand des in der Spezifikation definierten Lebenszyklus aus, veröffentlicht diese in Verzeichnissen und stellt gewünschte Ressourcen bereit. EJBs bauen auf weiteren Java EE Spezifikationen auf, wie etwa der CDI-, JNDI- und JPA-Spezifikation. [27, S. 43ff]

Die **Java Servlets, JSPs und JSFs** werden im Web Container ausgeführt und stellen eine Möglichkeit zur Umsetzung der Präsentationsschicht dar. Die Anwendungen können als Schnittstelle (REST oder SOAP) bereitgestellt werden oder als visuelle Komponente, etwa als dynamische Webseite.

Die Spezifikationen sind in mehreren Dokumenten veröffentlicht, etwa dem JSR 315 für Servlets, JSR 245 für JSP und JSR 314 für JSF. Die Spezifikation der Servlets liefert die Grundlage zum Bereitstellen von Request-Response-basierten Schnittstellen über einen beliebigen Connector, z. B. mittels HTTP. Ein „Servlet“ ist eine Ableitung einer speziellen Servlet Klasse, z. B. *HttpServlet* für HTTP-basierte Servlets, und implementiert verschiedene Methoden zur Verarbeitung von Ereignissen, z. B. *doGet* und *doPost* für die HTTP-Methoden *GET* und *POST*. Die JSPs dagegen sind in ihrer Art mit der Sprache PHP vergleichbar. Es wird Java Code in einem HyperText Markup Language (HTML)-Dokument eingebettet und bietet somit dynamisch Elemente in einem statischen strukturierten Textdokument, z. B. HTML. Zur Laufzeit interpretiert der Applikationsserver dieses Dokument und führt es auf ein Servlet zurück – häufig mittels Codegenerator und einer Ableitung der *HttpServlet* Klasse. Die JSFs, als neuste Komponente für dynamische Webseiten, liefern ein Framework zur Entwicklung von grafischen Benutzeroberflächen. JSFs werden in XML-Dokumenten realisiert und arbeiten mit JavaBeans zusammen, die als Logikkomponente den dynamischen Zugriff auf Inhalte ermöglicht und in dem XML Dokument referenziert werden. [18, 55, 66]

Das **JNDI** ist ein Verzeichnisdienst welches den Zugriff auf benannte Ressourcen ermöglicht. Dieses kann entweder in derselben Ausführungsumgebung verfügbar sein oder mittels entfernten Zugriff in die Anwendung eingebunden werden. Applikationsserver bedienen sich für die EJBs dem Verzeichnis, um diese zu veröffentlichen. [71, S. 13]

Die beiden Spezifikationen **JDBC und JPA** stellen Werkzeuge zur Interaktion zwischen Anwendung und Datenbank bereit. Während JDBC einen abstrakten Zugriff auf eine Datenbank, z. B. mittels SQL, ermöglicht, ermöglicht die JPA-Spezifikation auf Basis der JDBC eine Abbildung von Objektinstanzen auf eine relationale Datenbank. JPA bedient sich hierfür den Annotationen die zur Definition von datenbankspezifischen Eigenschaften dienen, vgl. ORM (siehe Kapitel 2.8.2, S. 24). Zum Beispiel wird definiert, welche Entität in der Datenbank abzulegen ist, welches Attribut für die Identifikation zu verwenden ist und wie Relationen zwischen Objekten aufgelöst werden können. [71, S. 12-13]

Die **JMX** definiert ein Grundgerüst für Services innerhalb einer Java An-

wendung in Form von Managed Beans (MBeans), welche ebenso in einem Applikationserver Verwendung findet. Zum einen kann der Entwickler den Service implementieren und zum anderen der Serverbetreiber den Service ein oder mehrmals instanziieren und konfigurieren. Die Implementierung einer Service-Komponente mittels JMX benötigt eine Klasse sowie eine Schnittstelle, welche den Namen der Klasse zzgl. „MBean“ trägt. Die Schnittstelle definiert dabei alle öffentlichen Methoden, erlaubt die Definition von modifizierbaren Laufzeitparametern über Getter/Setter Paare und die Verwendung der Callback-Methoden *init*, *start*, *stop* und *destroy*, welche Rückschluss auf den Lebenszyklus der Service-Komponente geben. Instanziiert wird diese Komponente über einen Deskriptor, welche die Klasse verwendet, einen Namen zuordnet und notwendige Parameter setzt. Der Deskriptor wird in XML formuliert und kann über verschiedene Wege in die Java EE Ausführungsumgebung gelangen. Will die Service-Komponente Kenntnis über ihre Ausführungsumgebung erhalten, kann sie die spezielle Schnittstelle *MBeanRegistration* implementieren und damit Kenntnis über Benennung und Kontext erhalten. [60, S. 21ff]

Die **JAX-RS und JAX-WS** definieren Annotationen und Schnittstellen zur Umsetzung von REST- und SOAP-basierten Diensten. Sie bauen auf Servlets auf und bedienen sich zur Abbildung von Objekten in XML oder JSON der JAXB-Spezifikation. [61, S. 8] [71, S. 15-16]

Die Spezifikation JSR 299 für **CDI** wurde in Java EE 6 eingeführt und spezifiziert die Bereitstellung von Ressourcen innerhalb der Ausführungsumgebung in einer Komponente, etwa einer EJB. Mithilfe von Annotationen können Produzenten und Konsumenten für Ressourcen definiert werden. Produzenten sind jedoch nur dann notwendig, wenn Ressourcen verwendet werden, die nicht automatisch bereitgestellt werden. Ergänzend liefert die CDI-Spezifikation eine einheitliche Realisierung des Observer-Patterns [40, S. 326] innerhalb des Applikationsservers um den entkoppelten Austausch von Informationen zwischen Komponenten zu erlauben. [50, S. 1ff]

Das Management von Transaktionen wird in der Spezifikation JTA geregelt. Die JTA fungiert als Schnittstelle zu dem lokalen Manager, der die Elemente die in einer Transaktion involviert sind, wie die Applikation, der Ressourcen Manager (z. B. Datenbanken) und der Applikationsserver, verwaltet. [71, S. 11]

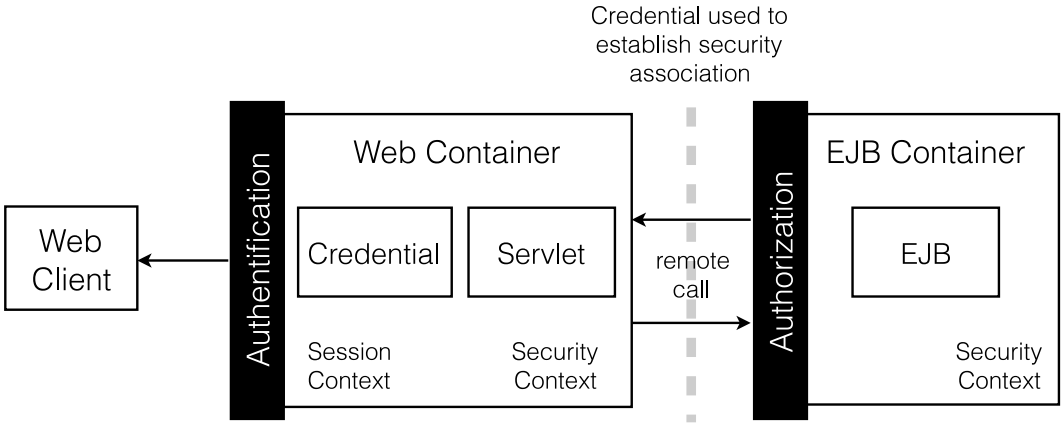



**Abbildung 3.15: Darstellung des Kommunikationswegs für Authentifizierung und Authorisierung. [71, S. 30]**

### 3.3.2 Sicherheit in Java SE und EE

Abschließend zwei Spezifikationen, die die Sicherheit und Transaktionen innerhalb eines Applikationsservers adressieren. Die Sicherheit kann nicht in einer einzelnen Spezifikation benannt werden, da Sicherheitsmechanismen bis auf die Java Virtual Machine (VM)-Ebene etabliert wurden. Die wichtigsten Vertreter in diesem Bereich sind die API JAAS und die für Java EE Umgebungen geltende JACC. Ziel dieser Mechanismen ist es, dass der Zugriff auf Methoden innerhalb der Geschäftslogik über unterschiedliche Kommunikationswege realisiert werden kann. Abbildung 3.15 zeigt, wie ein Web Client über den Web Container authentifiziert wird und innerhalb des Applikationsservers die Autorisierung beim Zugriff auf die Modellkomponente sichergestellt wird. [71, S. 27ff]

### 3.3.3 Beispiel-Server mit Java und REST

In Kapitel 3.2.2 (S. 46) wurde ein TCP-basierter Server in Java implementiert. Der folgende Ansatz realisiert das Spielmodell- und die API-Implementierung auf dieselbe Art und Weise. Der Controller wird mithilfe einer EJB realisiert, welche die eingehenden Kommunikationsobjekte prüft und an die API weiterreicht. Ergänzend kommt eine JAX-RS-basierte Schnittstelle zum Einsatz die die Funktionalitäten für den Client bereitstellt.

Im Fall der Verwendung von Java EE zur Realisierung ist die Implemen-

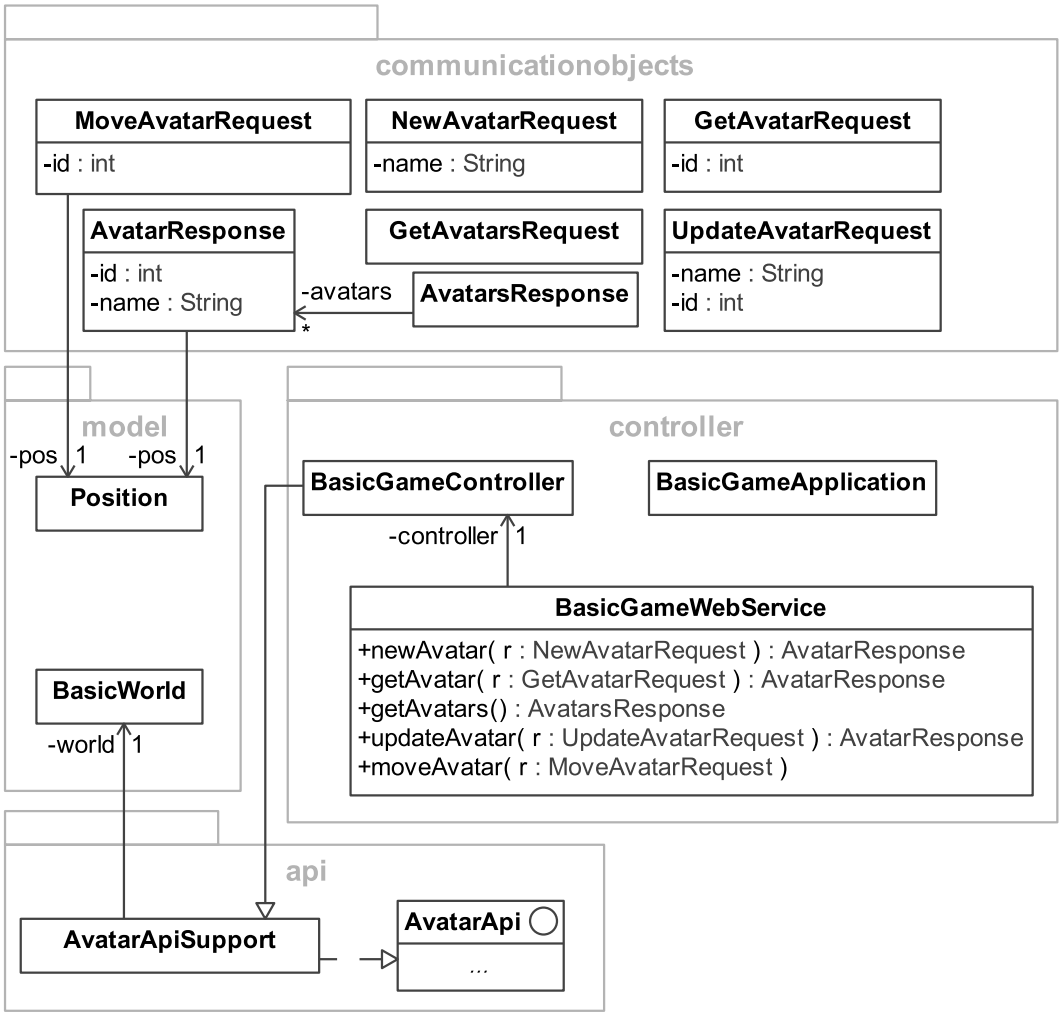



**Abbildung 3.16: Klassendiagramm für die Beispiel-Implementierung einer Java EE REST-Schnittstelle des minimalen Spielmodells aus Abbildung 2.2.**

tierung, wie in Abbildung 3.16 dargestellt, einer EJB *BasicGameController* und der JAX-RS *BasicGameWebService* notwendig. Durch die Spezifikationen in Java EE, bzw. der daraus folgend verwendeten Ausführungsumgebung, sind bereits TCP-Server, Sitzungsverwaltung, Kommunikationsprotokoll auf Basis von HTTP und die Verarbeitung von XML oder JSON basierten Datenpaketen vorhanden. Mit dem Eintreffen einer REST-Anfrage wird durch die Ausführungsumgebung die entsprechende Methode in einer Instanz der Klasse *BasicGameWebService* aufgerufen. Diese erhält bereits die fertige Instanz der erforderlichen Request-Klasse und reicht den Aufruf unter Verwendung deren Attribute an die korrekte API-Methode in die im System existierende Instanz

von *BasicGameController*, welche den *AvatarApiSupport* spezialisiert. Je nach gewählter Ausführungsumgebung kommt ergänzend die Administrationsfähigkeit der bereitgestellten Anwendungen hinzu.

## 3.4 Werkzeuge zur Spielentwicklung

Seit der Verfügbarkeit von bidirektionalen Kommunikationslösungen auf Plug-in-Basis im Browserbereich existieren Werkzeuge zur Unterstützung der Spielentwicklung. Diese wurden im Laufe der Zeit auf neue Technologien wie WebSockets in HTML5 erweitert, erlauben jedoch weiterhin die Realisierung von Spiele-Clients auf Basis von Browser-Plug-ins. Ein Einblick wird im Folgenden in die Middleware SmartFoxServer 2X (SmartFoxServer) [42], den Electroserver 5 [32] und den Photon Server [34] genommen. Ziel ist die Skizzierung der Vor- und Nachteile der betrachteten Lösungsansätze. Die Middleware SmartFoxServer wird darüber hinaus in Kapitel 3.4.5 (S. 74) zur Realisierung des Beispiels aus Kapitel 2.9 (S. 28) in Anlehnung an Kapitel 3.2.2 (S. 46) verwendet.

Alle drei Produkte stellen Lösungen für die Kommunikation zwischen Client und Server in einer verteilten Serverarchitektur dar. Die Lösungen behandeln die Verteilung von Nachrichten von einem Client zu einem anderen. Weniger im Fokus der Lösungen steht darüber hinaus die Simulation einer virtuellen Umgebung, Spielelemente und Ereignisverarbeitung, zur Vereinfachung werden jedoch in den einzelnen Lösungen Ansätze bereitgestellt.

Betrachtet werden im ersten Schritt der Aufbau und die Erweiterbarkeit dieser Werkzeuge. Dabei ist von Interesse, wie Spielmodelle zur Ausführung gebracht und Funktionalitäten für einen Client bereitgestellt werden. Ergänzend wurden die Nachrichtenpakete, die zwischen Werkzeug und Client ausgetauscht werden, in Anlehnung an die Betrachtung von XML und JSON, analysiert.

### 3.4.1 SmartFoxServer 2X

Der SmartFoxServer ist eine javabasierte Lösung des italienischen Unternehmens gotoAndPlay(). Die Serverlösung für browserbasierte Spielanwendungen,

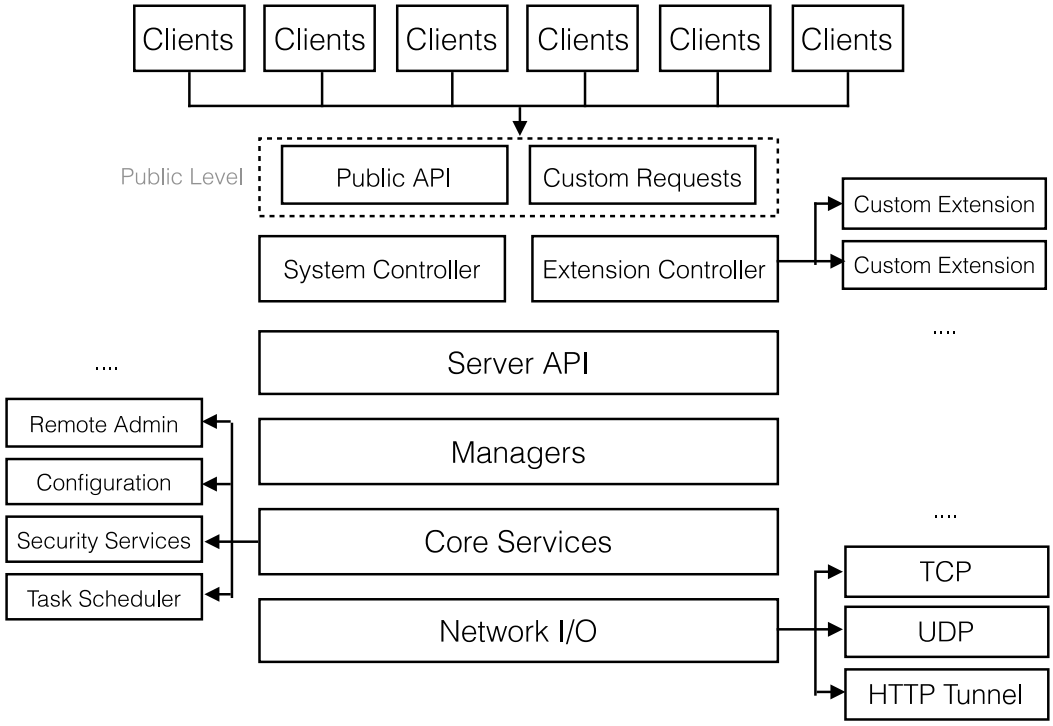



**Abbildung 3.17: Darstellung des Applicationserver-Stacks von SmartFoxServer [44].**

kann in der aktuellen Version mit gängigen Clienttechnologien, wie AS2/3, JavaScript, Java, C# und Objective-C interagieren. [44]

Aufgebaut, wie in Abbildung 3.17 dargestellt, ist der SmartFoxServer als eigenständiger Applikationsserver mit Lösungsansätzen zur Ausführung von Spielmodulen. Aufbauend auf dem Module *Network I/O* zur Kommunikation über TCP und UDP werden Dienste bereitgestellt, die den Informationsaustausch zwischen Spiel und Spieler realisieren. Eingebettet wird der individuelle Informationsaustausch mittels Erweiterungen in einem eigenen *Extension Controller*. Dieser realisiert Anfrageverarbeitungen und übernimmt die in Kapitel 2 (S. 7) skizzierten Aufgaben zur Kommunikationsinfrastruktur, Persistenz und Simulation des Spielmodells. Vergleichbar ist dieser Aufbau mit Java EE basierten Applikationsservern, wobei der SmartFoxServer einem monolithischen Ansatz folgt. [44]

Die Kommunikationsarchitektur ähnelt einer zonen- bzw. raumbasierten Struktur von Chatservern. Verbindet sich ein Nutzer mit dem Server, wird er einem Raum zugeordnet und erhält Nachrichten, die in diesem Raum an-

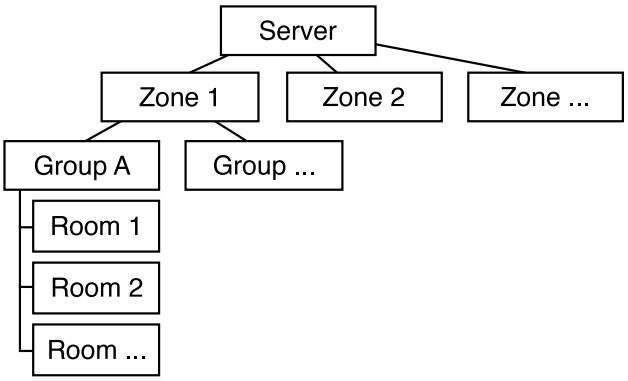



**Abbildung 3.18: Darstellung von Zonen, Gruppen und Räumen in einem SmartFoxServer [43].**

fallen, jedoch nicht die, die bereits in der Vergangenheit propagiert wurden. Räume sind zudem gruppiert und die Nutzer können Nachrichten die dieser Gruppe zugeordnet sind erhalten. Abbildung 3.18 zeigt den Aufbau grafisch. Diese Chatserver orientierte Architektur bietet eine solide Basis für gefilterte Kommunikation: Nur wer in demselben Raum ist, erhält Aktualisierungen. Raumgrenzen sind in einem Spiel z. B. auf Gebiete, Städte oder Spielbretter (z. B. das Brett eines Schachspiels) abbildbar. Grenzen sorgen zwischen den Räumen für harte Übergänge, welche in einem Spiel meist mittels Zonengrenzen (z. B. durchquert man ein Portal um in die nächste Zone zu gelangen) dargestellt werden. [43]

Die Architektur erlaubt es Spielmodelle, die ohne Persistenz auskommen, ohne serverseitige Anwendungslogik umzusetzen. In einem solchen Szenario genügt es einen Client zu konstruieren. Dieser verbindet sich mit dem Server und verteilt Änderungen am Spielmodell über die Räume an andere Spielteilnehmer.

Um Funktionalitäten auf die Seite des Servers zu verlagern, lassen sich Erweiterungen über die bereitgestellte API in Java und Python umsetzen. Dies können Algorithmen, Prozesse oder ganze Spielmodelle sein, die ohne Anwesenheit des Spielers den Zustand des Spiels verändern könnten. Wichtiges Werkzeug bei der Realisierung der Erweiterungen ist die Verarbeitung von Ereignissen und Anfragen.

Das im SmartFoxServer enthaltene Kommunikationsprotokoll und Abbildungsformat erlaubt das Versenden von Objekten mithilfe einer Datenstruktur

die Key-Value-Pairs verwaltet. Das Abbildungsformat dabei eine Grundmenge an Basisdatentypen. Dazu zählen verschiedene Ganzzahlen und Fließkommazahlen sowie Zeichenketten. Erlaubt sind ebenso Unterobjekte und Reihungen der genannten Datentypen. Die Nutzdaten werden mit Datentyp und Attributnamen abgebildet und können komprimiert werden. Abbildung 3.2 zeigt ein Beispielobjekt. In Anhang G (S. 247) wird das Beispielobjekt als Datenblock in Hexadezimaldarstellung abgebildet. Markiert sind in dieser Abbildung die variablen Daten, nicht markierte Bereiche enthalten Strukturdaten wie Attributnamen und Typinformationen. [45]

Wertet man das Datenvolumen des Beispielobjekts aus Abbildung 3.2 aus, so erhält man eine Gesamtgröße von 351 Byte, wobei in diesem 60 % Strukturdaten enthalten sind, die bei jeder Datenübertragung mitgeführt werden, sich jedoch nie ändern (vgl. Abbildung 3.19). Die prozentuale Menge an Strukturdaten ist nicht repräsentativ, da je nach Menge an Instanzen, Komplexität der Attributnamen und Struktur der Klassen dieser Wert schwankt. Im gezeigten Beispiel ist der Wert niedriger ausgefallen, da einige der übertragenen Daten vom Datentyp String sind, die mehr Volumen im Datenstrom einnehmen als zum Beispiel eine Ganzzahl, insofern deren Länge größer als 2 Byte ist. Dies ist in der Tatsache geschuldet, dass als Präfix für jede Zeichenkette eine Längenangabe notwendig ist, die 2 Byte benötigt. Für eine Zeichenkette der Länge 1 sind damit 3 Byte nötig.

Für die Kommunikation zu Webanwendungen steht zusätzlich eine JSON basierte Repräsentation von Datenobjekten zur Verfügung. Wird diese Repräsentation gewählt, wird das Volumen des übertragenen Datenstroms umfangreicher.

Der Analyse der übertragenen Datenströme folgt die Betrachtung der Anwendung dieser innerhalb des SmartFoxServers. Wenn in einer beliebigen Spielanwendung serverseitige Ausführungslogik im SmartFoxServer notwendig ist, kann diese Mithilfe von sogenannten Extensions realisiert werden. Innerhalb dieser Extensions können Daten verarbeitet und ggf. persistent hinterlegt werden. Quellcode 3.5 zeigt einen Ausschnitt zum Versenden von Daten aus einem SmartFoxServer. Neben der Generierung des Kommunikationsobjektes (Zeile 1) werden Attribute in datentypspezifischen Aufrufen (z. B. *putInt* für eine 32 Bit Ganzzahl) dem Objekt hinzugefügt (Zeile 2 bis 4). Anschließend

```
1 ISFSObject respObj = new SFSObject();
2 respObj.putInt("x", moveX);
3 respObj.putInt("y", moveY);
4 respObj.putInt("t", user.getPlayerId());
5 send(CMD_MOVE, respObj,
6   gameExt.getGameRoom().getUserList());
```

**Quellcode 3.5: Versenden einer Antwort. Parameter werden in einem Kapselobjekt mit Namen und Datentyp hinzugefügt.**

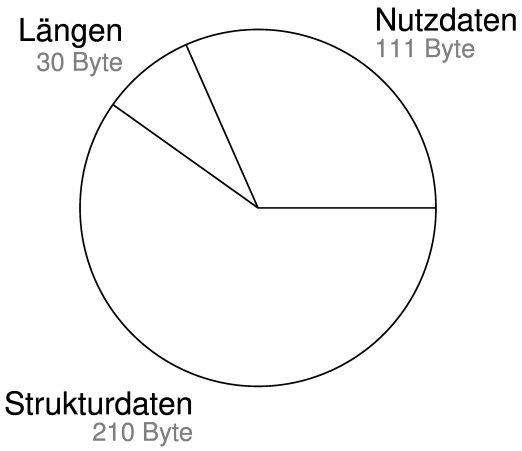



**Abbildung 3.19: Prozentuale Verteilung der Nutz- und Strukturdaten bei Verwendung des Abbildungsformates des SmartFoxServer.**

kann das Objekt an eine spezifische Menge von Nutzer gesendet werden.

Um ein Datenpaket zu empfangen bedarf es der Implementierung einer verarbeitenden Methode. Diese *handleClientRequest*-Methode bekommt bei Aufruf einen Verweis auf den aufrufenden Nutzer sowie das übertragene Datenpaket. Quellcode 3.6 zeigt die Methodensignatur (Zeile 2) und das Auslesen zweier Attribute $x$ und $y$ (Zeile 4 und 5).

Quellcode 3.5 und Quellcode 3.6 sollen zeigen, wie in der durch den SmartFoxServer umgesetzten Kommunikationsarchitektur eine Infrastruktur für den Versand und Empfang von Nachrichten bereitgestellt wird. Die Ausführungsumgebung präsentiert sich dabei als Hilfsmittel zur Übermittlung von Daten zwischen zwei voneinander getrennten Umgebungen. Für eine modellgetriebene Spielentwicklung muss der Anwendungsentwickler sich auf diese Umgebung einlassen. In der Verarbeitungslogik (siehe Quellcode 3.6) der

```
@Override
public void handleClientRequest(User user,
  ISFSObject params) {
  // ...
  int moveX = params.getInt("x");
  int moveY = params.getInt("y");
  // ...
```

**Quellcode 3.6: Behandeln einer Anfrage. Parameter werden in einem Kapselobjekt transportiert die einen Zugriff mittels Attributnamen ermöglichen.**

demonstrierten *handleClientRequest*-Methode musst das Datenpaket mithilfe des übergebenen Parameters vom Typ *ISFSObject* interpretiert werden und die daraus folgenden Anweisungen auf das Spielmodell dirigiert werden. Für das Ergebnis wird eine Instanz vom Typ *SFSObject* erzeugt und an den Client mit einem Aufruf von *send* übermittelt (siehe Quellcode 3.5). Es entsteht ein Controller (vgl. Model-View-Controller (MVC) Pattern [19, S. 125]) der zwischen Spielmodell und SmartFoxServer API vermittelt. Innerhalb dieser Extension kann das Spielmodell simuliert werden und die Persistenz des Zustandes realisiert werden.

### 3.4.2 ES5 Electroserver

Eine weitere Lösung aus dem Bereich der browserbasierten MMOG-Middleware ist der javabasierte Electroserver 5 (ES5) des amerikanischen Unternehmens Electrotank. Der ES5 unterstützt eine breite Menge an Clienttechnologien, von AS2/3, über C#, Java, Objective-C bis hin zu JavaScript. Asynchrone Kommunikationen ist jedoch nur bei AS3, C#, Java und Objective-C über dauerhaft aktive Verbindungen möglich. [31]

Basierend auf einem monolithischen Kernel kann, ähnlich wie bei dem SmartFoxServer in Kapitel 3.4.1 (S. 61), eigene Logik ausgeführt werden, die auf verschiedene Kommunikationsanfragen reagiert und Informationen an Clients verteilt. Eine spezielle Erweiterung stellt Electrotank mit dem Electrotank Universe Platform (EUP) bereit. Das EUP ist eine Spielinfrastruktur, die aufbauend auf dem ES5 Events, Transaktionen und Entitäten verwalten und persistieren kann. Abbildung 3.20 zeigt die Grundstruktur des

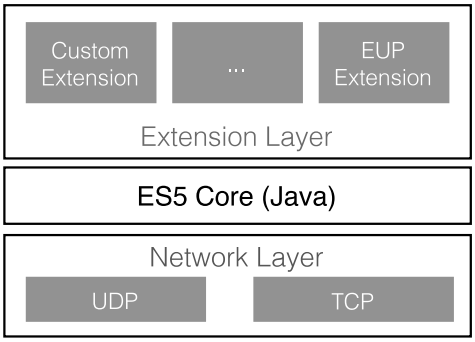



**Abbildung 3.20: Struktur des ES5 Electroservers.**

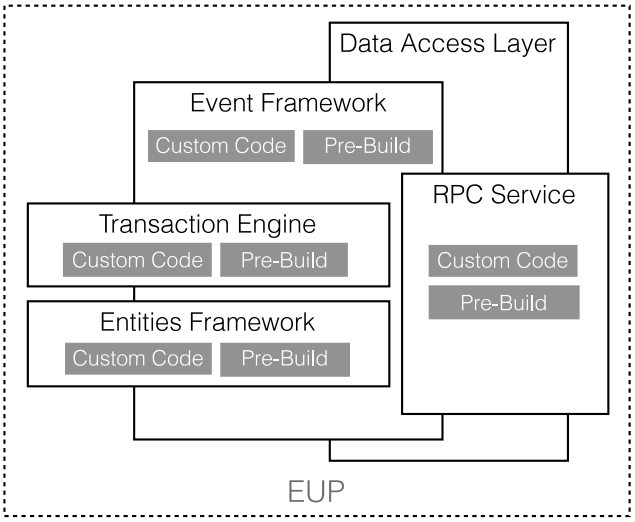



**Abbildung 3.21: Struktur des ES5 zzgl. Erweiterungen [30].**

ES5 und wo in diesem Erweiterungen platziert werden können. Ebenso zeigt Abbildung 3.20 wie sich EUP in dem ES5 integriert. Abbildung 3.21 liefert ergänzend einen Einblick in die Erweiterung EUP. [31]

Die Kommunikation im ES5 findet, wie im SmartFoxServer, über eine Chatraum-Struktur statt. Spieler können einem oder mehreren Räumen zugeordnet werden und über diese Räume propagierte Nachrichten empfangen. Im Gegenzug zur Konkurrenz verzichtet diese Middleware auf eine weitere Gruppierung der letzten Ebene. Abbildung 3.22 zeigt den Aufbau grafisch. [31]

Der ES5 verwendet eine Datenstruktur mit Key-Value-Pairs zur Bereitstellung von Daten, die zu übertragen sind oder empfangen wurden. Diese *EsObjectRO*-Objekte enthalten alle Parameter, die mit einer Anfrage übertragen

```
1 @Override
2 public void request(String playerName, EsObjectRO
    requestParameters) {
3   EsObject messageIn = new EsObject();
4   messageIn.addAll(requestParameters);
5   String action = messageIn.getString(PluginConstants.ACTION
    , "");
6   if (action.equals(PluginConstants.USER_LIST_REQUEST)) {
7     handleUserListRequest(playerName);
8   } else if (action.equals(PluginConstants.BROADCAST_REQUEST
    )) {
9     handleBroadcastRequest(playerName, messageIn);
10  // ...
11  } else if (action.isEmpty()) {
12    sendErrorMessage(playerName, PluginConstants.
    MISSING_ACTION);
13  } else {
14    sendErrorMessage(playerName, PluginConstants.
    INVALID_ACTION);
15  }
16 }
```

**Quellcode 3.7: Codeausschnitt aus den Beispielen des ES5. Zu sehen ist, wie in Zeile 3 das eingehende Datenpaket in ein Datenobjekt eingelesen wird und anschließend die Zeichenkette mit der ID PluginConstants.ACTION ausgelesen wird. In den folgenden Zeilen folgt eine Bedienungsweiche um das Anfrageobjekt entsprechend seiner Typisierung zu verarbeiten.**

wurden und können mittels Verwendung von Bezeichnern ausgelesen werden. Unterstützt werden die in Java verfügbaren Basisdatentypen sowie Zeichenketten und eindimensionale Reihungen der genannten Typen. Ergänzend kann ein *EsObject* wieder ein oder mehrere Daten des Types *EsObject* enthalten, sowie Reihungen dieser enthalten. Abbildung 3.7 zeigt einen Codeausschnitt für eine Anfrageweiche. Zeile 3 bis 4 erzeugt ein *EsObject* aus den Eingabedaten, an denen letztlich mittels typisierten Methodenzugriff (z. B. `getInteger(name)`) der entsprechende Anfrageparameter ausgelesen werden kann. Im Folgenden muss der Entwickler die Anfrage behandeln - hierfür wird im Beispiel eine If-Cascade verwendet. Abschließend folgt die Konstruktion der Antwort wie im Quellcode 3.8 dargestellt und wird an ein oder mehrere Empfänger zurückgesendet.

Wie der SmartFoxServer verwendet der ES5 ein proprietäres Abbildungsformat zu Serialisierung der Datenpakete. Der ES5 unterstützt die Spielent-

```
1 EsObject messageOut = new EsObject();
2 messageOut.setString(
3   PluginConstants.ACTION,
4   PluginConstants.BROADCAST_UDP_EVENT);
5 messageOut.setString(PluginConstants.USER_NAME, playerName);
6 getApi().sendPluginMessageToRoom(
7   getApi().getZoneId(),
8   getApi().getRoomId(),
9   messageOut
10 );
```

**Quellcode 3.8: Codeausschnitt aus den Beispielen des ES5. Es ist zu sehen wie in Zeile 1 ein Datenobjekt entsteht und in Zeile 2-4 und 5 jeweils zwei Zeichenketten in diesem Objekt gesetzt werden. In Zeile 6-10 wird dieses Objekt dann an den aktuellen Nutzer gesendet.**

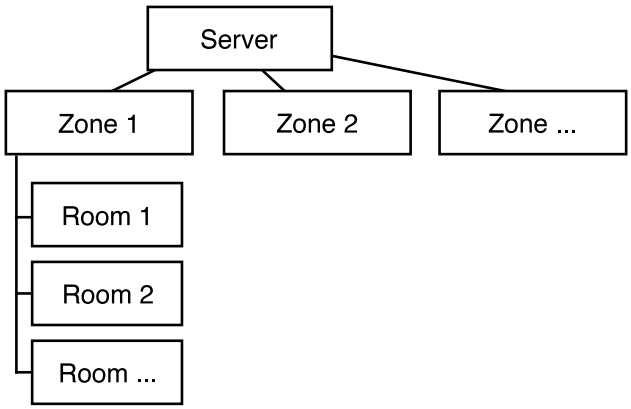



**Abbildung 3.22: Raumstruktur zur Kommunikation innerhalb des ES5 Electroservers [31].**

wicklung durch eine stabile Infrastruktur zur Kommunikation zwischen zwei Plattformen und ermöglicht die Interaktion mit einer hohen Menge (>1000) von Clients. Für ein Spielmodell, das ausschließlich spielrelevante Implementierung enthält, muss verarbeitende Logik implementiert werden, die die eingehenden Nachrichten (in der *request*-Methode) interpretiert und auf dem Modell anwendet. Diese Methode wird im Sinne eines MVC [19, S. 125] als Controller fungieren und übernimmt das Dirigieren von Aufgaben. Wie bei dem SmartFoxServer muss das Spielmodell innerhalb der Erweiterung verwaltet und simuliert werden. Die Persistenz kann über ES5 Schnittstellen realisiert werden, muss jedoch vom Entwickler eigenständig angesprochen werden. [31]

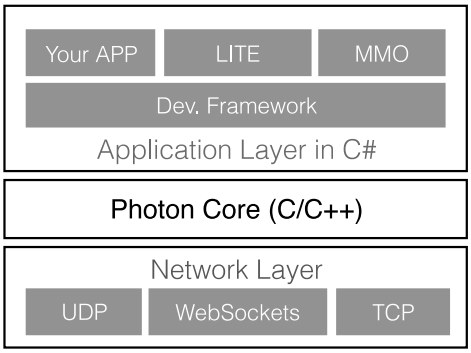



**Abbildung 3.23: Darstellung der Architektur der Photon Serverplattform [35].**

### 3.4.3 Photon Server

Eine weitere kommerzielle Serverplattform, die nicht Java als Ausführungsumgebung verwendet, ist der Photon Server des deutschen Unternehmens Exit Games. Die für C# verfügbare Lösung basiert auf einem in C/C++ geschriebenen monolithischen Kernel und führt, ähnlich wie der SmartFoxServer und der ES5, Spiele als Applikation, bzw. Erweiterung aus. Der Photon Server bietet eine einfache Möglichkeit diese als Webanwendung bereitzustellen, sowie mehrere Kommunikationsschnittstellen für dauerhaft aktive Verbindungen wie UDP (Reliable[3]), TCP und die seit HTML5 verfügbaren WebSockets. Exit Games bietet für den Photon Server APIs in allen gängigen Clientplattformen, um eine einfache Anbindung an die Serverlandschaft zu garantieren. Dazu zählen HTML5 mittels JavaScript, Flash mittels AS3, iOS und Mac mittels C-Objective, Unity mittels C#, sowie für Android mittels Java API. [35]

Die Sprachunterschiede ignorierend, arbeitet der Photon Server vergleichbar wie die beiden Konkurrenten. Darauf aufbauend bietet die Lösung die für Spiele notwendigen Basiskonzepte wie Prozess-, Event-, Encryption- und Data-Handling. Abbildung 3.23 stellt die Architektur des Photon Servers grafisch dar. [35]

Im Vergleich zu den javabasierten Lösungen SmartFoxServer und ES5 werden abstrakte Spielmodelle bereitgestellt, die das Umsetzen von speziellen Spieltypen (wie Role Playing Game (RPG) oder Strategie) vereinfachen sollen.

[3]Erlaubt einen garantierten Datentransport durch Erweiterung des UDP Protokolls um z. B. Acknowledge Packages und Neuversand verlorener Datengramme.

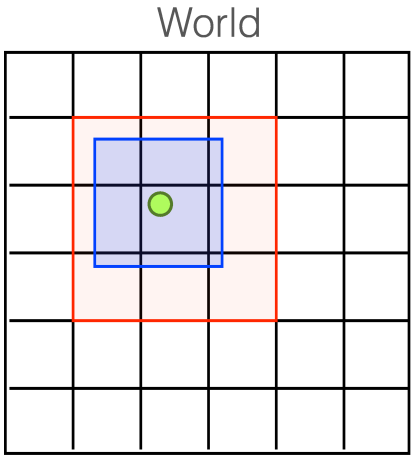



**Abbildung 3.24: Darstellung einer 2D Abbildung von Interessengebieten (blau) von einem Objekt (grün) und den daraus resultierenden Suchbereichen (rot) [35].**

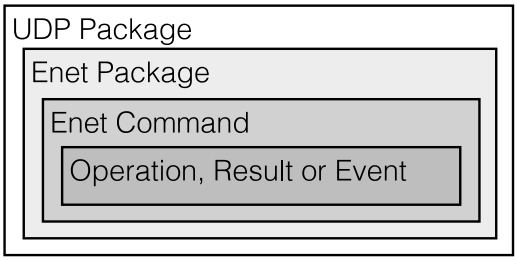



**Abbildung 3.25: Darstellung der Paketstrukur der Photon Servers Kommunikation [35].**

Diese sogenannten „Power Ups“ stehen in Form von Schnittstellendefinitionen und abstrakten Klassen in einer API bereit und können implementiert und erweitert werden. [35]

Power Ups stellen im Photon Server ein herausstechendes Merkmal zu den im ES5 und SmartFoxServer verwendeten starren Raum Modell dar. Während die anderen beiden Lösungen die Umsetzung von Erweiterungen ermöglichen, die direkt mit den Räumen arbeiten, versucht der Photon Server diese technische Eigenschaft in einem abstrahierten spielähnlichen Model zu kapseln. Im Detail enthält das „First Person Shooter Power Up“ das bekannte Raummodell und spezialisiert dieses im „MMO Power Up“ zu einem Raster aus Planquadraten, in denen Objekte des Spiels positioniert werden. Hierfür werden positionierbare Objekte in fixe Regionen eingruppiert und bei Bedarf zwischen diesen migriert (z. B. im Falle einer Bewegung). Basierend auf dieser Struktur

| *Datenblock* | *Beschreibung* |
|---|---|
| F100000016025A00 | Op.Header |
| 0001 | Anzahl Parameter 1 |
| 04 | Parameter Identifikator |
| 73 | Datentyp (String) |
| 0008 | 8 Zeichen |
| 736F6D6567616D65 | „somegame“ |

**Tabelle 3.1: Aufschlüsselung eines Commandoblocks des Photon Servers [35].**

```
00  F1 00 00 00 16 02 5A 00 00 01 04 73 00 08 73 6F   ......Z....s..so
10  6D 65 67 61 6D 65                                 megame
```

**Abbildung 3.26: Darstelllung eines kompletten Commandoblocks für einen Raumbeitrittbefehls, die Aufschlüsselung der Daten ist in Quellcode 3.1 [35].**

lassen sich effizienter Suchanfragen gestalten, um Objekte ausfindig zu machen, die bei einer Aktualisierung informiert werden müssen. Abbildung 3.24 zeigt ein Objekt (grün) welches dem umliegenden Planquadrat in Abhängigkeit von seiner Positionierung zugeordnet werden kann. Abhängig von seinem Interessengebiet (blau) gibt es eine Gesamtanzahl von Planquadraten (rot), die für dieses Objekt von Interesse sein könnten und durchsucht werden können. Aus Softwaresicht lässt sich dieses Konstrukt als wohlindizierte Sammlung von Räumen auffassen, in denen Empfänger gelistet werden. Änderungen werden in diesem Fall nicht nur in den aktuellen Raum, sondern an alle Räume im Interessengebiet weitervermittelt. Entscheidend für die Ausmaße des Suchbereichs ist dabei die maximal mögliche Interessensweite des beobachtenden Objektes. [35]

Nachrichten werden im Photon Server in einer Datenstruktur aus Key-Value-Pairs für die Übertragung bereitgestellt. Dabei werden diese Paare in einer Operation gesammelt, welche wiederum in einem Kommandoblock zusammengefasst werden. Der Photon Server versucht auf diese Art und Weise den Daten-Overhead, der durch UDP, TCP und WebSocket Paketheader entsteht minimal zuhalten und versendet hierfür Pakete zumeist in optimalen Paketgruppen. Abbildung 3.25 zeigt den hierarchischen Aufbau der Paketelemente im Photon Server. Ergänzend zeigt Quellcode 3.1 eine Aufschlüsselung

```
1 // Read some parameter from data container
2 int myActorNr = (int)operationResponse.Parameters[LiteOpKey.
    ActorNr];
3 Console.WriteLine(" ->My PlayerNr (or ActorNr) is:" +
    myActorNr);
```

**Quellcode 3.9: Codeausschnitt zum Verarbeiten von Nachrichten. Der Zugriff auf die Daten des Pakets erfolgt mittels Index, vergleichbar dem Zugriff auf Elemente einer Kette.**

```
1 // Create new data container
2 Dictionary<byte, object> opParams = new Dictionary<byte,
    object>();
3 opParams[LiteOpKey.Code] = (byte)101;
4 opParams[LiteOpKey.Data] = "Hello World!";
5 // Submit data container
6 peer.OpCustom((byte)LiteOpCode.RaiseEvent, opParams, true);
```

**Quellcode 3.10: Codeausschnitt zum versenden einer Nachricht. In Zeile 2-4 entsteht das Datenpaket in Form eines Key-Value-Pairs. In Zeile 6 wird dieser versendet.**

des Datenblocks in Abbildung 3.26. Im Gegensatz zum SmartFoxServer steht für den Photon Server nur wenig detailliertes Material zur genauen Protokollstruktur bereit. Betrachtbar ist der Datenblock, Bezug nehmend auf Abbildung 3.25, für die Operationen. Verglichen mit dem SmartFoxServer wird bei diesem Server ein geringerer Anteil an Strukturdaten transportiert, da für die Identifikation von Parametern nur 8 Bit Ganzzahlen gestattet sind. Dies ist hilfreich bei der Reduktion des Datenvolumens, jedoch umständlicher bei der Anwendung für den Programmierer, da jeweils der nicht intuitiv erkennbare Wert zu verwenden ist.

Quellcode 3.9 und Quellcode 3.10 zeigen kurze Ausschnitte, wie die Daten ein- und ausgehend Verarbeitet werden. Die Behandlung von eingehenden Nachrichten erfolgt wie bei den Vergleichslösungen in einer Handler Methode. Ausgehende Nachrichten können über die beschriebenen Strukturen an zuvor gruppierte Nutzer versendet werden. Die Daten werden mithilfe eines Abbildungsformates in einen Datenstrom überführt und übermittelt. Zur Abbildung wird eine kompakte Repräsentation verwendet, bei der Key-Value-Pairs hintereinander abgebildet werden, wobei jedes Kommando mit einem Befehlsin-

dikator startet.

### 3.4.4 Überblick

Bei einem Vergleich der verschiedenen Lösungen bleiben wenig unterschiedliche Merkmale über. Die Fähigkeiten zum Übertragen von Nachrichten und die Erweiterbarkeit, sind in jeder Middleware identisch. Lediglich die Anzahl und Ausprägung der unterstützten Clientplattformen variiert. Alle Ansätze sind proprietär und realisieren einen eigenen monolithischen Kernel, welcher spielbezogene Erweiterungen erlaubt. Quellcode 3.2 zeigt einen Überblick über einzelne Unterscheidungsmerkmale.

Alle Lösungen erlauben die Kommunikation über TCP oder UDP sowie spezielle Protokolle wie das Flash Policy Handling, WebSockets und HTTP an. Ebenso unterstützen alle die am stärksten verbreiteten Clienttechnologien wie HTML5 (JavaScript), Flash (ActionScript) und Unity (C#) und bieten entsprechende APIs an. Ergänzend werden administrative Tools und Verbindungsgruppierungen (Chatraumfilter oder nach Positionen) angeboten.

In allen Ansätzen wird eine Infrastruktur geschaffen, jedoch ist der Entwickler weiterhin verantwortlich Datenpakete zu konstruieren, die er an den Client übermitteln lässt und dort in das vorliegende Modell einpflegt. Fraglich ist, warum trotz Entwicklung in objektorientierten Sprachen die Abbildung von Daten über Strukturen auf Key-Value-Pair-Basis gelöst wird.

### 3.4.5 Beispiel-Server mit SmartFoxServer 2X

Wie im vorangegangenen Kapitel 3.2.2 (S. 46), Kapitel 3.3.3 (S. 59), Kapitel 3.2.4 (S. 51) und Kapitel 3.2.5 (S. 53) wird im Folgenden das Beispiel aus Kapitel 2.9 (S. 28) verwendet und mit dem Werkzeug SmartFoxServer realisiert. Verwendet werden dafür die sogenannten Extensions, welche in Java und Python bereitgestellt werden können. Die Realisierung des Spielmodells ist identisch mit der aus Kapitel 3.2.2 (S. 46). Die für dieses Beispiel notwendigen Klassen sind in Abbildung 3.27 in einem Klassendiagramm zusammengefasst dargestellt.

Der SmartFoxServer verwendet, wie das Beispiel in Kapitel 3.3.3 (S. 59), Kommunikationsobjekte, welche nach Empfangen der Key-Value-Pairs intern

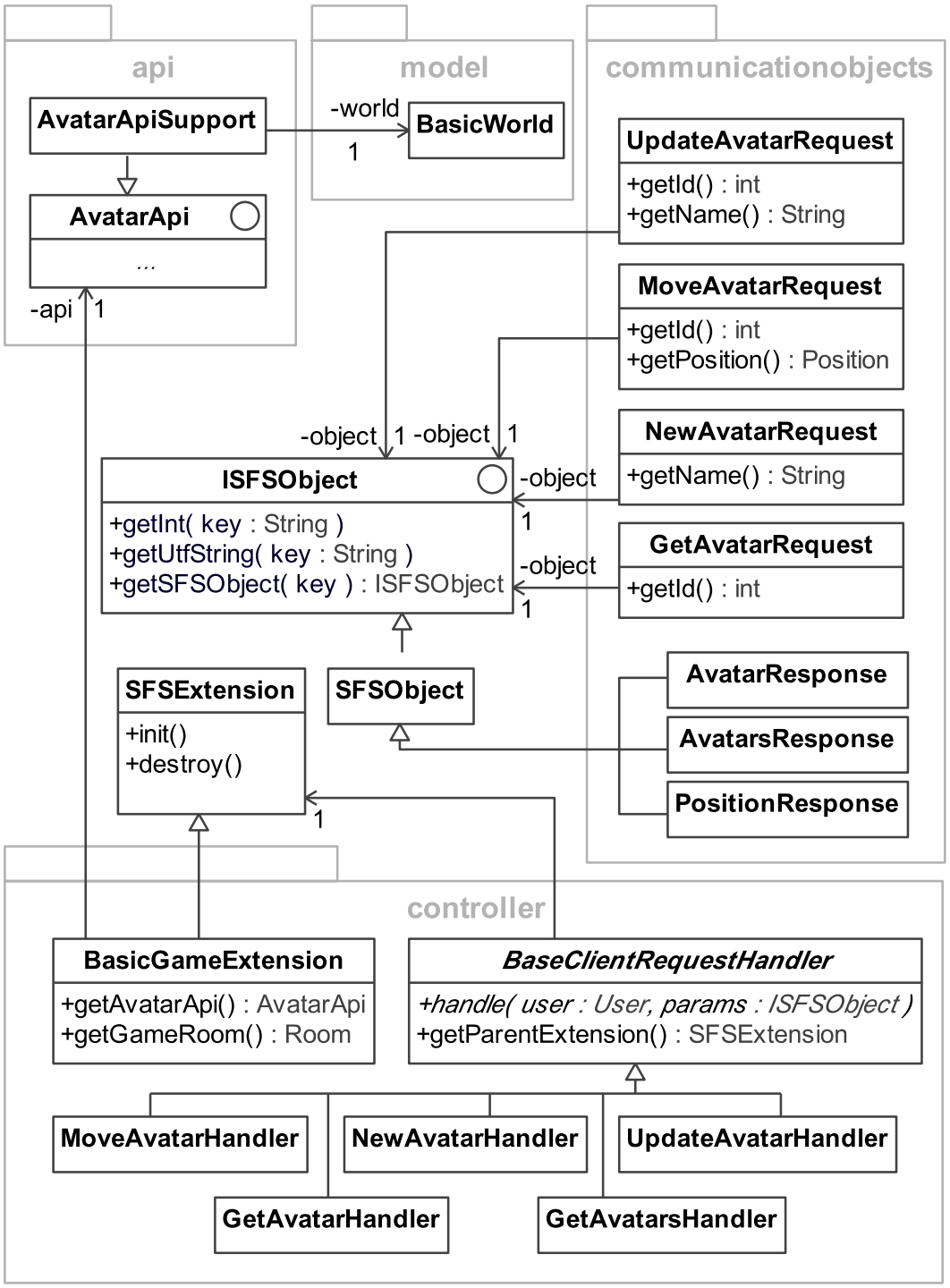



**Abbildung 3.27: Klassendiagramm für die Beispiel-Implementierung des minimalen Spielmodells aus Abbildung 2.2 im SmartFoxServer.**

| | *SmartFoxServer* | *ES5* | *Photon Server* |
|---|---|---|---|
| Plattform | Java | Java | C# |
| Betriebssystem | Windows, Mac, Linux, Unix | | Windows |
| | *Schnittstellen* | | |
| Binary TCP | ✓ | ✓ | ✓ |
| Binary UDP | ✓ | ✓ | ✓(Reliable) |
| Flash Policy | ✓ | ✓ | ✓ |
| WebSockets | ✓ | ✓ | ✓ |
| HTTP | ✓ | ✓ | ✓ |
| | *Client API* | | |
| ActionScript 2/3 | ✓ | ✓ | ✓ |
| JavaScript | (✓)nur JSON | (✓)nur HTTP | ✓ |
| Objective C | ✓ | ✓ | ✓ |
| C# | ✓ | ✓ | ✓ |
| Java | ✓ | ✓ | ✓ |
| | *Sonstiges* | | |
| Nutzerverwaltung | ✓ | ✓ | ✗ |
| Raumverwaltung | ✓ | ✓ | ✓ |
| Package Encryption | ✗ | ✓ | ✓ |

**Tabelle 3.2: Vergleich von Kommunikation, Client und sonstigen Merkmalen.**

zur weiteren Verarbeitung genutzt werden. Zur vereinfachten Behandlung, erhalten alle Request-Objekte einen Verweis auf die empfangene *ISFSObject*-Instanz und jeder Getter greift auf das passende Attribut in der Datenstruktur zu. Response-Objekte spezialisieren die Klasse *SFSObject*, welche eine Realisierung der *ISFSObject*-Schnittstelle ist, und haben in diesem Ansatz einen Konstruktor der die Datenkapsel basierend auf den übergebenen Parametern befüllt.

Für die Implementierung wird eine Spezialisierung der SmartFoxServer-Klasse SFSExtension benötigt. Diese wird von der Ausführungsumgebung instanziiert und definiert die zu verarbeitenden Anfragen in einem speziellen SmartFoxServer-Kommunikationsraum (vgl. raumbasierte Kommunikation in Kapitel 3.4.1 (S. 61)). Jede Anfrage ist in diesem Fall eine Spezialisierung von *BaseClientRequestHandler*. Mit der zu implementierenden Methode *handle* wird beschrieben, was im Fall einer Anfrage passieren soll. Im Fall dieser Beispiel-Implementierungen wird die empfangene Datenkapsel in das entsprechende Request-Objekt überführt und die API mit den aus dem Objekt be-

ziehbaren Attributen aufgerufen.

Bedingt durch die fertige Ausführungsumgebung ist die Implementierung von Socket, Kommunikationsprotokoll, Message Handling und Sitzungsverwaltung nicht notwendig. Ergänzend kommen spielbezogene Werkzeuge, Administrationsoberflächen und Nutzerverwaltungen hinzu, die die Entwicklung von Spielen vereinfachen.

## 3.5 Vergleich der Abbildungsformate

Verglichen werden die Abbildungsformate zur Serialisierung der Instanzen aus Abbildung 3.2. Jedes dieser Formate konnte hinsichtlich Nutz- und Strukturdaten analysiert werden. Während z. B. JOS und SmartFoxServer Datentypinformationen, i. A. eine vollständige Beschreibung des Attributs mit zugehöriges Objekt, Typ und Namen enthalten, werden in XML und JSON lediglich Strukturdaten wie Namen der Attribute transportiert. Für den folgenden Vergleich werden Zusatzinformationen, die nicht Nutzdaten sind, zu Strukturdaten zusammengefasst.

Für die Darstellung werden zusätzlich die Längeninformationen von ProtoBuf, JOS und SmartFoxServer von Reihungen und Zeichenketten gesondert dargestellt. Im Fall von XML und JSON ist diese Information nicht notwendig, da sie durch den Syntax explizit vorgegeben sind. Hierfür dienen entweder Anführungszeichen oder die Kombination aus Bezeichner, Zuweisungsoperator, sowie Whitespace oder blockschließende Syntaxelemente wie `>`, `}` oder `,`.

In der Abbildung 3.28 sind die Werte des Beispiels für die Abbildungsformate gegenübergestellt. Für einen Vergleich könnten die Wertlängen ignoriert werden - demnach haben die Nutzdaten von ProtoBuf, JOS und SmartFoxServer im Vergleich zu XML und JSON das kleinste Volumen. Addiert man Nutzdatenlängen und Nutzdaten, ist das Volumen vom JOS und SmartFoxServer Abbildungsformat umfangreicher. Um Gleichheit herzustellen, müssten jedoch die syntaxbedingten Elemente, z. B. Anführungszeichen, zur Markierung der Nutzdatenlängen herausgearbeitet werden. Bezüglich den betrachteten Abbildungsformaten lässt sich feststellen:

- Jedes Format hat einen protokollbedingten Overhead.

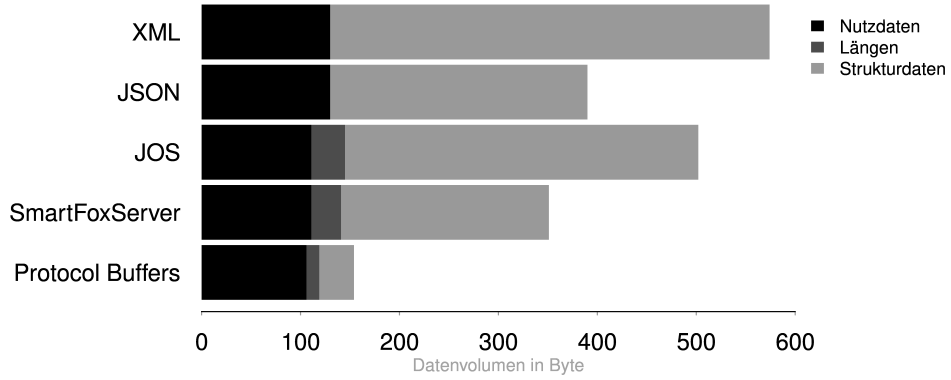



**Abbildung 3.28: Vergleich der Abbildungsformate XML, JSON, JOS, ProtoBuf und SmartFoxServer.**

- ProtoBuf ausgenommen, ist der Overhead im Verhältnis von ca. 1:2 bezüglich der Nutzdaten.
- Teil des Overheads sind Informationen bzgl. Aufbau der Daten, welche in jedem Datenpaket transportiert werden.
- Teil des Overheads sind Namen und Datentypen.
- Die Übertragung von Ganzzahlen, Fließkommazahlen und Wahrheitswerten repräsentiert als Zeichenketten kann zu einer Expansion der Nutzdaten führen. Ausschlaggebend sind die verwendeten Datentypen bzgl. den zu erwartenden Werten.

## 3.6 Vergleich der Beispiel-Implementierungen

Für die Entwicklung eines browserbasierten MMOG-Servers sind – bezogen auf die gezeigten Beispiele in Kapitel 3.2.2 (S. 46), Kapitel 3.3.3 (S. 59), Kapitel 3.2.4 (S. 51), Kapitel 3.2.5 (S. 53) und Kapitel 3.4.5 (S. 74) – verschiedene Elemente notwendig um Funktionalitäten für einen Client bereitzustellen, welche sich im Folgenden charakterisieren lassen. Diese Charakteristiken leiten sich aus der Referenzarchitektur in Abbildung 2.2 ab. Dabei kommen insbesondere die Ebenen zur Realisierung des Servers zum Tragen sowie die zwischen diesen Ebenen existierenden Abhängigkeiten:

1. **Spielmodell**
   Basis Komponente einer Spielidee, für welche Funktionalität bereitzustellen ist.

2. **API**
   Komponenten, die für den Zugriff auf das Spielmodell durch den Controller verwendet werden.

3. **Controller**
   Implementierungen um Anfragen von Clients zu validieren und zu verarbeiten, sowie Ereignisse des Aufrufs am Spielmodell in Antworten für den Client abzubilden.

4. **Kommunikationsobjekte**
   Objekte für den internen Datenaustausch zwischen Message Handler und Controller.

5. **Message Handler**
   Komponente die eingehende Datenströme in interne Kommunikationsobjekte (Anfragen) und ausgehende Kommunikationsobjekte (Antworten) in Datenströme umzusetzen.

6. **Konfigurationen**
   Anpassung der zuvor benannten Elemente durch Konfigurationen um Verhalten zu beeinflussen.

7. **Socketserver**
   Komponenten zur Übermittelung und zum Empfangen von Datenströmen auf Basis des Kommunikationsprotokolls.

8. **Sitzungsverwaltung**
   Komponente zur Verwaltung von aktiven Client-Verbindungen.

Beispiel 5 skizziert, das sechs dieser Charakteristiken notwendig sind, um z. B. die Funktionalität zum Bewegen des Avatars bereitzustellen, der Socketserver und die Sitzungsverwaltung sind nur einmalig notwendig und nicht je bereitgestellte Funktionalität. Das Spielmodell gilt als konstant und initial gegeben.

**Beispiel 5** *Damit der Avatar im Spielmodell aus Abbildung 2.2 bewegt werden kann, enthält der Entitätstyp Avatar die Methode move mit dem Parameter to vom Typ Position. Zur Bereitstellung dieser Methode, bekommt die API eine Methode moveAvatar mit den Parametern id vom Typ int und to vom Typ Position und liefert einen Verweis auf den Avatar zurück. Als Anfrageklasse wird MoveAvatarRequest mit den Parametern der Methodensignatur von moveAvatar in der API definiert, sowie (falls noch nicht verfügbar) ein Antwortklasse AvatarResponse für den Avatar bereitgestellt. Der Message Handler muss einen eingehenden Datenstrom, der die Anfrage zur Bewegung repräsentiert, in eine Instanz von MoveAvatarRequest überführen und an den Controller geben, welcher die erhaltenen Daten für einen Aufruf an der API verwendet.*

Für den Vergleich der verschiedenen Ansätze können die genannten acht Charakteristiken genutzt werden und um drei Charakteristiken erweitert werden, die im speziellen Fall der Spielentwicklungen von Interesse sind:

9. **Nutzerverwaltung**
   Komponenten zur Authentifizierung und Autorisierung von Nutzern sowie der Steuerung dessen Zugriff auf spezifische Elemente des Spielmodells.

10. **Werkzeuge**
    Komponenten zur Unterstützung der Spielentwicklung in Form von Editoren, API und vorbereiteten Strukturen, die im Bereich von Spielen häufig Anwendung finden.

11. **Administration**
    Komponenten zur Verwaltung verschiedener Instanzen, Statistiken und grafische Oberflächen, die den Betrieb des Spiels vereinfachen.

Quellcode 3.3 zeigt die acht Kommunikations- und die drei zusätzliche Spiel-Charakteristiken für den Server. Alle Beispiele erfordern eine Implementierung des Spielmodells, einer API über der mit dem Spielmodell interagiert werden kann und einem Controller der Kommunikationsobjekte auf die API anwendet. Kommunikationsobjekte müssen für alle Strategien bereitgestellt werden

| | | Java TCP<br>TCP/IP Socket | JavaScript REST<br>node.js / express | JavaScript WebSocket<br>node.js / ws | Java REST<br>EJB / JAX-RS | SmartFoxServer |
|---|---|---|---|---|---|---|
| 1 | Spielmodell | | | | | |
| 2 | API | | | | | |
| 3 | Controller | | | | | |
| 4 | Kommunikationsobjekte | | (✓) | (✓) | | |
| 5 | Message Handler | | ✓ | ✓ | ✓ | ✓ |
| 6 | Konfigurationen | ✓ | | | ✓ | (✓) |
| 7 | Socketserver | | ✓ | ✓ | ✓ | ✓ |
| 8 | Sitzungsverwaltung | ✓ | ✓ | ✓ | ✓ | ✓ |
| 9 | Nutzerverwaltung | | | | ✓ | ✓ |
| 10 | Werkzeuge | | | | | ✓ |
| 11 | Administration | | | | | ✓ |

**Tabelle 3.3: Vergleich der verschiedenen Beispiel-Implementierungen. Markiert sind Elemente, welche durch die verwendeten Ansätze nicht für das jeweilige Beispiel umgesetzt werden mussten.**

- ausgenommen sind JavaScript basierte Ansätze, da durch die Spracheigenschaften eine explizite Deklaration nicht notwendig ist. Die Sitzungsverwaltung, das Message Handling und das Kommunikationsprotokoll muss nur im TCP-Ansatz implementiert werden. Die alternativen Strategien haben dies bereits durch die verwendete Strategie vorgegeben. Der Socket ist in allen Ansätzen vorimplementiert und zusätzliche Charakteristiken, speziell für die Spielentwicklung, werden nur vom proprietären System SmartFoxServer bereitgestellt, ausgenommen Administrationswerkzeuge, diese sind ebenso bei Java EE-basierten Ausführungsumgebungen vorhanden.

Basierend auf den Charakteristiken wird im Folgenden versucht den Aufwand zu quantifizieren. Neben einem zeitlichen Aspekt steht hier als quantifizierbares Merkmal das Volumen der Implementierung zur Verfügung – je umfangreicher die Implementierung ist, desto mehr Aufwand ist notwendig um eine Strategie zu realisieren. Bei der Ermittlung dieser sogenannten Lines

of Code (LoC) sind verschiedene Ansätze möglich. Eine Variante ist das Zählen aller verfügbarer LoC einer Implementierung. Der resultierende Wert ist abhängig vom Programmierstil, Leerzeilen und Umfang an Kommentaren, was das Ergebnis der Analyse verzerren würde. Optimieren lässt sich dieser Wert indem bestimmte Zeilen ignoriert werden. Dies betrifft insbesondere leere Zeilen, Kommentarzeilen und Zeilen mit weniger als drei Zeichen (um z. B. Zeilen schließende Klammern herauszufiltern). Der resultierende bereinigte LoC-Wert kann für einen Vergleich verwendet werden, lässt sich weiter für einen objektiven Vergleich optimieren. Hierfür kommt das Kostenmodell COCOMO II zum Einsatz, welches für die Aufwandsabschätzung von Projekten verwendet wird. Dieses Modell beschreibt einen standardisierten Weg zur Ermittlung von sogenannten LLoC. Diese LLoC bereinigen das Zählergebnis von sprachspezifischen Merkmalen, um ein vergleichbares Ergebnis zu erhalten. [76, S. 77]

Zur Ermittlung des Volumens wird ein regelbasiertes Werkzeug eingesetzt, welches für diesen Zweck entwickelt wurde. Das Werkzeug erleichtert das systematische Zählen und Kategorisieren von Quellcode in beliebig vielen Dimensionen. Für die folgende Analyse wurde der Beispiel-Quellcode hinsichtlich der Dimensionen Strategie, Charakteristik, sowie bereitgestellte Funktionalität betrachtet. Stategien $S$ umfassen: Java TCP (1), JS REST (2), JS WebSocket (3), Java REST (4) und SmartFoxServer (5), sowie zwei spezielle Strategien für mehrfach verwendete Java-Implementierungen „All“ (6) und JavaScript-Implementierungen „JS Both“ (7); für die Charakteristiken $C$ werden die bereits herausgestellten Merkmale verwendet: Model (1), API (2), Controller (3), Kommunikationsobjekte (4), Message Handler (5), Konfiguration (7) und Socket (8); und für die Funktionalitäten $F$ werden die Methodensignaturen der *AvatarApi* (siehe Kapitel 2.9, S. 28) sowie ein initialer Zustand verwendet: init (1), *getAvatar* (2), *getAvatars* (3), *moveAvatar* (4), *newAvatar* (5) und *updateAvatar* (6). Im Folgenden werden Strategie, Charakteristik und Funktionalität numerisch angegeben und sind durch deren Mengen $S$, $C$ und $F$ definiert, wobei im folgenden gilt $s \in S, c \in C, f \in F$:

$$S := \{1, 2, 3, 4, 5, 6, 7\} \tag{3.1}$$

$$C := \{1, 2, 3, 4, 5, 6, 7, 8\} \tag{3.2}$$

$$F := \{1, 2, 3, 4, 5, 6\} \tag{3.3}$$

Zur Kategorisierung flossen bei der Analyse mithilfe des Werkzeuges Dateinamen, Ordner und implizites Wissen über interne Strukturen ein. Interne Strukturen wurden in regulären Ausdrücken zum Aufsplitten einer Quellcodedatei abgebildet und dienten dazu z. B. den Aufwand für die Implementierungen einer einzelnen Methode, die zur Realisierung einer Funktionalität notwendig ist, zuzuordnen. Die mittels Regelmenge partitionierten Quellcode-Ausschnitte wurden abschließend mithilfe des Werkzeuges Unified Code Counter (UCC) [75] bewertet um LoC und LLoC zu ermitteln.

Quellcode 3.4 zeigt einen Einblick in das Analyseergebnis, welches im Folgenden für Vergleiche verwendet wird. Das Werkzeug hat z. B. im gezeigten Ausschnitt die Schnittstelle *AvatarApi* in mehrere Einträge aufgeteilt und die resultierenden Teilmengen der jeweiligen Funktionalität zugeordnet. Anzumerken ist, dass im Sinne der LLoC des COCOMO II Modells die Annotationen zur Konfiguration der javabasierten REST-Schnittstelle, z. B. im Fall der *BasicGameApplication*, mit null bewertet werden. Eine vollständige Übersicht über die ermittelten Werte der Beispiel-Implementierung ist in Anhang O (S. 277) zu finden.

| *Datei* | *Strat.* | *Funk.* | *Char.* | *LoC* | *LLoC* |
|---|---|---|---|---|---|
| Avatar | all | init | model | 38 | 20 |
| AvatarApi | all | init | api | 6 | 4 |
| AvatarApi | all | NewAvatar | api | 1 | 1 |
| AvatarApi | all | GetAvatar | api | 1 | 1 |
| AvatarApi | all | GetAvatars | api | 1 | 1 |
| AvatarApi | all | UpdateAvatar | api | 1 | 1 |
| AvatarApi | all | MoveAvatar | api | 1 | 1 |
| BasicGameApplication | rest | init | controller | 8 | 4 |
| BasicGameApplication | rest | init | config. | 1 | 0 |
| BasicGameController | rest | init | controller | 14 | 7 |
| BasicGameExtension | smartfox | init | controller | 42 | 16 |
| BasicWorld | all | init | model | 30 | 18 |

| | | | | | |
|---|---|---|---|---|---|
| model | jsboth | init | model | 70 | 45 |
| Position | all | init | model | 37 | 19 |
| rest | jsrest | init | controller | 37 | 21 |
| rest | jsrest | NewAvatar | controller | 3 | 3 |
| rest | jsrest | GetAvatar | controller | 3 | 4 |
| rest | jsrest | GetAvatars | controller | 3 | 4 |
| rest | jsrest | UpdateAvatar | controller | 3 | 3 |
| rest | jsrest | MoveAvatar | controller | 3 | 3 |
| Server | tcp | init | socket | 34 | 19 |

**Tabelle 3.4: Einblick in das Analyseergebnis für die Beispiel Implementierung.**

Als Ergebnis dieser Messung kann jede Zeile $z \in Z$ der Implementierung für die Ermittlung des Aufwands $a_1(s, c, f)$ der Implementierung bzgl. der Strategie, Charakteristik und Funktionalität bewertet werden. Hierfür sollen die Funktionen $Strategy(x) \in S$, $Characteristic(x) \in C$ und $Functionalty(x) \in F$, zur Ermittlung der Kategorien, und die Bewertungsfunktion $Countable(x) \in [0, 1]$, zur Entscheidung, ob eine Zeile zählbar im Sinne des COCOMO II LLoC Modells ist, dienen. Die Anzahl der daraus resultierenden Zeilen sei wie folgt definiert:

$$\begin{aligned} a_1(s, c, f) = |\{x | x \in Z \wedge \\ \text{Strategy}(x) = s \wedge \\ \text{Characteristic}(x) = c \wedge \\ \text{Functionalty}(x) = f \wedge \\ \text{Countable}(x) = 1\}| \end{aligned} \tag{3.4}$$

Für die Analyse werden die Strategien Java TCP, JS REST, JS WebSocket, Java REST und SmartFoxServer mit den Zusammenfassungen All und JS Both kombiniert:

$$a_2(s,c,f) = \begin{cases} a_1(s,c,f) + a_1(6,c,f) & \text{wenn } s = 1 \\ a_1(s,c,f) + a_1(7,c,f) & \text{wenn } s = 2 \\ a_1(s,c,f) + a_1(7,c,f) & \text{wenn } s = 3 \\ a_1(s,c,f) + a_1(6,c,f) & \text{wenn } s = 4 \\ a_1(s,c,f) + a_1(6,c,f) & \text{wenn } s = 5 \text{und } c < 4 \\ a_1(s,c,f) & \text{sonst} \end{cases} \tag{3.5}$$

Sowie, für deren Vergleich über Programmiersprachengrenzen hinweg, anschließend der mit $a_2(s,c,f)$ ermittelbare Aufwand gegen das Spielmodell der jeweiligen Strategie normalisiert:

$$a_3(s,c,f) = \frac{a_2(s,c,f)}{a_2(s,1,f)} \tag{3.6}$$

Abbildung 3.29 zeigt darauf aufbauend alle Beispiel-Implementierungen mit normalisierten Gesamtaufwänden im finalen Zustand im Vergleich, welche sich aus $a_5(s)$ ergeben:

$$a_4(s,c) = \sum_{f \in F} a_3(s,c,f) \tag{3.7}$$

$$a_5(s) = \sum_{c \in C} a_4(s,c) \tag{3.8}$$

Die Abbildung 3.29 teilt den Gesamtaufwand je Strategie zusätzlich in die Aufwände (welche sich aus $a_4(s,c)$ ergeben) je Charakteristik und verdeutlicht, was in welcher Strategie zzgl. zum Spielmodell welchen ergänzenden Aufwand bedeutet. Im Durchschnitt liegt das dargestellte Verhältnis zwischen dem Aufwand für die Spielmodell- und der Infrastruktur-Implementierung ($c \in [2,8]$) bei ca. 1:3,01. Im Idealfall liegt dieser Aufwand im Rahmen der JavaScript WebSockets Strategie bei 1:1,267, da hier keine Kommunikationsobjekte notwendig sind.

Die Resultate, die mithilfe der Beispiel-Implementierungen gewonnen wur-

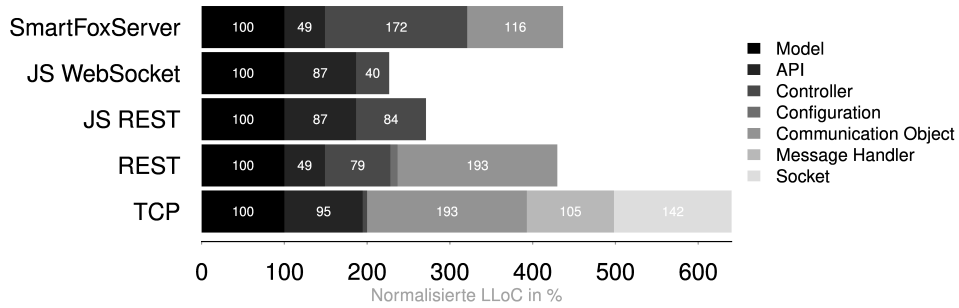



**Abbildung 3.29: Darstellung der normierten Aufwände für die Realisierung des minimalen Spielmodells aus Abbildung 2.2.**

den, dienen für einen Einblick, was in Ergänzung zur Entwicklung des Spielmodells für die jeweiligen Alternativen notwendig ist. Die Ergebnisse lassen sich an dieser Stelle jedoch nicht verallgemeinern. Auch wenn der eigentliche Implementierungsaufwand abhängig vom Autor der jeweiligen Strategie ist, kann jedoch festgestellt werden, dass jedes Werkzeug für sich und dessen verwendbaren APIs anzuwenden ist. Demnach würde ein Aufwand der Höhe 0 bedeuten, dass das Werkzeug nicht verwendet wurde, um eine Funktionalität bereitzustellen. Ziel der Betrachtung der Beispiel-Implementierung ist es zu zeigen, dass zur Bereitstellung von Funktionalität, in egal welcher Strategie, ein Mehraufwand notwendig ist, womit in diesem Fall gilt:

$$a_4(s, 1) < \sum_{c \in C} a_4(s, c) \tag{3.9}$$

Zusätzlich zum Verhältnis zwischen Spielmodell- und Infrastruktur-Implementierung, sowie der Existenz des Mehraufwandes kann betrachtet werden, wie sich der Aufwand je Funktionalität entwickelt:

$$a_6(s, f) = \sum_{c \in C} a_3(s, c, f) + a_6(s, f - 1) \tag{3.10}$$

Analysiert man das normalisierte Ergebnis für $a_6(s, f)$ für jedes $s \in S$, ergibt diese eine kumulierte Aufwandsentwicklung je Funktionalität $f \in F$ wie in Abbildung 3.30 dargestellt. Die Darstellung zeigt, dass auf Basis eines konstant

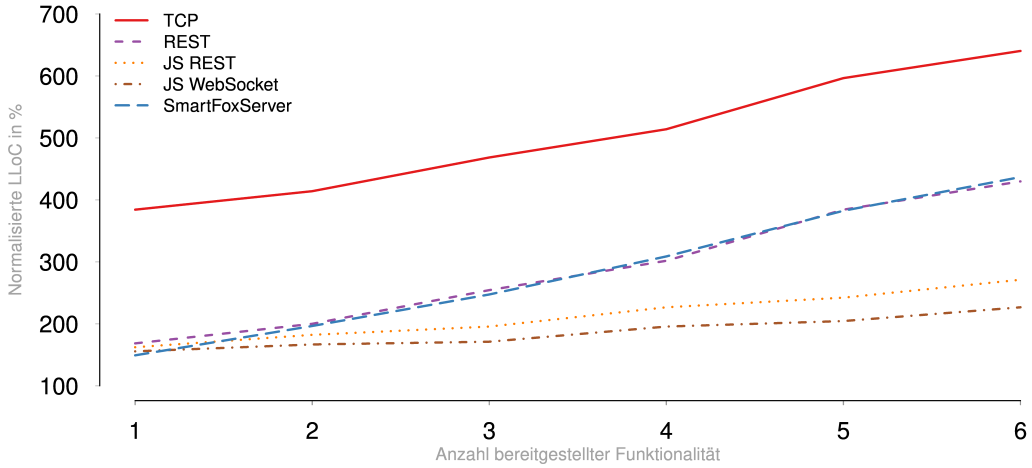



**Abbildung 3.30: Darstellung der normierten Entwicklung des Aufwands je Entwicklungsschritt für die Realisierung des minimalen Spielmodells aus Abbildung 2.2.**

angenommen Spielmodells für jede exponierte Funktionalität ein zusätzlicher Aufwand entsteht, womit im Fall der Beispiel-Implementierungen gilt:

$$a_6(s, f) > a_6(s, f-1) \quad \text{wenn } s > 1 \tag{3.11}$$

Im Anhang P (S. 281) werden alle kumulierten Aufwände je Funktionalitäten detailliert gegenübergestellt. Diese Darstellungen zeigen, dass sich der identifizierte Aufwand in der Erweiterung der API, dem Controller und den Kommunikationsobjekten begründet. Da im Beispiel jede neue exponierte Funktionalität für ein Spielmodell eben jene Schritte benötigt ist nicht davon auszugehen, dass dieser Anstieg konvergiert. Je komplexer die mögliche Interaktion mit dem Spielmodell, desto mehr Schritte entstehen und desto größer der Overhead, der für die Entwicklung der Infrastruktur zu erwarten ist. Ebenso verdeutlichen diese Beispiele, dass dieser Overhead in der Implementierung über alle Beispiele hinweg im besten Fall ein Verhältnis von 1:1,267, im Durchschnitt ein Verhältnis von 1:3,01 einnimmt. Vermeiden lässt sich dieser Overhead mit keinem Werkzeug. Der Overhead kann darüber hinaus als notwendiger, mit dem Umfang der bereitgestellten Funktionalität stetig steigender, Aufwand zur Realisierung eines Spiels aufgefasst werden.

# 4. Thesen

Kapitel 2 (S. 7) und Kapitel 3 (S. 31) skizzieren im Rahmen einer Grundlagenbetrachtung, was Spiele sind und wie diese realisiert werden können. Mithilfe eines minimalen Beispiels, basierend auf einem erarbeiteten abstrakten Spielmodells, wurden verschiedene Ansätze realisiert, charakterisiert und anschließend verglichen. Im Ergebnis zeigt die bisherige Betrachtung, dass die Ansätze die Entwicklung von MMOGs unterstützen, jedoch ein Verhältnis zwischen Spielmodell und Overhead von 1:3,01 haben. Basierend auf dieser Betrachtung und unter der Fragestellung bezüglich der Redundanz und Vermeidbarkeit von verschiedenen Elementen innerhalb einer MMOG Architektur lassen sich die folgenden Thesen formulieren.

Grundelemente von Spielen sind Regeln die im Rahmen einer „pretended reality“ gelten, sowie mindestens ein nicht triviales Ziel des Spiels. Unabhängig vom Genre, Art des Spiels oder ob von einem MMOG die Rede ist, sind diese Elemente in Spielkonzepten wiederfindbar. Wie bereits in Kapitel 2 (S. 7) hergeleitet, ist ein abstraktes Spielmodell denkbar. Dieses kann insbesondere für die Analyse eines Spiels genutzt werden, um die spezifischen Entitäten zu klassifizieren. Ein solches Enterprise Gaming Metamodel (EGM) kann die Entwicklung auf Basis einer initialen Struktur, die sich auf verschiedene Genre anwenden lässt, vereinfachen und im Folgenden von einer Ausführungsumgebung genutzt werden, um das Spiel zu analysieren:

**These 1** *Enterprise Gaming Metamodel: Spielkonzepte lassen sich auf einen EGM-Standard reduzieren.*

In Ergänzung zu einem EGM konnte bereits in einem vereinfachten Beispiel demonstriert werden, dass bisherige Entwicklungsprozesse und Werkzeuge einen Overhead zur Bereitstellung von Funktionalität zeigen. Wird eine Spielidee als browserbasiertes MMOG realisiert, stellt sich neben der eigentlichen

Logik des Spiels ein nicht unbedeutender und mit dem Umfang des Spiels stetig steigender Overhead ein. Dabei ist es in erster Linie unabhängig ob Message Handler selbst implementiert werden oder durch Werkzeuge und Frameworks vorgegeben sind – der Overhead ist notwendig um die Funktionalitäten bereit zustellen:

**These 2** *Aktuelle Architekturen und die damit verbundenen Entwicklungsprozesse weisen vermeidbaren Overhead zur Bereitstellung von Funktionalität für eine Client-Anwendung auf.*

Die Entwicklungsschritte zur Bereitstellung von Funktionalitäten sind selten komplex und bedienen Arbeitsschritte, die letztlich auf die Anwendung des Spielmodells zielen. Denkbar wäre, dass diese Schritte aus dem bereits verfügbaren Modell abgeleitet werden. Eine Ausführungsumgebung dient dann zur Identifikation von bereitzustellenden Funktionalitäten, initialisiert die notwendige Infrastruktur und delegiert Aufrufe an das Spielmodell. Mit dem Ziel eines standardisierten javabasierten Containers soll einem Spielmodell eine Umgebung geboten werden, die die notwendige Infrastruktur durch modellunabhängige Elemente, wie einer spezifischen Konfiguration, bereitstellt.

Diese Ausführungsumgebung zur Bereitstellung von Funktionalitäten muss in herkömmlichen Plattformen integrierbar sein. Als Entwicklungsumgebung wird dafür Java genutzt, wobei die verfügbaren Spezifikationen im Enterprise-Bereich nur bedingt effizient auf Spielszenarien anwendbar sind. In diesem Umfeld ist zu berücksichtigen, dass Spiele nutzerunabhängige Prozesse benötigen um deren eigenständige, z. B. zeitbedingte, Entwicklung sicherzustellen. Die zu schaffende Ausführungsumgebung soll sich in das Ökosystem der Java EE integrieren und existierende Spezifikationen nutzen, um ein Spielmodell bereitzustellen. Die Kombination von Infrastrukturbereitstellung und abstrakten Spielmodell als Entwicklungsbasis, unter Berücksichtigung vererbter Spezifikationsanforderungen, schafft einen Ausführungscontainer, der auf dauerhaft aktive, simulationsähnliche Modelle ausgelegt ist: der Game Container.

**These 3** *Der Game Container: Ein EGM-Standard ist in gängigen Ausführungsumgebungen lauffähig, insbesondere im JBoss.*

Mit der steigenden Komplexität einer Anwendungen, insbesondere eines Spiels, erhöht sich die Menge an bereitzustellenden Funktionalitäten. Resultie-

rend aus den Thesen 1, 2 und 3 ist es möglich eine Ausführungsumgebung zu schaffen, die unabhängig von der Clienttechnologie eine Kommunikationsform etabliert bei der der direkte Nachrichtenaustausch hinter objektorientierter Logik gekapselt ist. Mithilfe von Code Generatoren, Analysen von Methodensignaturen und Objektattributen, sowie bereitgestellter Infrastruktur lassen sich gängige Aufgaben automatisieren und die Spielentwicklung auf Client- und Serverseite ohne Details bzgl. dem Kommunikationsprotokoll, dem Controller, der API und dem Message Handler realisieren:

**These 4a** *Spieleentwicklung ist ohne direkte Anwendung eines spezifischen Kommunikationsprotokolls, unter besonderer Berücksichtigung einer automatisierten Modellsynchronisation und plattformunabhängigen entfernten Methodenaufrufen, möglich.*

Um objektorientierte Client-Server-Kommunikation existieren generische bzw. abstrakte oder performante Implementierungen. Ein clienttechnologieunabhängiger generischer Ansatz, bei dem kein zusätzlicher Entwicklungsaufwand und kein Verlust von Performance entsteht, bei gleichzeitig minimalem Datenpaketvolumen, ist für die geforderte Reaktionszeit von Spielen erforderlich. Unter besonderer Berücksichtigung technologiespezifischer Datentypen und der Minimierung von zeichenkettenbasierter Verarbeitungslogik von numerischen und boolschen Typen soll das Abbildungsformat Eigenschaften von Entitäten über die Technologiegrenzen hinweg erhalten und Daten ohne Informationsverlust transportieren.

**These 4b** *Eine typsiche Objektserialisierung ist metadatenfrei mit minimalem Performance-Overhead erreichbar als Basis für plattformunabhängige Kommunikation.*

# 5. Architekturentwurf

MMOG-Server Architekturen wie der gezeigte Java TCP-Server, die Java EE oder JavaScript basierten Anwendungen, sowie Lösungen mithilfe des SmartFoxServer, die Photon Game Engine oder der ES5 haben im gezeigten Beispiel einen messbaren Overhead bei der Implementierung, der mit der Anzahl der bereitgestellten Funktionalitäten steigt (siehe Kapitel 3.6, S. 78). Jedes dieser Werkzeuge bietet Unterstützung zur Anwendung des selbigen, teilweise in Bezug zu einer bestimmten Zielgruppe bzw. einem bestimmten Anwendungsbereich. Der SmartFoxServer ist z. B. darauf ausgelegt die Spielentwicklung zu unterstützen und bietet hierfür Ansätze, um die Verteilung von Nachrichten und Ereignissen in Zonen und Räumen zu unterteilen. Die Java EE bietet umfangreiche Spezifikationen und Standards, mit denen sich die verschiedenen Schichten einer multi-tier Anwendung realisieren lassen. Webdienste auf Basis von JavaScript lassen sich agil umsetzen und mithilfe von zusätzlichen Modulen können Standards und Paradigmen für den Datenaustausch angewendet werden. Diese Werkzeuge haben eines gemeinsam: sie optimieren den Umgang mit ihnen. In einem konkreten Anwendungsfall wird das gewählte Werkzeug genutzt, um Funktionalitäten der eigenen Logik zu exponieren. Dieses Vorgehen wird mit der Realisierung dieser Architektur invertiert. Basierend auf einem bereitgestellten Spielmodell wird eine Ausführungsumgebung die notwendigen Schritte ableiten, die zur Bereitstellung der Funktionalitäten notwendig sind. Die folgende Architektur beschreibt die Komponenten und Lösungsstrategien, die zum Erreichen dieser Zielstellung notwendig sind.

In Anlehnung an das Architektur-Template arc42 [69, 80] wird der fachliche und technische Kontext, die möglichen Lösungsstrategien, die Bausteinsicht auf Level 1 und 2 sowie die getroffenen Entwurfsentscheidungen skizziert und bilden die Basis für die Realisierung der Ausführungsumgebung Kapitel 6 (S. 123). Bezüglich dem Template arc42 wird auf eine detaillierte Betrachtung

der Verteilungs- und Laufzeitsicht, den Qualitätsszenarien und der Risiken verzichtet. Diese Architekturdokumentation zielt auf die Darstellung der Strategien und den damit in Verbindung stehenden Komponenten, erhebt jedoch keinen Anspruch auf Vollständigkeit.

## 5.1 Aufgabenstellung

Die in Kapitel 3 (S. 31) herausgearbeiteten Beispiele zur Realisierung eines browserbasierten MMOGs und den in Kapitel 2.8 (S. 20) zusammengetragenen Anforderungen ermöglichen die Ableitung der Aufgabenstellung für die zu skizzierende Ausführungsumgebung. Ziel der Ausführungsumgebung ist es, Lösungsstrategien bereitzustellen die die Entwicklung auf Basis eines Spiels, bzw. dessen Spielmodells, unterstützt, um die gezeigten Anwendungsdefizite bei den Realisierungen des Beispiels zu vermeiden. Die gesammelten Anforderungen aus Kapitel 2.8 (S. 20) ergeben dabei die folgenden Aufgaben für die Ausführungsumgebung:

- Ausführbarkeit eines Spielmodells, insbesondere ihren Spielobjekten innerhalb einer dauerhaft aktiven Spielwelt
- Verarbeiten von zeitbasierten und bedingten Ereignissen
- Etablierung einer Infrastruktur zur Kommunikation zwischen Client und Server
- Verfügbarkeit der ausgeführten Spielwelt für den parallelen Zugriff von mehreren Spielern
- Realisierung einer performante Kommunikationsinfrastruktur
- Persistenz des Spielmodells über die Ausführung hinaus
- Möglichkeit zur Ergänzung der ausgeführten Spielmodelle, um zusätzliche Dienste (z. B. Authentifizierung, Foren, Wikis, Bezahldienste, Nutzerkontenverwaltung, Nachrichten, Statistiken und Blogs)

## 5.2 Qualitätsziele

Die zentralen Qualitätsziele der Ausführungsumgebung werden in der folgenden Liste dargestellt. Sie gelten als Rahmen bei der Realisierung und den verwendeten Lösungsstrategien. Ihre Abfolge bildet eine grobe Bewertung der Wichtigkeit ab und beginnt mit dem wichtigsten Qualitätsziel.

- **Geringe Einstiegshürden**
  Die Anwendung der Middleware muss mit geringen Einstiegshürden möglich sein. Grundlegende Mechanismen müssen verständlich und intuitiv realisiert werden.

- **Konvention vor Konfiguration**
  Klare Konventionen sollten der Notwendigkeit zur Konfiguration vorgezogen werden.

- **Stabile Ausführung**
  Die Ausführungsumgebung muss unter Last stabil Anfragen bedienen können.

- **Wartungsarme Ausführung**
  Die Ausführungsumgebung muss wartungsarm und zuverlässig Spiele bereitstellen.

- **Skalierbarkeit**
  Die Ausführungsumgebung muss skalierbar sein um an steigende Nutzerzahlen anpassbar zu sein.

- **Erweiterbarkeit**
  Komponenten, insbesondere Kommunikationsprotokoll und Abbildungsformate, müssen austauschbar bleiben.

## 5.3 Randbedingungen

Die Ausführungsumgebung muss technische Randbedingungen berücksichtigen. Diese adressieren Werkzeuge, Abhängigkeiten und dessen Ausführbarkeit

unter verschiedenen Bedingungen und ermöglichen ein breites Einsatzspektrum.

- **Verschiedene Clienttechnologien**
  Die betrachteten Lösungen zur Realisierung von browserbasierten MMOGs, wie der SmartFoxServer, zeigen in ihrem Vergleich eine hohe Bandbreite an verwendbaren Clienttechnologien. Diese Vielfalt muss erhalten bleiben.

- **Implementierung in Java**
  Die Ausführungsumgebung und Spielmodelle sind in der Java SE 5, 6 und 7 ausführbar.

- **Realisierung auf Basis von Java EE**
  Die Ausführungsumgebung muss in einen Java EE 6 zertifizierten Application Server ausführbar sein.

- **Unabhängig und selbstständig Ausführbar**
  Die Ausführungsumgebung muss unabhängig von einem Java EE Application Server ausführbar sein.

- **Freie Verfügbarkeit von Fremdsoftware**
  Abhängigkeiten der Ausführungsumgebung müssen frei verfügbar sein, dies Betrifft insbesondere Bibliotheken, die von der Ausführungsumgebung verwendet werden.

- **Betriebssystemunabhängige Entwicklungsmöglichkeit**
  Die Entwicklung von Spielen auf Basis der Ausführungsumgebung soll betriebssystemunabhängig möglich sein, insofern Java für das Betriebssystem zur Verfügung steht.

- **Ausführbarkeit auf verschiedenen Serversystemen**
  Die Ausführung des Systems soll auf verschiedenen javafähigen Serversystemen möglich sein.

## 5.4 Kontextabgrenzung

Die Kontextabgrenzung skizziert die zu leistenden Merkmale des Systems [80] aus fachlicher und technischer Sicht. Der fachliche Kontext führt Komponenten auf, die für die Realisierung der Ausführungsumgebung aus inhaltlicher Sicht notwendig sind. Der technische Kontext platziert die im System verwendeten Komponenten, wie Datenbank, Dienstleister, Ausführungsumgebung und Client aus technischer Sicht und setzt sie in Abhängigkeit.

### 5.4.1 Fachlicher Kontext

Aus fachlicher Sicht existiert im Kontext dieser Ausführungsumgebung ein Spieler – i. A. der Akteur, der mit dem System interagiert – der Zugang zu Informationen und Diensten benötigt. Diese sind im Komponentendiagram in Abbildung 5.1 dargestellt. Im vereinfachten Fall ist dies der Zugang zum Zustand des Spielmodells. Im erweiterten Fall begrenzt sich dieser Zugang zum Zustand auf eine spielerzentrische Teilmenge und ergänzt sich um Visualisierung, Möglichkeiten zur Interaktion, Mechanismen um den Zugang zum Spielzustand zu schützen (z. B. durch klassische Nutzerkonten mit Authentifizierung), sowie Dienste bezüglich Bezahlung, Informationsbereitstellung und Kommunikationsplattformen.

- **Game State**
  Der vollständige Zustand des Spieles. Im Allgemeinen der Zustand aller Spieler und Spielobjekte in einer Spielwelt.

- **Personalized Game State**
  Spielerzentrische Sicht auf den Zustand des Spieles.

- **Visualization**
  Visualisierung des eigenen Zustandes des Spiels, i. A. die grafische Darstellung im Client.

- **Interacting**
  Möglichkeiten mit dem Spielmodell, basierend auf dem eigenen Zustand, zu interagieren.

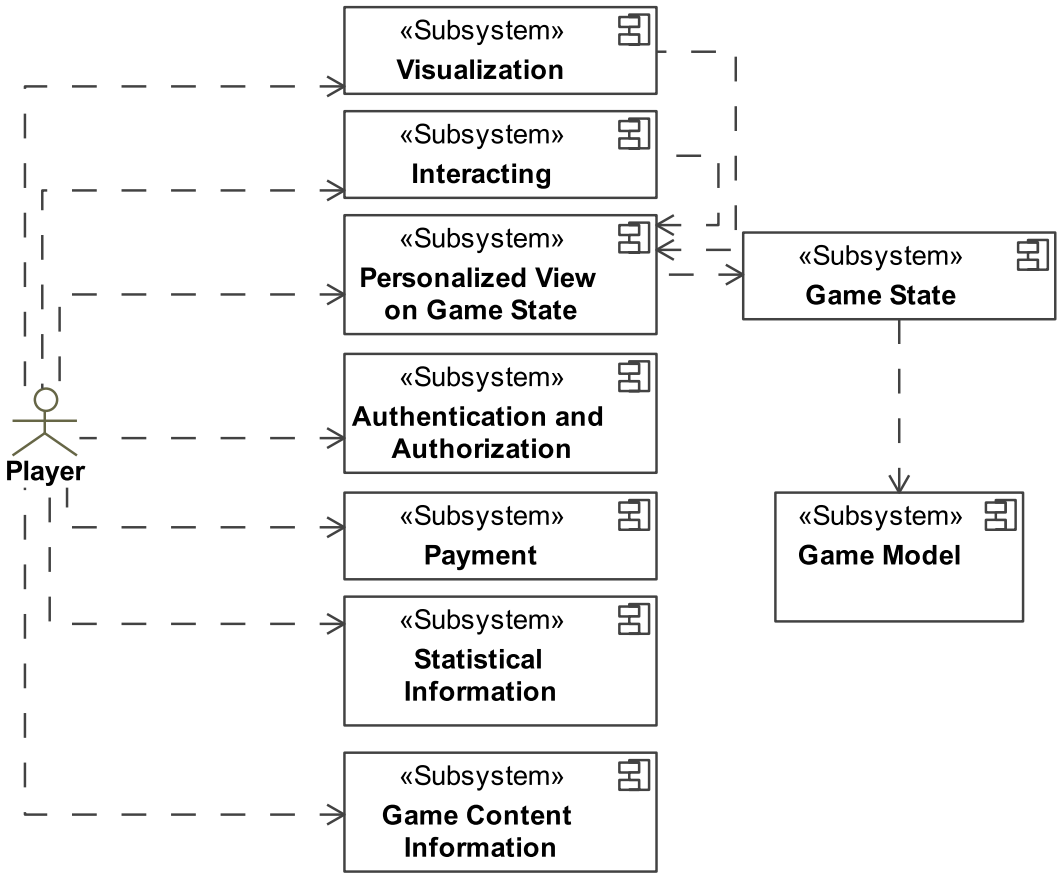



**Abbildung 5.1: Komponentendiagramm des fachlichen Kontexts der Architektur der Ausführungsumgebung.**

- **Authentication and Authorization**
  Mechanismen zum Autorisieren und Authentifizieren des Nutzers. Diese Komponente ist besonders im Fall von MMOGs von Bedeutung um den Spieler in unterschiedlichen Sitzungen exakt identifizieren zu können.

- **Payment**
  Bezahlmöglichkeiten für spezielle Features im Spiel zur Monetarisierung des Spiels.

- **Statistical Information**
  Statistische Informationen die aus dem Spiel gewonnen und erstellt werden und öffentlich oder privat bereitgestellt werden.

- **Game Content Information**
  Informationen zu den Inhalten des Spiels in Form, z. B. Wikis und Blogs.

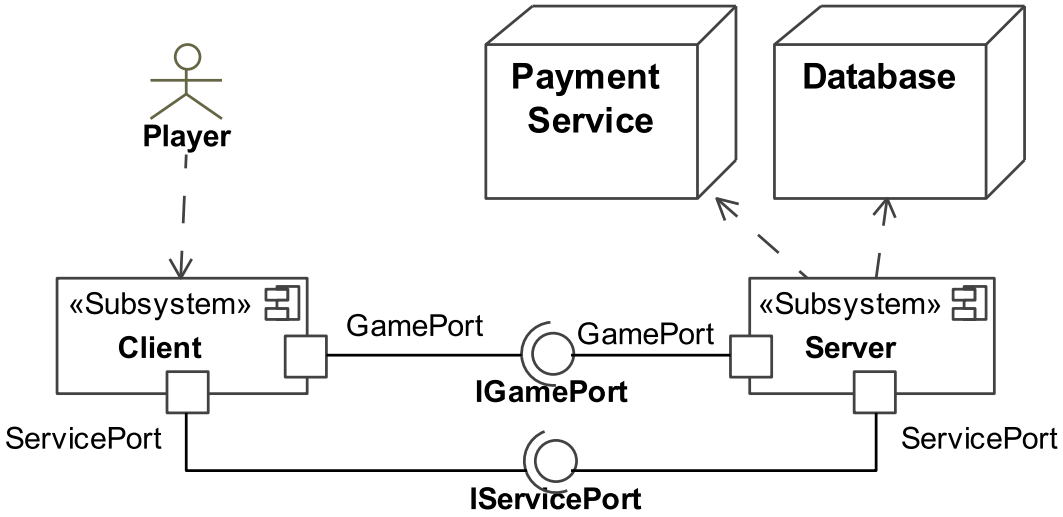



**Abbildung 5.2: Komponentendiagramm des technischen Kontexts der Architektur der Ausführungsumgebung.**

### 5.4.2 Technischer Kontext

Die Ausführungsumgebung differenziert sich in der technischen Betrachtung im Komponentendiagramm in Abbildung 5.2 in die Betrachtung eines Clients und eines Servers. Der Client wird vom Spieler, i. A. der Akteur der mit dem System interagiert, genutzt und tauscht über zwei vom Server bereitgestellte Schnittstellen Daten aus.

Die Kommunikation über die *IGamePort* Schnittstelle kann mittels UDP und TCP realisiert werden. Sie dient zur direkten Kommunikation mit der Spielwelt. Die Kommunikation über die *IServicePort* Schnittstelle dagegen dient für Service Aufgaben, wie Authentifizieren, Bezahlung und zusätzliche Informationsplattformen.

- **Server**
  Der Server, auf dem die beschriebene Ausführungsumgebung läuft und das Spielmodell ausgeführt wird. Der Server stellt zwei Schnittstellen für den Client bereit und inkludiert Komponenten, die die Infrastruktur aus dem Spielmodell für die Schnittstelle *IGamePort* ableitet.

- **Client**
  Die vom Spieler verwendete Komponente. Sie verwendet die Schnittstellen, die der Server bereitstellt.

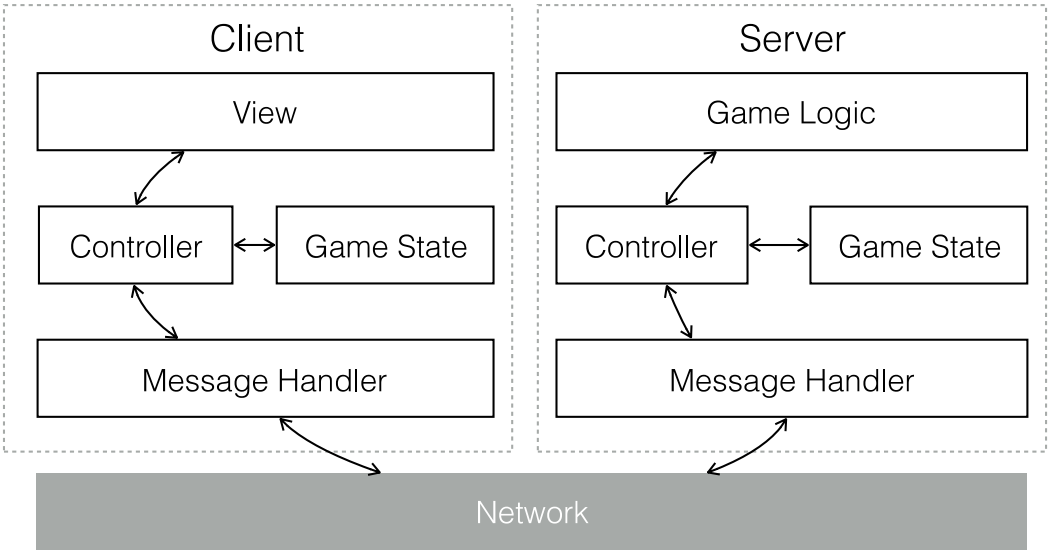



**Abbildung 5.3: Überblick der Architektur für ein browserbasiertes MMOG auf Basis einer Referenzarchitektur [21].**

- **IGamePort**
  Schnittstelle zur Interaktion mit einem Spielmodell.

- **IServicePort**
  Schnittstelle zur Interaktion mit Diensten zu einem Spiel.

## 5.5 Lösungsstrategie

Die Umgebung bringt ein Spielmodell zur Ausführung und stellt dessen Funktionalitäten für Clients bereit. Zur Vermeidung des in Kapitel 3 (S. 31) herausgearbeitete Overheads gilt es, die herkömmliche Art im Umgang mit Werkzeugen zu invertieren. Untersuchte Werkzeuge forderten, dass die vom Werkzeug bereitgestellten APIs angewendet werden, um Funktionalität für einen Client zugreifbar bereitzustellen. Folgend wird die Bereitstellung aus dem Spielmodell durch das Werkzeug abgeleitet und mittels Automatisierung der identifizierte Overhead generiert.

Zur Erreichung dieser Ziele werden im Folgenden Lösungsstrategien dargestellt, die die Anforderungen zur Unterstützung der Entwicklung auf Basis des Spielmodells adressieren. Dieser Architekturentwurf für eine Ausführungsumgebung basiert auf einer Referenzarchitektur für MMOGs [21] welche ab-

gewandelt in Abbildung 5.3 dargestellt wird. Die Abwandlung reduziert die Komponenten zur Visualisierung im Client auf einen View, greift Erkenntnisse aus den realisierten Beispielen in Kapitel 3 (S. 31) und greifen die Aufgabenstellung aus Kapitel 5.1 (S. 94) auf.

Im der dargestellten Überblick zur Architektur in Abbildung 5.3 dient der *Message Handler* als Bindeglied zwischen Client und Server. Dieser Handler wandelt interne Kommunikationsobjekte in Datenströme und Datenströme in interne Kommunikationsobjekte. Ergänzend dient dieser Handler zur Weitergabe ankommender Datenströme an die jeweiligen *Controller* auf Client- und Serverseite. Auf der Serverseite validiert und bearbeitet der *Controller* die Kommunikationsobjekte und ruft die Modelmethoden je nach Anfrage auf. Das Ergebnis des Aufrufs wird persistiert und die Antwort über den *Message Handler* zurück an den Client gegeben. Auf der Clientseite werden Kommunikationsobjekte in dessen *Controller* validiert und bearbeitet um einen lokal verwalteten Zustand des Spielmodells zu aktualisieren und anschließend Prozesse anzustoßen, die den *View* des Spiels aktualisieren. Nutzerereignisse im *View* werden durch den *Controller* entgegen genommen und an den Server in Form von Anfragen weitergegeben.

Abbildung 5.4 zeigt, basierend auf der Übersicht in Abbildung 5.3, eine Ansicht der Ausführungsumgebung. Während in Abbildung 5.3 Komponenten zur Realisierung eines Spiels dargestellt werden, positioniert Abbildung 5.4 diese Komponenten zur Realisierung der Ausführungsumgebung, welche mehrere Spielmodelle parallel verwaltet. Automatisiert und generisch bereitgestellt werden in diesem Aufbau die Kommunikationsobjekte, die Behandlungsroutinen von Anfragen, die Abbildung von Kommunikationsobjekten in Datenströme und der Prozess zur Persistierung des Spielmodells sowie das Modellabbild auf der Clientseite. Die Kommunikationsprotokolle, zur Bereitstellung von Web-, Flash- und Binary-Sockets über einen TCP- oder UDP-basierten Kanal, können statisch realisiert und von generischen Konzepten angewendet werden.

Der *Game Model Container* dient als zentrales Element bei der Realisierung der Ausführungsumgebung. Dieser kapselt verschiedene Spielmodelle und erhält eingehende Anfragen über das *Request Handling* sowie unabhängig entstehende Ereignisse die während der Simulation des Spielmodells zuverarbeiten sind.

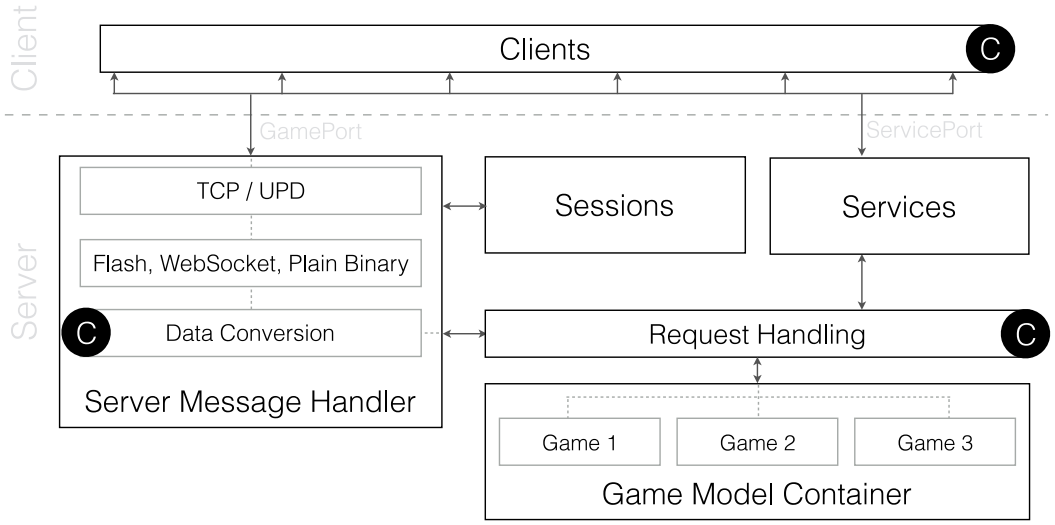



**Abbildung 5.4: Darstellung des Grundaufbaus, Elemente die durch einen Code Generator unterstützt werden sind mit einem „C“ markiert.**

Der *Server Message Handler* dient der Verarbeitung von Anfragen und Antworten. Diese werden durch ein Abbildungsformat (*Data Conversion*) serialisiert, bzw. deserialisiert. Da diesem Element bei der Kommunikation zwischen Server und Client eine hohe Last zukommt, ist dessen Effizienz entscheidend. Die Bedeutung der Performance dieser Komponente kann ebenso in existierenden Lösungen wiedergefunden werden (siehe Kapitel 3.4, S. 61). Wichtig bei dem Transport ist die Berücksichtigung der verfügbaren Datentypen in Client- und Serversystemen, sowie der Erhalt der Objektstrukturen um zu verhindern, dass durch den Datenaustausch im Client zusätzlicher Aufwand zur Transformation anfällt. Zur Differenzierung der Verarbeitung verschiedener Kommunikationsprotokolle, i. A. Websockets, Flash Sockets oder schlicht binär, ist eine Zwischenschicht notwendig. Diese Zwischenschicht wird auf spezifische Merkmale der entsprechenden Protokolle reagieren und den Kommunikationskanal für den jeweiligen Client modifizieren. Abschließend werden die Daten durch den TCP- oder UDP-Socket mit dem Client ausgetauscht.

Die *Services* stellen die im fachlichen Kontext genannten Dienste zur Bezahlung, inhaltlichen und statistischen Spielinformationen sowie Authentifizierung und Autorisierung bereit. Diese Dienste werden vom Client als Webservices über HTTP eingebunden. Der *Game Model Container* nutzt die Services um Informationen zu prüfen, z. B. ob ein Nutzer, der über den *Message Handler*

mit dem Spiel kommuniziert, authentifiziert ist.

Zur Abbildung des gesamten Kommunikationsablaufes – vom Spielmodell bis zur Visualisierung im Client – sind je nach Anwendungsfall verschiedene Client-Implementierungen notwendig. Die Lösungsstrategie dieser Ausführungsumgebung sieht einen Code Generator zur Unterstützung vor, der aus dem Spielmodell die jeweilen Elemente ableitet und damit die erforderlichen Model- und Kommunikationsobjekte auf Clientseite automatisiert repliziert. Die Ableitung übersetzt öffentliche Methoden in Form eines Proxies und hält Attribute auf Clientseite vor. Die Algorithmik der Methoden wird nicht übersetzt.

### 5.5.1 Kommunikationsobjekte und Verarbeitungsinstruktionen

Zur Optimierung der Entwicklung eines MMOGs muss der Overhead zur Bereitstellung von Funktionalität minimiert werden und durch die Verwendung von bereits vorhandenen Informationen im Spielmodell erzeugt werden. Bezüglich den herausgearbeiteten Charakteristiken, ist die Ableitung der Kommunikationsobjekte für Anfragen und Antworten, die Ableitung von zu übertragenden komplexen Datentypen, des Spielmodells in Kommunikationsobjekten und die Verarbeitung der eingehenden Anfragen notwendig. Für das Kommunikationsprotokoll, die Sitzungsverwaltung und den *Server Message Handler* können Werkzeuge genutzt werden, die anschließend von generischen Erzeugnissen aus den Ableitungen des Spielmodells verwendet werden. Dies inkludiert Werkzeuge wie APIs für TCP oder UDP Sockets, Bytecode Generator zur Interpretation von Java Quellcode zur Laufzeit, sowie die durch die Laufzeitumgebung bereitgestellten Dienste.

Im Beispiel aus Kapitel 2.9 (S. 28) sind fünf Funktionalitäten für einen Client bereitzustellen. Am Beispiel der Erstellung eines neuen Avatars, die *newAvatar* Methode der *BasicWorld*, wird in Abbildung 5.5 dargestellt, wie die verschiedenen Klassen aus der Signatur der Methode inkl. dem Rückgabewert hergeleitet werden können. Für die Anfrage wird eine Klasse *NewAvatarRequest* und für die Antwort wird eine Klasse *NewAvatarResponse* erzeugt. Die Klasse *NewAvatarRequest* enthält die in der Signatur wiederzufindenden Pa-

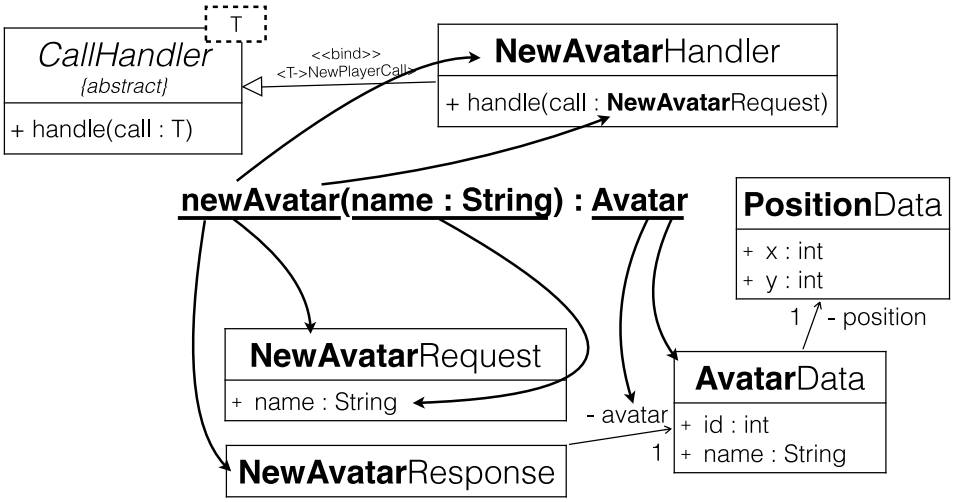



**Abbildung 5.5: Ableiten der notwendigen Elemente zur Vervollständigung des Kommunikationsablaufes am Beispiel der Funktionalität *newAvatar* aus dem minimalem Beispiel aus Kapitel 2.9 (S. 28).**

rameter und die Klasse *NewAvatarResponse* enthält ein Attribut vom Typ des Rückgabewerts. Enthält die Signatur Parameter oder Rückgabewerte mit komplexen Datentypen, ist es erforderlich, diese ebenso abzuleiten. Im Fall der *newAvatar* Methode ist es notwendig, den Rückgabewert vom Typ *Avatar* in eine Datenkapsel *AvatarData* zu überführen. Diese enthält alle öffentlichen, nicht transienten Attribute der abgeleiteten Klasse. Innerhalb des Beispiels entspricht dies der ID und dem Namen sowie der Position. Die Position, welche erneut ein komplexer Datentyp ist, muss ebenso in eine Datenkapsel, der *PositionData*, überführt werden.

Die Verarbeitung der Anfrage wird in der Klasse *NewAvatarHandler* generiert. Diese generierte Anfrageverarbeitung spezialisiert eine abstrakte Klasse *CallHandler*, welche den Umgang mit dem speziellen Handler generalisiert. Die generisch bereitgestellte Anfragebehandlung enthält die notwendigen Instruktionen zur Verarbeitung der Anfrage:

- Validierung der im Kommunikationsobjekt erhaltenen Parameter.
- Auffinden der lokalen Entitäten des Spielmodells.
- Aufruf der Methode.
- Abfangen von möglichen Fehlerzuständen.

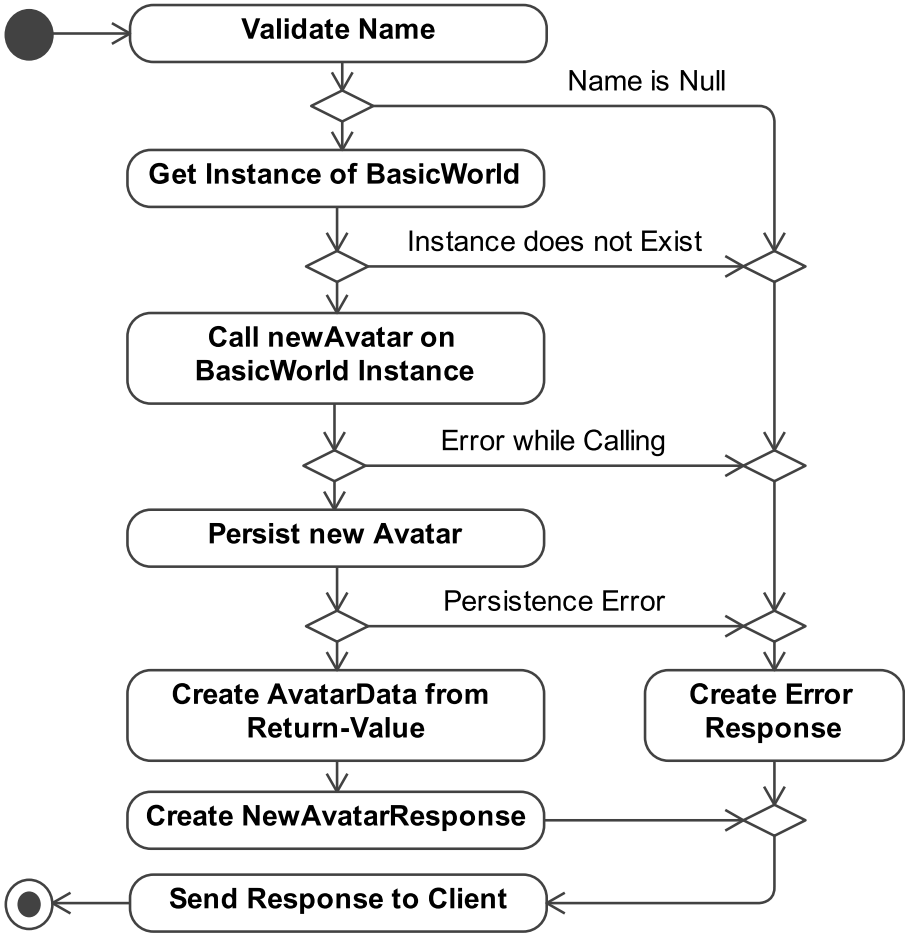



**Abbildung 5.6: Darstellung der Instruktionen zur Verarbeitung einer *NewAvatarRequest* Anfrage innerhalb des *NewAvatarHandler*.**

- Persistieren von Änderungen am Zustand des Modells.
- Erzeugen der Antwort für den Aufruf.
- Im Falle eines Fehlers wird eine Fehlermeldung versendet, im Fall des Erfolges wird die Antwort versendet.

Angewendet auf das Beispiel entstehen Instruktionen zur Verarbeitung wie in Abbildung 5.6 dargestellt. Es muss sichergestellt werden, dass der übertragene Name nicht *null* ist, die Spielwelt bezogen wird und die Methode *newAvatar* am Spielmodell aufgerufen wird. Der neue Avatar muss persistiert werden und als Antwort ist eine Instanz der Klasse *NewAvatarResponse* mit einer Instanz der Datenkapsel *AvatarData* für den neuen Avatar zu versenden.

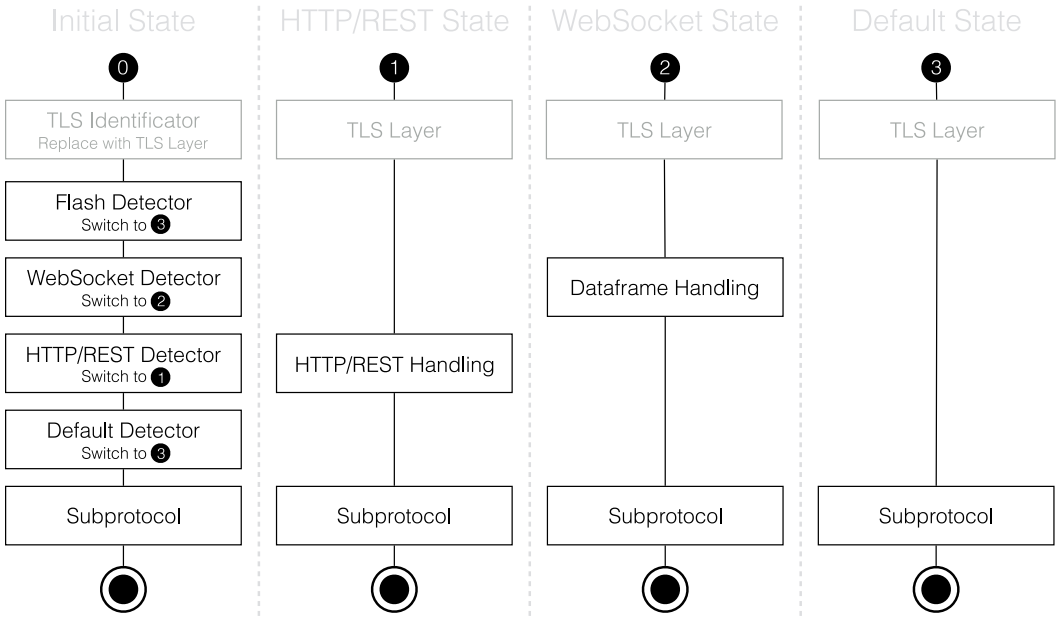



**Abbildung 5.7: Zustände der Verarbeitungspipeline im Server je Client. Beginnend mit dem Initialen Zustand können Detektoren diese Pipeline in verschiedene Zustände rekonfigurieren. Abgeschlossen wird die Verarbeitung immer mit dem Subprotokoll und begonnen mit einem optionalen Layer zur Sicherung der Datenübertragung.**

### 5.5.2 Kommunikationsprotokoll

Für die Kommunikation verwendet die Ausführungsumgebung ein eigenes Subprotokoll. Dieses Protokoll wird auf existierenden Standards aufsetzen und sich je nach verbundenen Client an die spezifischen Merkmale dessen anpassen. Verbindet sich ein Client direkt über TCP, wird der Datenaustausch ohne Zwischenschicht möglich sein. Verbindet sich ein Client mittels WebSocket, wird der Datenaustausch mit dem WebSocket Standard [36] als Zwischenschicht auf Basis von TCP mit diesen Client durchgeführt. Verwendet der Client HTTP, wird die Kommunikation mittels REST auf Basis von HTTP als Zwischenschicht durchgeführt. Verbindet sich ein Flash-Client, dann muss ein initialer XML-Handshake durchgeführt werden und anschließend eine TCP-basierte Kommunikation ohne Erweiterungen verwendet. Die Lösungsstrategie der Middleware sieht eine zur Laufzeit je Client anpassbare Pipeline [19, S. 53] zur Datenverarbeitung, vor welches am Ende der Verarbeitungskette das eigene Subprotokoll zur Interpretation des Datenstroms anwendet.

Die Pipeline zur Datenverarbeitung besitzt einen initialen Zustand, der für neue Verbindungen verwendet wird. Dieser initiale Zustand enthält eine Abfolge von Verarbeitungsschritten zur Detektion von protokollspezifischen Merkmalen und dient der Identifikation des Clients. Wird ein Merkmal identifiziert, rekonfiguriert der identifizierende Verarbeitungsschritt die Pipeline und passt die Verarbeitung an die notwendigen Spezifika des Clients an. Der initiale Zustand enthält als abschließenden Schritt die Verarbeitung des Subprotokolls. Diesem abschließenden Schritt vorgeordnet ist ein Schritt zur Rekonfiguration der Pipeline im Fall des Nichtzutreffens vorgelagerter Schritte zur Identifikation von protokollspezifischen Merkmalen. Abbildung 5.7 zeigt die Zustände grafisch und stellt diese gegenüber.

Das Subprotokoll bedient sich einem austauschbaren Abbildungsformat, verzichtet auf einen zusätzlichen Header und erlaubt einen bidirektionalen Austausch von Paketen. Damit zusammenhängende Datenpakte in einer Sequenz gekennzeichnet sind, insbesondere Sequenzen aus Anfrage und Antwort, muss eine Sequenz-ID zur Identifikation im Datenpaket eingebettet werden.

### 5.5.3 Abbildungsformat

Die Ausführungsumgebung erlaubt den Einsatz eines beliebigen Abbildungsformates für die Kommunikation zwischen Client und Server. Zur Optimierung des protokollbedingten Overheads wird ein austauschbares proprietäres Format vorgeschlagen, welches die in Kapitel 3.5 (S. 77) dargestellten Schwierigkeiten adressiert. Dies bezieht sich auf den bisherigen Transport von Strukturdaten sowie die Abbildung von Wahrheitswerten, Fließ- und Ganzzahlen als Zeichenketten in XML und JSON. Für die Zielformulierung des zu erreichenden Volumens kann das Format ProtoBuf verwendet werden. Dabei ist der Einsatz von ProtoBuf als Abbildungsformat möglich. Das eigene Abbildungsformat, sowie die damit verbundene Logik, soll jedoch den Umweg über eine externe Beschreibungssprache für Nachrichten, wie sie bei ProtoBuf verwendet wird, vermeiden und Informationen eigenständig erheben können.

Ziel des Formates ist es folgende Eigenschaften in Anlehnung an die Betrachtung von JSON, ProtoBuf und XML bei der Abbildung von Objekten zu berücksichtigen:

- Abbildung von Basisdatentypen wie Ganzzahlen, Fließkommazahlen, Wahrheitswerte und Zeichenketten.
- Abbildung von komplexen Datentypen deren Attribute als Basisdatentyp oder komplexer Datentyp deklariert sind.
- Abbildung von Reihungen von komplexen Datentypen oder Basisdatentypen, sowie Reihungen zur Realisierung von Mehrdimensionalität.
- Berücksichtigung von variablen Datentypen, sogenannte Wildcards.
- Berücksichtigung von zyklischen Assoziationen, z. B. Objekt A assoziiert Objekt B welches Objekt A assoziiert.
- Identifikation des Datenpaketes.

### 5.5.4 Spielmodell

Die Ausführungsumgebung stellt ein abstraktes Spielmodell zur Verfügung, welches wiederkehrende Aufgaben bei der Spielentwicklung vereinfacht. Die Anwendung dieses Spielmodells durch Entwickler ist optional. Ein Spielmodell muss diese abstrakte Vorlage nicht spezialisieren, um von der Ausführungsumgebung ausgeführt zu werden. Das abstrakte Spielmodell enthält die in Kapitel 2.6 (S. 14) herausgearbeiteten Elemente, insbesondere eine abstrakte Spielwelt, abstrakte Spielobjekte und Aktionen, die in einem zeitlichen Kontext ausgeführt werden. Zusätzlich werden organisatorische Elemente zur Zeitgebung und Zeiteinheiten bereitgestellt, sowie die Konstruktion von raumbasierten Verwaltungsstrukturen (siehe Kapitel 3.4.1, S. 61) und positionsbasierte Verwaltungsstrukturen (siehe Kapitel 3.4.3, S. 70) unterstützt.

### 5.5.5 Konfiguration

Damit ein Spielmodell in der Ausführungsumgebung bereitgestellt werden kann, sind Informationen zu diesem notwendig, welche mittels Konfiguration bereitgestellt werden müssen. Als primäres Mittel zur Konfiguration dienen Annotationen. Diese definieren Metainformationen und damit grundlegende Parameter zu einem Spielmodell. Als zweite Ebene werden XML Dateien

verwendet, die es erlauben die Parameter zu überschreiben und im Fall von Instanziierungen des Modells jene Parameter zu individualisieren. Folgend werden drei notwendige Konfigurationen und optionale Konfigurationen vorgestellt.

Die erste notwendige Konfiguration betrifft Kommunikationsobjekte. Während es möglich ist, Entitäten ohne zusätzliche Konfiguration mit einem Abbildungsformat zu verarbeiten, ist dies in der Praxis nicht sinnvoll. Java verwendet für die Markierung von serialisierbaren Entitätstypen ein Interface und JAXB erfordert für die Markierung von serialisierbaren Entitätstypen eine Annotation selbiger. Diese bewusste Markierung ermöglicht, dass ggf. nicht zu übertragende Daten nicht zufällig durch Assoziationen im Datenmodell übertragen werden. Die Ausführungsumgebung fordert daher, dass jede Entität, welche zwischen Client und Server ausgetauscht werden, kann mit einer speziellen Annotation markiert wird.

Eine weitere notwendige Konfiguration betrifft die zentrale Spielwelt. Diese Spielwelt muss als Einstiegspunkt mit einer Annotation markiert werden. Die Ausführungsumgebung sucht nach dieser und instanziiert diese bei Start der Ausführungsumgebung.

Die dritte notwendige Konfiguration betrifft die öffentlich bereitgestellten Funktionalitäten. Ähnlich wie bei Kommunikationsobjekten, wäre es möglich jede Funktionalität bereitzustellen, jedoch erfordert ein Markierungszwang der bereitzustellenden Funktionalität eine bewusste Entscheidung und ein Werkzeug zur Kontrolle.

Optional und unter Berücksichtigung der Randbedingung bezüglich Konvention über Konfiguration gelten die folgenden Konfigurationen:

- **Markierung von Identifikatoren**
  Die Entitäten eines Spielmodells müssen identifizierbar sein und benötigen einen eindeutigen Identifikator. Welches Attribut diese Identifikationsaufgabe übernimmt, muss bekannt gegeben werden. Gibt es eine Attribut mit dem Namen ID, so gilt dies als Identifikator der Entität.

- **Markierung von Attributen**
  Entitäten, die kommunizierbar sind, enthalten Attribute, die transportiert und konfiguriert werden müssen. Alle Attribute, die einen öffent-

lichen Getter und Setter besitzen, werden automatisch kommuniziert. Verhindert werden kann dieses Verhalten mittels Markierung als transientes Attribut.

- **Dependency Injection**
  Entitäten die Abhängigkeiten zu Zeitgebern, Datenstrukturen oder anderen Entitäten des Spielmodells haben, können diese per Dependency Injection, vgl. CDI, bereitstellen lassen.

- **Antwort als Aktualisierung von Entitäten des Spielmodells**
  Wird ein kommuniziertes Antwortobjekt empfangen, welches über die Zustandsänderung einer Entität des Spielmodells informiert, so kann dieses mittels Konfiguration an den entsprechenden Entitätstyp gekoppelt werden. Sind Entitäten des Spielmodells kommunizierbar und werden kommuniziert, so gelten sie immer als Antwortobjekt, welche den Zustand aktualisieren.

- **Zugangsbeschränkungen**
  Sind öffentlich bereitgestellte Funktionalitäten unter bestimmten Bedingungen verwendbar, z. B. nur von einem Besitzer, so kann dies mittels Annotation realisiert werden.

## 5.6 Bausteinsicht

Für die Betrachtung der Bausteinsicht wird der technische Kontext aus Kapitel 5.4 (S. 97) weiter differenziert. Abbildung 5.8 zeigt die Differenzierung auf Level 1, bei der die notwendigen Komponenten zur Realisierung der Client- und Serverseite dargestellt werden. Die verwendeten Komponenten lehnen sich an den in Abbildung 5.3 dargestellten Aufbau an.

Der Server in Abbildung 5.8 empfängt über den *GamePort* im *Server Message Handler* Anfragen und sendet die daraus entstehenden sowie die durch Ereignisse erzeugten Antworten an den Client. Die Anfragen werden in der Komponente *Request Handling* in Funktion eines Controllers verarbeitet und an das im *Game Model Container* verwaltete Spielmodell, zum Aufruf der eigentlichen Funktionalität, übergeben. Der *Game Model Container* überwacht im Fall von

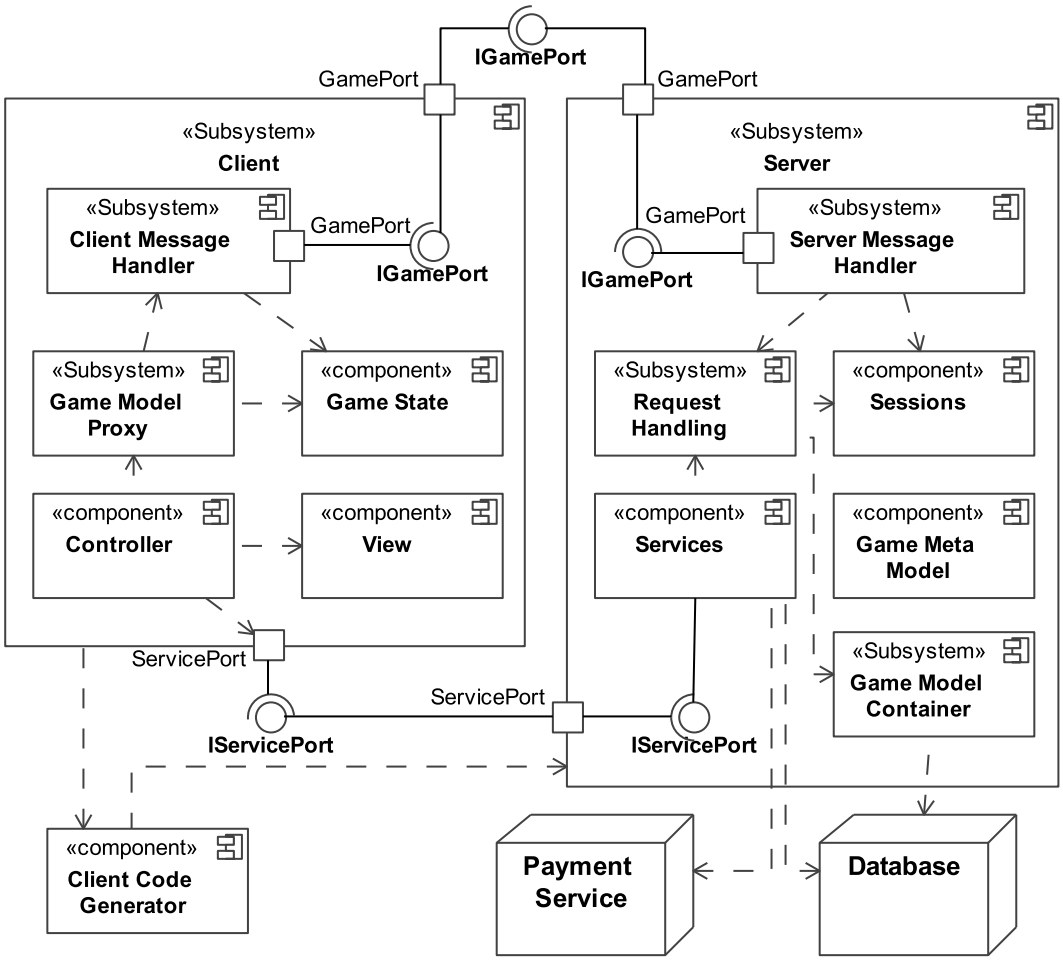



**Abbildung 5.8: Komponentendarstellung auf Level 1 der Architektur als Erweiterung des technischen Kontexts.**

Aufrufen die Veränderungen am Model und persistiert sie bei Bedarf. Die Komponente *Sessions* verwaltet aktive Verbindungen auf Serverseite, sichert Daten bezüglich der Sitzung und verwaltet die durch den Client beobachteten Datenobjekte. Zusätzlich enthält der Server das abstrakte Spielmodell, auf dessen Basis Spielmodelle realisiert werden können. Diese Komponente enthält insbesondere den Zeitgeber, die abstrakten Aktionen, die abstrakte Spielwelt und die abstrakten Spielobjekte sowie die raum- und positionsbasierten Strukturen.

Der Client in Abbildung 5.8 nutzt den *Client Message Handler* als Gegenstück zum *Server Message Handler*. Empfange Antworten werden genutzt um den lokal verwalteten Zustand zu aktualisieren, sowie das *Game Model Proxy* zu informieren, dass eine Antwort eingegangen ist. Der *Controller* im Client überwacht diesen Proxy und kann bei Zustandsänderungen die Anzeige des

Spiels aktualisieren. Nutzerbasierte Ereignisse, z. B. Interaktionen mit dem Spiel, können vom *Controller* abgefangen werden und in Aufrufen auf dem *Game Model Proxy* überführt werden. Dieser erzeugt die notwendigen Anfrageobjekte und übergibt sie für den Versand an den Server zum *Client Message Handler*.

Auf den folgenden Abbildungen zur Darstellung der Komponenten auf Level 2 werden die *Message Handler* auf Client- und Serverseite betrachtet, sowie das *Request Handling*, der *Game Model Proxy* und der *Game Model Container*. Die verbleibenden Komponenten aus Level 1 werden nicht weiter differenziert.

Level 2 des *Server Message Handler* in Abbildung 5.9 zeigt die Komponenten *Event Loop*, *Data Protocol* und *Connection*. Aufgabe des Subsystems *Server Message Handler* ist es, Datenströme entgegen zu nehmen, das Kommunikationsprotokoll (siehe Kapitel 5.5.2, S. 106) zu koordinieren, sowie das Abbildungsformat anzuwenden (siehe Kapitel 5.5.3, S. 107). Die Komponente *Event Loop* wird genutzt um neue Verbindungsanfragen, sowie eingehende und ausgehende Datenpakete auf Basis von systemnahen APIs zur Realisierung von TCP- oder UDP-basierten Servern zu verarbeiten[1]. Die Verarbeitung geschieht auf Basis von Ereignissen und werden im Fall des Eintretens an eine neue oder existierende Verbindung, welche durch die Komponente *Connections* verwaltet wird, disponiert. Diese Verbindung bildet das für diesen Client anzuwendende Kommunikationsprotokoll ab und verarbeitet abschließend mithilfe der Komponente *Data Protocol* das Datenpaket um es in ein Kommunikationsobjekt zu transformieren.

Der Transformation in Kommunikationsobjekte durch den *Server Message Handler* folgt die Verarbeitung der Anfragen im Subsystem *Request Handling*, dargestellt in Abbildung 5.10. Dieses System enthält zur Verarbeitung die Komponente *Dispatcher* und *Request Handler Pool*, welcher auf eine Menge von *Generative Request Handler* und *Custom Request Handler* zurückgreift. *Generative Request Handler* werden wie in Kapitel 5.5.1 (S. 103) beschrieben bereitgestellt und können im Fall der Verarbeitung des zugehörigen Anfrageobjektes am Pool erfragt und anschließend ausgeführt werden. *Custom Re-*

[1] TCP-APIs in Programmiersprachen wie Java und C# unterscheiden zwischen Thread-basiertem IO und Event-basiertem IO. Während das Thread-basierte IO einfach zu realisieren ist, erlaubt die Event-basierte Verarbeitung eine effiziente Verwaltung von hohen Verbindungszahlen [8, S. 18ff].

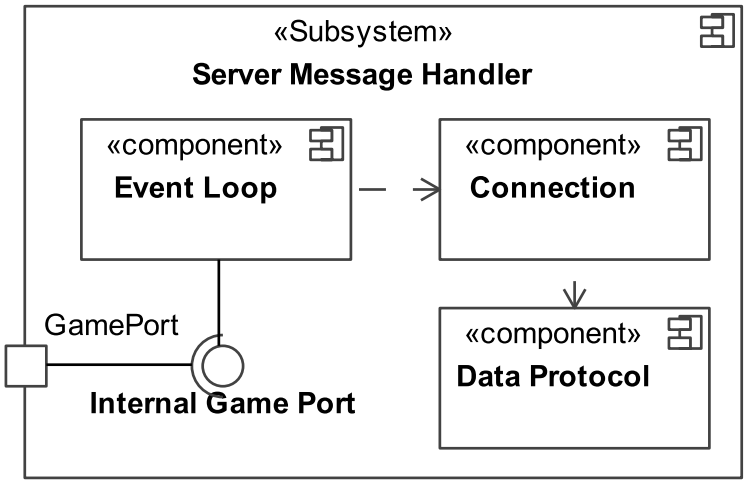



**Abbildung 5.9: Komponentendarstellung auf Level 2 der Architektur mit Fokus auf den *Server Message Handler*.**

*quest Handler*, welche eine Generalisierung der *Generative Request Handler* sind, werden von vordefinierten, speziellen Anfragen oder für Spezialfälle bei der Spielentwicklung genutzt. Diese speziellen Behandlungsroutinen werden ebenso im Fall der Verarbeitung des zugehörigen Anfrageobjektes – welches in diesem Fall nicht generischer Natur ist – am Pool erfragt und anschließend ausgeführt.

Der *Game Model Container* in Abbildung 5.11 dient zur Verwaltung von ein oder mehreren Instanzen von Spielmodellen. Dieser Container muss die Instanzen dauerhaft aktiv halten und kontinuierlich simulieren. Clients formulieren und transportieren über den *Server Message Handler* und *Request Handling* Anfragen, welche in Aufrufen auf den verwalteten Instanzen resultieren. Im Rahmen der Ausführung eines Spielmodells werden Komponenten bereitgestellt, die die verschiedenen Anforderungen erfüllen:

- **Id Generator**
  Stellt innerhalb des Containers eindeutige IDs zur Verfügung, wenn z. B. neue Entitäten erzeugt werden. Diese IDs sind insbesondere innerhalb einer verteilten Ausführung und über Neustarts hinweg eindeutig.

- **Persistence Manager**
  Dient zur Aktualisierung der Datenbank im Sinne eines vereinfachten ORM.

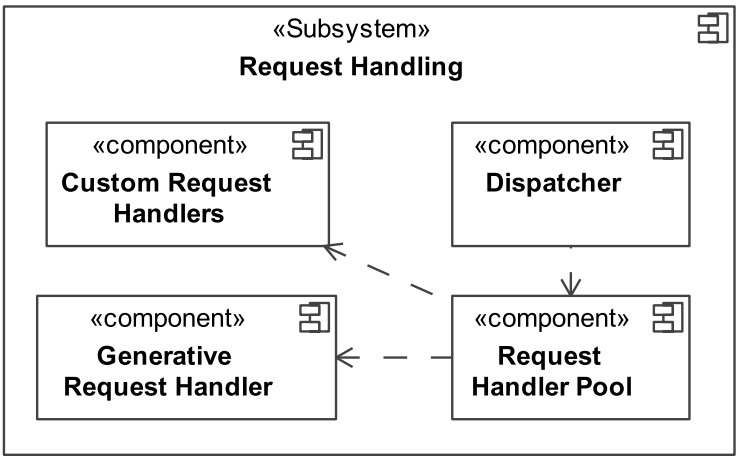



**Abbildung 5.10: Komponentendarstellung auf Level 2 der Architektur mit Fokus auf das *Request Handling*.**

- **Lifecycle Management**
  Verwaltet den Lebenszyklus von Entitäten des Spielmodells.

- **Timer**
  Steuert zeitliche Ereignisverarbeitung und stellt Verwaltungsschnittstellen zur Verfügung um Aktionen hinzuzufügen, zu ändern oder zu entfernen.

- **Invocation Handler**
  Wird von generischen und speziellen Bearbeitungsroutinen genutzt um das Modell aufzurufen.

Der *Client Message Handler* in Abbildung 5.11 ist dem *Server Message Handler* ähnlich. Dieses Subsystem verwendet die Komponenten *Event Loop*, *Data Protocol* und *Connection*. Die *Event Loop* verarbeitet auf Clientseite nach Verbindungsaufbau die ein- und ausgehenden Datenströme. Die Komponente *Connection* dient auf Clientseite zur Verwaltung einer Verbindung, während im Server mehrere verwaltet werden. Das *Data Protocol* übernimmt dieselben Aufgaben wie auf der Serverseite.

Das letzte im Detail betrachtete Subsystem ist der *Game Model Proxy*, dargestellt in Abbildung 5.13. Dieser Proxy dient als generische erzeugte Fassade in der jeweiligen Client-Sprache für den Zugriff auf Objektinstanzen des Servers. Innerhalb des Subsystems müssen Antworten in der Komponente *Respon-*

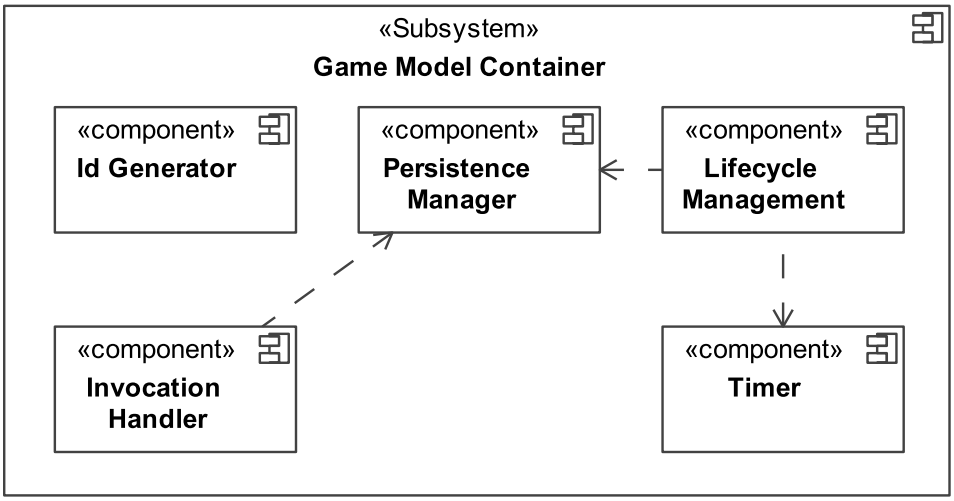



**Abbildung 5.11: Komponentendarstellung auf Level 2 der Architektur mit Fokus auf den *Game Model Container*.**

*se Handling* verarbeitet werden und Beobachter über dessen Eintreffen informiert werden, sowie Anfragen mit der Komponente *Request Handling* an den *Client Message Handler* weiter gegeben werden. Die öffentlich bereitgestellten Methoden in den Proxies implementieren die Anwendung der Komponente *Request Handling* und *Response Handling*.

## 5.7 Entwurfsentscheidungen

Im Rahmen der Dokumentation dieser Architektur für eine Ausführungsumgebung für Spielmodelle wurden verschiedene Entwurfsentscheidungen getroffen. Im Folgenden wird auf vier Entscheidungen, welche die Kommunikationsinfrastruktur der Ausführungsumgebung betreffen, im Detail eingegangen und diese nachvollziehbar dargestellt.

### 5.7.1 Schnittstelle je Instanz eines Spielmodells

Wie kommuniziert ein Client mit der Instanz eines Spielmodells? Zur Realisierung der Kommunikationsinfrastruktur kommt, neben der Entscheidung bezüglich Protokoll und Abbildungsformat, die Frage nach dem Kontext der technischen Schnittstelle auf: Spiegelt ein Kommunikationsport den Zugang zu einer Instanz des Spielmodells, einem Spiel oder einer Ausführungsumgebung

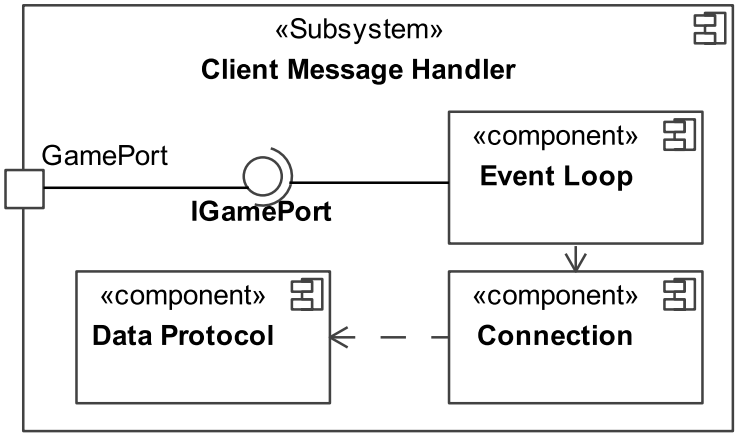



**Abbildung 5.12: Komponentendarstellung auf Level 2 der Architektur mit Fokus auf den *Client Message Handler*.**

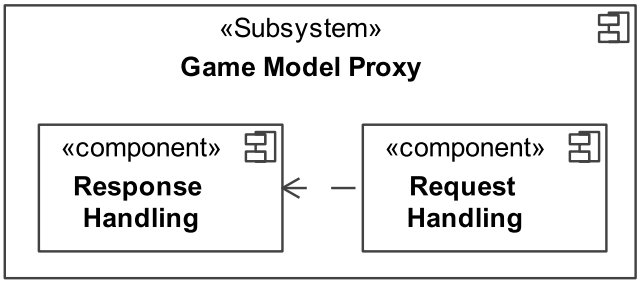



**Abbildung 5.13: Komponentendarstellung auf Level 2 der Architektur mit Fokus auf den *Game Model Proxy*.**

wieder?

Relevante Einflussfaktoren sind an dieser Stelle die geringen Einstiegshürden, die einen einfachen und intuitiven Umgang mit der Infrastruktur fordern, sowie die wartungsarme Ausführung und Erweiterbarkeit des Systems. Ergänzend ist die Randbedingung zur technischen Anbindbarkeit von verschiedenen Clienttechnologien von Bedeutung.

Angenommen wird, dass eine Ausführungsumgebung mehrere Spiele ausführen kann und ein Spiel in mehreren Spielwelt Instanzen betrieben werden kann. Es ist zusätzlich anzunehmen, dass bei Spielen mit hoher Nutzerzahl nur eine Instanz des Spielmodells je Ausführungsumgebung betrieben wird und alternative Instanzen in eigenständigen Installationen betrieben werden.

Zur Realisierung der technischen Schnittstelle können, wie in Abbildung 5.14 dargestellt, drei Alternativen betrachtet werden. Die erste Alternative

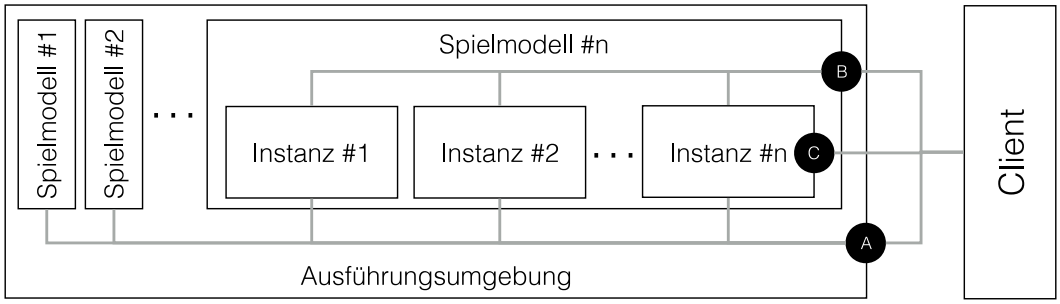



**Abbildung 5.14: Darstellung der Alternativen zur Realisierung des *GamePort*.**

(A) realisiert je Ausführungsumgebung eine technische Schnittstelle über die ein Client mit einer Spielwelt kommuniziert. Dies erfordert die Identifizierung des Spiels und der aktuellen Instanz, mit der der Spieler interagiert, im Kommunikationsprotokoll. Die zweite Alternative (B) betrifft die Realisierung einer technischen Schnittstelle je Spiel, dies ermöglicht eine implizite Identifikation des Spiels, mit dem der Client aktuell interagiert, erfordert jedoch weiterhin die Identifikation der Instanz. Die dritte Alternative (C) realisiert eine technische Schnittstelle je Instanz. In diesem Fall sind Spiel und Instanz implizit durch den Kommunikationskanal vorgegeben. Diese Alternative erfordert jedoch, dass der Client im Fall des Verbindungsaufbaus wissen muss, wie dieser zu adressieren ist.

Im Sinne eines intuitiven Umgang mit der Ausführungsumgebung wurde die Entscheidung für einen Kommunikationskanal je Instanz des Spielmodells getroffen. Damit ist bei einem Kommunikationsaufbau kein zusätzlicher Datenaustausch notwendig um die Identifikation des jeweiligen Spiels und der aktuellen Instanz notwendig. Zudem ist es möglich den Datenverkehr einfacher über gängige Strategien zur Lastverteilung zu koordinieren, ohne vorher den Datenverkehr zu analysieren.

### 5.7.2 Schnittstelle für Services

Neben der technischen Schnittstelle zur Interaktion mit Instanzen des Spielmodells stellt sich die Frage, wie Services zum Spiel bereitgestellt werden. Diese sind spielweltunabhängig und an Nutzerkonten gebunden. Entschieden werden muss, ob die Dienste über die technische Schnittstelle zu einer Instanz des

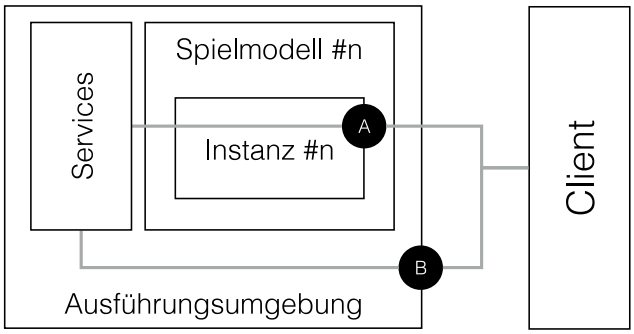



**Abbildung 5.15: Darstellung der Alternativen zur Realisierung des *ServicePort*.**

Spielmodells oder über eine eigene Schnittstelle betrieben werden.

Es kann angenommen werden, dass die Dienste nicht Teil der Ausführungsumgebung für Spielmodelle sind, jedoch im Rahmen eines Spiels bereitgestellt werden müssen. Hierfür wurde die Wahl zur Verwendung der Java EE getroffen, auf dessen Basis webbasierte Dienste realisiert werden können. Die Einbettung der Ausführungsumgebung in eine Java EE Landschaft ermöglicht die Anwendung bereits existierender Technologien und erfordert die Bereitstellung dieser über jene Standards.

Bei der Betrachtung dieser Entscheidung standen die zwei Alternativen in Abbildung 5.15 im Vordergrund. Die erste Alternative (A) bettet die Kommunikation zwischen Client und Dienst in den Kanal zum Aufruf von Funktionalität der Instanz ein. Die zweite Alternative (B) nutzt die Standards der Java EE Landschaft, z. B. EJBs, Servlets und JAX-RS.

Da die Ausführungsumgebung zur Bereitstellung von Spielmodellen dient und Dienste nicht automatisch aus diesem bereitgestellt werden, wurde in diesem Fall als Entscheidung festgehalten, dass die als z. B. JAX-RS- und EJB-basierten Dienste über jene vorgesehenen technischen Schnittstellen bereitgestellt werden, um die Komplexität der Ausführungsumgebung nicht unnötig zu steigern und die Aufgabe dieser auf das wesentliche fokussiert: der Bereitstellung der Kommunikationsinfrastruktur zur Interaktion mit einem Spielmodell.

### 5.7.3 Modifizierbare Datenverarbeitungspipeline

Eine Ausführungsumgebung für browserbasierte MMOGs muss Möglichkeiten bieten, um verschiedenen Clienttechnologien zur Verfügung zu stellen. Die

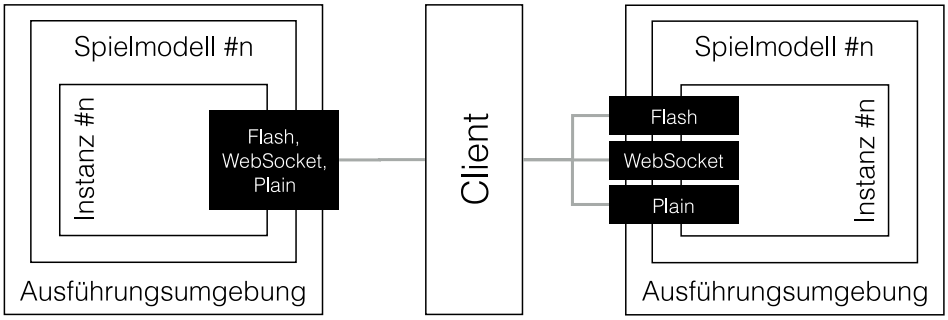



**Abbildung 5.16: Darstellung der Alternativen zur Realisierung des *GamePort* für verschiedene Clienttechnologien.**

Realisierung bedingt je Clienttechnologie ggf. zusätzliche Anpassungen an der technischen Schnittstelle. Wie diese speziellen Anpassungen je Clienttechnologien in der Ausführungsumgebung berücksichtigt werden, wird im Folgenden entschieden.

Als Einflussfaktoren sind in dieser Entscheidungsfrage die verschiedenen Clienttechnologien der Randbedingungen zu berücksichtigen. Dessen Bandbreite wurde in Kapitel 3.4 (S. 61) in verschiedenen verfügbaren Werkzeugen zur Unterstützung der MMOGs-Spielentwicklung analysiert. Zusätzlich sind Qualitätsziele wie geringe Einstiegshürden, Konvention vor Konfiguration und wartungsarme Ausführung zu berücksichtigen.

Bei dieser Betrachtung kann angenommen werden, dass in einem Spiel in den überwiegenden Fällen eine einzige Clienttechnologie zum Einsatz kommt. Es ist jedoch darüber hinaus denkbar, dass eine Kombination eingesetzt wird, um verschieden Funktionalitäten umzusetzen oder alternative Clients anzubieten, die in Plattformen ausgeführt werden, in denen nur eingeschränkte Wahlmöglichkeiten existieren.

Die Realisierung kann auf zwei Alternativen, siehe Abbildung 5.16, reduziert werden. In der ersten Alternative wird der *Message Handler* auf die Anwendung eines speziellen Protokolls spezialisiert. Dies vereinfacht die Komplexität des Message Handlers, ermöglicht eine effiziente Verarbeitung und reduziert Fehleranfälligkeit. Diese Alternative erfordert jedoch, dass je Spiel und Instanz zu konfigurieren ist, welche Art Clienttechnologie anbindbar ist und welche technischen Schnittstellen benötigt werden. In der zweiten Alternative ist die technische Schnittstelle multifunktional und erkennt eigenständig die Clients-

pezifika. Dies vereinfacht den Umgang mit der Schnittstelle, erhöht jedoch die Komplexität im *Message Handler*.

Die Entscheidung fiel zugunsten der multifunktionalen technischen Schnittstelle je Spielwelt. Sie deckt sich mit den Qualitätszielen zu geringen Einstiegshürden, Konvention vor Konfiguration und wartungsarmer Ausführung. Eine technische Schnittstelle symbolisiert im Sinne der Ausführungsumgebung den Zugang zu einer Spielwelt – unabhängig von der anfragenden Clienttechnologie.

### 5.7.4 Proprietäres Datenaustauschformat

Wie werden die Daten abgebildet? Für den Transport der Kommunikationsobjekte zwischen Client und Server ist ein Abbildungsformat notwendig und eine Entscheidung notwendig, wie dieses aussieht. Kapitel 3.1 (S. 31) zeigt XML und JSON als standardisierte Abbildungsformate sowie proprietäre Ansätze von MMOG-Werkzeugen. Für die intendierte Ausführungsumgebung muss entschieden werden, welche Abbildung für den Transport der Daten anzuwenden ist.

Als Einflussfaktoren spielen das breite Spektrum an Clienttechnologien sowie Qualitätsziele bezüglich den geringen Einstiegshürden und der Konvention vor Konfiguration eine besondere Rolle. Ergänzend muss hier die Erweiterbarkeit berücksichtigt werden, um spezielle Szenarien berücksichtigen zu können.

Im Vergleich zu Serviceschnittstellen, die meist einem unbekannten Anwender offen, interpretierbar und abwärts kompatibel zur Verfügung stehen müssen, kann im Fall von einem MMOG die Annahme getroffen werden, dass Client und Server parallel entwickelt und synchron aktualisiert werden. In diesem Szenario steht insbesondere die Performance im Vordergrund – je schneller eingehende Anfragen bearbeitet werden, desto besser.

Basierend auf den Betrachtungen in Kapitel 3.1 (S. 31) ergeben sich zwei Alternativen, die in Abbildung 5.17 dargestellt sind: ein standardisiertes oder ein proprietäres eigenes Abbildungsformat.

Unter Berücksichtigung der Annahme, dass Client und Server synchron entwickelt und aktualisiert werden, wurde die Entscheidung zugunsten eines eigenen proprietären Abbildungsformates getroffen. Diese Entscheidung bedingt jedoch, dass das Abbildungsformat austauschbar sein muss, damit spezielle Ein-

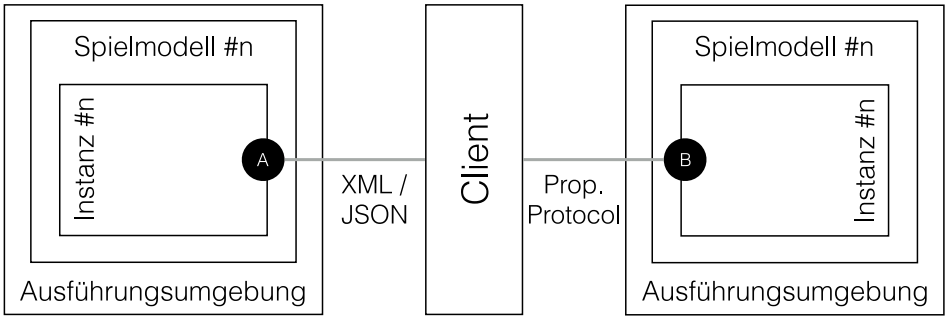



**Abbildung 5.17: Darstellung der Alternativen zur Realisierung eines *Data Protocol*.**

satzszenarien nicht ausgeschlossen werden. Ein eigenes Abbildungsformat im Rahmen der geplanten generierten Kommunikationsobjekte kann effizient eingesetzt werden. Aufgrund der Synchronität zwischen Client und Server ist der Transport von Strukturdaten, wie Attributnamen oder Typisierungen, nicht notwendig. Diese könnten bereits bei Veröffentlichung einer neuen Version in der Anwendung verankert werden und müssen nicht während des Datenaustauschs transportiert werden.

# 6. Ausführungsumgebung

Für die Realisierung der Ausführungsumgebung, deren Architektur in Kapitel 5 (S. 93) dargestellt wird, werden im Folgenden drei Aufgabenbereiche im Detail aufgegriffen. Der *Game Model Container* (siehe Kapitel 6.1, S. 123) beschreibt, wie Spiele simuliert und persistiert werden, sowie Ereignisse propagiert werden. Die Kommunikationsinfrastruktur (siehe Kapitel 6.2, S. 134) realisiert den *Server Message Handler* und das *Request Handling*. Anschließend folgt die Realisierung im Client (siehe Kapitel 6.3, S. 155), welcher auf das Gegenstück der Kommunikationsinfrastruktur eingeht, sowie die Ableitung von Kommunikations- und Modellcode in die verwendete Clientsprache darstellt. Abgeschlossen wird die Darstellung der Ausführungsumgebung mit der Realisierung des Game Meta Models (siehe Kapitel 6.4, S. 167), welche die Entwicklung unterstützt und vereinfacht, sowie eine Realisierung des Beispiels aus Kapitel 2.9 (S. 28) unter Verwendung der realisierten Ausführungsumgebung.

## 6.1 Game Model Container

Der *Game Model Container* ist das Herz zur Ausführung von Spielmodellen. Dessen Aufgabe ist es, eine oder mehreren Instanzen von Spielmodellen zu verwalten. Teil dieses Subsystems zur Verwaltung, dargestellt in Abbildung 6.1, sind *Lifecycle Mangement*, *Persistence Manager*, *Id Generator*, *Timer* und *Invocation Handler* (siehe Kapitel 5.6, S. 110).

Repräsentiert wird dieses Herz durch die Klasse *GameController*, dargestellt in Abbildung 6.2. Der *GameController* ist als MBean realisiert und implementiert die durch die Spezifikation geforderte Schnittstelle *GameControllerMBean* (siehe Kapitel 3.3.1, S. 56), welche denselben Namen zzgl. „MBean“ tragen muss und die öffentlich verfügbaren Methoden der Service-Komponente inner-

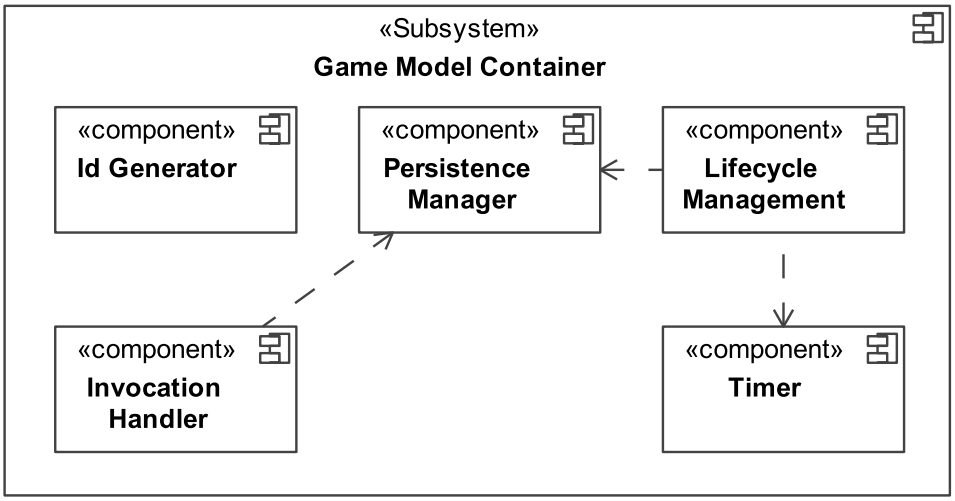



**Abbildung 6.1: Komponentenübersicht des *Game Model Container* (siehe Kapitel 5.6, S. 110).**

halb einer Java EE Umgebung definiert. Darüber hinaus steuert und verwaltet der Controller die ihm zur Verwaltung übergebenen Spielwelten und korrespondierenden *Server Message Handler* (siehe Kapitel 6.2, S. 134).

Damit ein Spielmodell in diesem Container ausgeführt werden kann, muss eine Entität des Modells als Ausgangselement markiert werden. Im Sinne eines Spiels und dessen Definition (siehe Kapitel 2.1, S. 7) ist dies die Spielwelt in der die zu verwaltenden Spielobjekte agieren. Die Markierung dieser Entität wird mithilfe der Annotation *@GameWorld* am Entitätstyp oder über eine externe Konfiguration durchgeführt. Bei der Markierung wird dem Entitätstyp eine eindeutige Spielweltkennung und der zu verwendende Kommunikationsport[1] zugewiesen.

Im Gegensatz zur Bereitstellung von Geschäftslogik mittels EJBs wird in der Ausführungsumgebung eine Instanz des Spielmodells mehreren Nutzern zur Verfügung gestellt. Die Ausführungsumgebung koordiniert in diesem Anwendungsrahmen den Zugriff und organisiert die Persistenz des Modellzustandes. Clients, welche mit dem Spielmodell interagieren wollen, formulieren über die Kommunikationsinfrastruktur – insbesondere den *Server Message Handler* und *Request Handling* – Anfragen und der Game Container nimmt die resultierenden Aufrufe über die Komponente *Invocation Handler* entgegen, delegiert diese und persistiert notwendige Änderungen.

[1] Im Sinne der Adressierung von Anwendungen im TCP- und UDP-Protokoll.

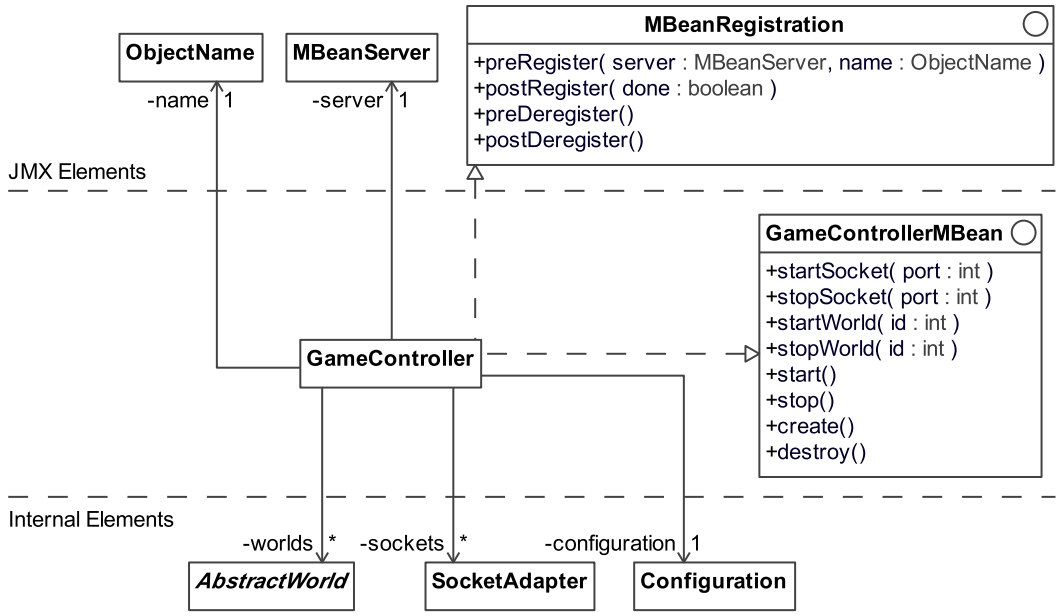



**Abbildung 6.2: Klassendiagramm der MBean *GameController* innerhalb der Komponente *Game Model Container*.**

Für einen sicheren und koordinierten Zugriff auf eine Entität des Spielmodells ist die direkte Interaktion mit den Entitäten zu vermeiden. Der *Game Model Container* macht zur Vermeidung dessen Gebrauch vom Proxy-Pattern [19, S. 263]. Ziel ist es, das der Rückgabewert von Methoden zum Erzeugen und Laden von Entitäten im *Game Model Container* immer ein Proxy auf die im Container verwalteten Entitäten ist. Mithilfe dieser Zwischenschicht ist es möglich, Aufrufe auf die Entitäten um zusätzlichen Instruktionen zu erweitern, sowie Änderungen an Clients und Datenbank zu verteilen. Ergänzend ermöglicht die Entkopplung von Zeigern und Entitäten die Möglichkeit einer freien Positionierung jener in einem Rechencluster. Methoden könnten über Systemgrenzen hinweg aufgerufen werden und die Verarbeitungslogik kann unabhängig von der physischen Positionierung der Entitäten ausgeführt werden. Eine Herausforderung ist jedoch die Realisierung in Java, wenn Proxies nicht von Schnittstellen, sondern von Klassen erzeugt werden müssen. Die Anwendung des Proxy-Patterns auf Java Klassen wird im folgenden Kapitel 6.1.1 (S. 126) gezeigt.

Dieses Vorgehen kann auf die Interaktion zwischen Entitäten innerhalb eines ausgeführten Spielmodells erweitert werden, erfordert jedoch ein spezielles

Vorgehen bei der Entwicklung selbiger. Der Vorteil ist, dass Skalierungs- und Persistierungsvorgänge bei Aufrufen zwischen den Entitäten des Spielmodells greifen. Der Nachteil ist, dass im Sinn von Java der Pointer *this* und der Konstruktor Aufruf mit *new* für neue Entitäten, die durch den Container verwaltet werden sollen, nicht verwendet werden darf.

### 6.1.1 Anwendung des Proxy-Patterns auf Java Klassen

In Java werden Proxies über den Aufruf der statischen Methode *newProxyInstance* an der Klasse *Proxy* erzeugt. Diese nimmt als Parameter eine Reihung von Schnittstellen, sowie eine zu instanziierenden Implementierung der Schnittstelle *InvocationHandler*. Das Resultat ist eine Instanz, welche die als Reihung übergebenen Schnittstellen implementiert und die Aufrufe an jener Proxy-Instanz über die Methode *invoke* der bereitgestellten Implementierung der Schnittstelle *InvocationHandler* leitet. Auf diese Weise kann das Verhalten bei Aufrufen von Methoden der implementierten Schnittstellen des Proxies beeinflusst werden. Dies ermöglicht, dass zum Beispiel zusätzliche Instruktionen je Aufruf, wie Logging oder Anweisungen zur Persistierung, ergänzt werden können.

Java bietet bis Version 7 keine Möglichkeit dies ebenso mit Klassen durchzuführen. Begründen lässt sich dies in schwierigen Spezialfällen, insbesondere im Fall des Zugriffes auf Attribute. Dessen Aufruf lässt sich nicht über eine spezielle Behandlungsmethode kanalisieren. Damit Aufrufe aus Anfragebehandlungen auf das Spielmodell jedoch durch den *Game Model Container* erfasst und beeinflusst werden können wurde mithilfe einer zusätzlichen Bibliothek gearbeitet, die das Proxy-Pattern über einen Trick nachahmt.

Der Trick setzt an der Funktionsweise von Klassenladern (engl. Class Loader) in Java, zum Bezug von Klassendefinitionen, an. Dieser erlaubt beliebige Datenquellen, welche physisch oder virtuell sein können, insofern Grundregeln zur Sicherheit der Java VM eingehalten werden. Verwendet man als Datenquelle eine generische Komponente – z. B. eine Quelle die zur Laufzeit Java Code erzeugt, in ausführbaren Java Bytecode übersetzt und das Ergebnis über einen Klassenlader in der Java VM bereitstellt – kann man durch Interpretation der verfügbaren öffentlichen Methoden von beliebigen Klassen ein generische Fassade erzeugen die dem Proxy-Pattern ähnelt. Damit lässt sich die Mög-

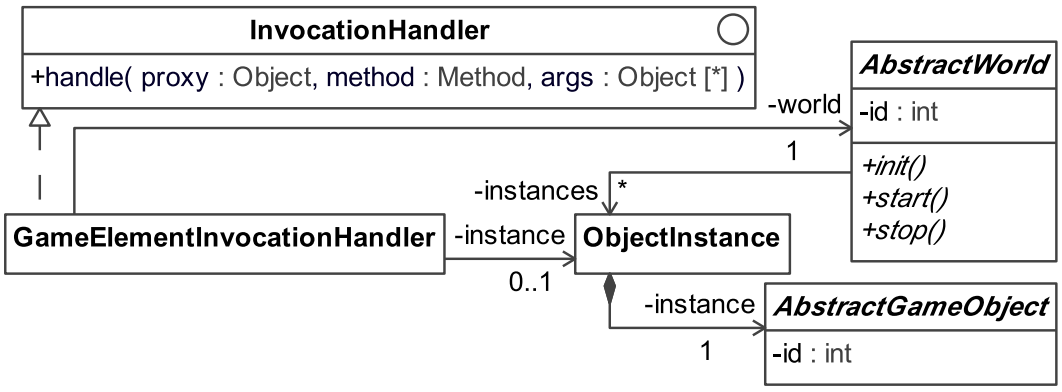



**Abbildung 6.3: Klassenübersicht *Invocation Handler*.**

lichkeit schaffen zwischen der Methodenausführung und dem Methodenaufruf zusätzliche Instruktionen zu injizieren. Dieser Ansatz funktioniert zuverlässig, erfordert jedoch spezielle Aufmerksamkeit. Diese Fassade kann durch die Spezialisierung Elemente enthalten, welche nicht überschrieben werden konnten, bzw. deren Aufruf nicht an die reale Instanz dirigiert werden kann. Dies betrifft z. B. Methoden die mit *final* markiert sind, sowie öffentliche Attribute, deren Modifikation keinen Einfluss hätte.

Die Konstruktion eines eigenen Java-Quellcode-Interpreters ist nicht notwendig. Zur Interpration von Quellcode und Bezug des Ergebnisses über einen Klassenlader kann das Werkzeug Javassist [22] verwendet werden. Dieses erlaubt die Verarbeitung einer beliebigen Zeichenkette, welche ausführbaren Java Quellcode wiederspiegelt und ermöglicht dessen Überführung in Java Bytecode um die repräsentierte Klasse anschließend über einen speziellen Klassenlader von Javassist zu beziehen und zu instanziieren.

Der Ausführungsumgebung definiert zur Bereitstellung von Proxies die Klasse *ProxyFactory*. Diese bietet statische Methoden zum Erzeugen von neuen Proxy-Instanzen, auf Basis von Klassen und Schnittstellen, sowie eine Methode zum Evaluieren, ob eine eine Instanz eine Proxy-Instanz ist. Die Evaluierungsmethode ist notwendig, um zu prüfen, ob ein Rückgabewert bereits ein Proxy ist.

### 6.1.2 Invocation Handler

Der *Invocation Handler* im *Game Model Container* dient der Bearbeitung von Anfragen an das Spielmodell, welche über einen Proxy – in der Regel immer dann, wenn Anfragen von außerhalb des Spielmodells entstehen – gestellt werden. Die Realisierung der Komponente *Invocation Handler*, im Klassendiagramm in Abbildung 6.3 ist dies die Klasse *GameElementInvocationHandler*, ist eine Implementierung der Java-Schnittstelle *java.lang.reflect.InvocationHandler*. Die durch die Schnittstelle definierte Methode *handle* dient zur Verarbeitung des Aufrufs. Die Identifikation, welche Methode aufgerufen wurde, sowie an welcher Proxy-Instanz (für den Fall, dass mehrere Instanzen mit einem Handler bearbeitet werden) und die Parameter für den Aufruf werden der Methode *handle* übergeben. Die implementierte Methode *handle* vom *GameElementInvocationHandler* arbeitet folgende Schritte ab:

1. **Validierung**
   Prüfen ob der Aufruf möglich ist. In erster Linie, ob die Instanz noch existiert. Anstatt mit schwachen Beziehungen zu arbeiten (vgl. Weak References), wird hier ein Instanzkapsel verwendet, welche den Zeiger auf die eigentliche Instanz enthält. Die Verwaltung der Kapsel obliegt dem Container. Wird eine Instanz zerstört, kann er die Kapsel leeren und damit alle für ihn unbekannten Proxies deaktivieren, da diese einen Zeiger auf die selbe Kapsel besitzen und fortfolgend mit einem geleerten Zustand agieren.

2. **Aufruf**
   Aufrufen der eigentlichen Funktionalität.

3. **Persistierung**
   Anstoßen der Persistierungsprozesse im Fall von Änderung und Abhängig von der Konfiguration (siehe Kapitel 6.1.3, S. 129).

4. **Verbreitung**
   Optionale Bekanntmachung des Aufrufs für andere Komponenten.

Abgeschlossen werden die vier Schritte mit der Rückgabe des Aufrufergebnisses an den Aufrufenden, was in der Regel eine Anfrage über die Kommunikationsinfrastruktur ist.

### 6.1.3 Persistenz des Spielmodells

Die Persistenz des Spielmodells realisiert der *Game Model Container* mithilfe der JPA Spezifikation der Java EE. Im Gegensatz zur Persistenz bei EJBs ist in diesem Anwendungsfall der dauerhaft aktive Container nicht direkt in Transaktionen unterteilbar und Instanzen müssen nicht je Anfrage bezogen werden. Instanzen existieren nach ihrer Initialisierung solange wie sie für die Simulation von Bedeutung sind und müssen innerhalb des Ausführungscontextes erhalten werden. Darüber hinaus ist davon auszugehen, dass diese einmalig existieren und Modifikationen an ihnen kontrolliert durchgeführt werden und nicht zu Konflikten führen.

Zur Realisierung der Persistenz arbeitet die Ausführungsumgebung mit den Werkzeugen der JPA Spezifikation, wie z. B. dem Laden, Speichern, Löschen und Suchen, jedoch nicht mit dem Transaktionsmodell. Wurden Modifikationen an einer Instanz erkannt, werden die Änderungen an die Datenbank weitergetragen.

Die Ausführungsumgebung ermöglicht indirekt das Erzeugen, Zerstören und Laden von Entitäten. Wird eine neue Entität angefordert und ist diese zu persistieren, wird ein entsprechender Befehl zur Erzeugung auf Datenbankebene ausgelöst. Wird eine persistent verwaltete Entität aus der Simulation entfernt, wird der entsprechende Befehl zum Löschen, bzw. deaktivieren, in der Datenbank ausgelöst. Da innerhalb des Containers der Zugriff auf Instanzen ausschließlich über generisch erzeugte Proxies möglich ist, kann nach jedem Methodenaufruf eine Prüfung auf Änderungen durchgeführt werden.

Damit ein Entität persistiert wird, muss der entsprechende Entitätstyp mit der aus der JPA Spezifikation stammenden Annotation *@Entity* markiert werden. Insofern diese Markierung existiert erkennt die Ausführungsumgebung die notwendige Persistenz dieser Entität und führt nach jedem Methodenaufruf einen Abgleich mit der Datenbank durch.

Neben dieser automatisierten Persistenz, welche in den meisten Fällen ausreichend ist, realisiert die Ausführungsumgebung zusätzliche Werkzeuge welche

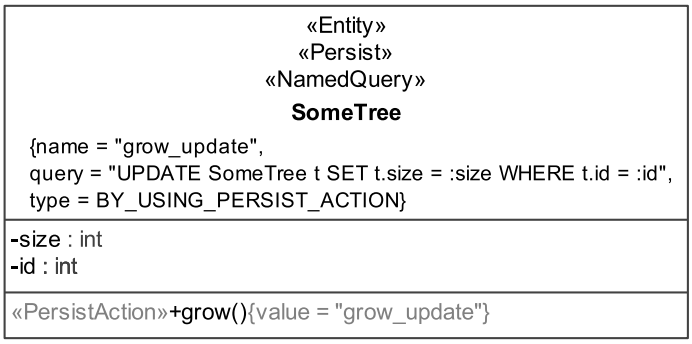



**Abbildung 6.4: Persistenzkonfiguration eines einfachen Spielentitätstyps. Eine Implementierung ist in Anhang J (S. 253) in Quellcode J.1 zu finden.**

genutzt werden können um zu konfigurieren, was nach dem Methodenaufruf in einer Datenbank aktualisiert werden soll.

Gesteuert wird die Konfiguration zur Persistierung über die Annotation *@Persist*. Diese Annotation erlaubt die Definition des für diese Entität anzunehmenden Verhaltens bei der Persistierung. Neben nie (*NEVER*) und nach jedem Aufruf (*AFTER_METHOD_CALL*) können Datenbankprozesse nur dann ausgeführt werden wenn eine spezielle Annotation platziert wurde (*BY_USING_PERSIST_ACTION*) oder einer Mischung aus „nach jedem Aufruf“ und „spezielle Annotation“ (*BY_USING_PERSIST_ACTION_OR_AFTER_METHOD_CALL*). In dem Fall hat die Annotation Vorrang.

Diese speziellen Annotation, die zur Markierung von Methoden die Datenbankaktualisierungen genutzt werden, heißen *@PersistAction* und *@PersistActions*. *@PersistActions* ist eine Sammlung von *@PersistAction* Annotationen. Die *@PersistAction* dient dem Mapping von vordefinierten benannten Datenbankstatements, welche mittels JPA eigenen *@NamedQuery*-Annotation definiert werden können. Abbildung 6.4 zeigt ein Beispiel in dem der Entitätstyp *SomeTree* eine Methode anbietet die im Fall des Aufrufs eine Datenbankaktualisierung auslöst. Mithilfe der Persist Annotation wurde für den Entitätstyp *SomeTree* definiert, dass nur Methoden die eine *@PersistAction* erhalten haben, eine Aktualisierung auf Datenbankebene auszuführen ist.

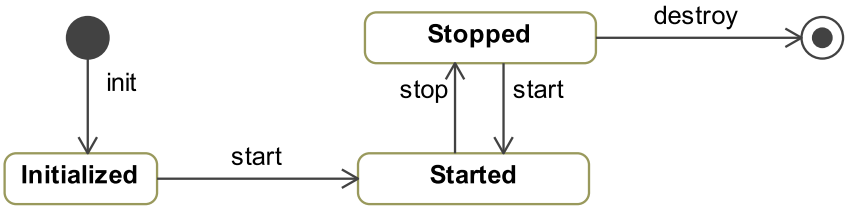



**Abbildung 6.5: Lebenszyklus von Spielobjekten.**

### 6.1.4 Simulation des Modells

Zuzüglich zur Verwaltung des Zugriffs und der Persistenz des Spielmodells übernimmt der *Game Model Container* Aufgaben zur Unterstützung der Simulation des Spielmodells. Darunter Fallen Werkzeuge zur Verwaltung von Zeit und Lebenszyklen von Spielwelt und Spielobjekten.

Der Lebenszyklus der Spielwelt, der Spielobjekte und der Aktionen innerhalb der Ausführungsumgebung basiert auf vier Zuständen, welche in Abbildung 6.5 dargestellt sind: (1) den initialisierten Zutand, indem eine Instanz vorbereitet wird und bereit zum Starten sein soll, (2) den gestarteten Zustand, in dem eine Instanz die Arbeiten beginnt, (3) den gestoppten Zustand, in dem eine Instanz die Arbeiten pausiert und (4) den finalen Zustand, in dem die Instanz notwendige Aufräumarbeiten vornimmt. Gewechselt werden die Zustände durch die Ausführungsumgebung und nicht durch den Entwickler. Der Entwickler kann durch die Implementierung der Methoden *init*, *start*, *stop* und *destroy* entscheiden, was im Fall des Zustandswechsels passieren soll.

Der *Timer* dient zum Verarbeiten von Aktionen in einem zeitlichen Kontext. Dabei stellt ein *Timer* sicher, dass die ihm zugeordneten Aktionen in korrekter Reihenfolge und möglichst zum geplanten Zeitpunkt bearbeitet werden. Der exakte Verarbeitungszeitpunkt wird dabei nicht garantiert, liegt aber zeitnah nach dem definierten Verarbeitungszeitpunkt.

Die in Abbildung 6.6 gezeigte Schnittstelle des Timers bietet zum platzieren von Aktionen vier Varianten: (1) Aktionen die zeitnah verarbeitet werden, (2) Aktionen die einmalig nach einer bestimmten Zeit verarbeitet werden, (3) Aktionen die x-mal nach einer jeweiligen bestimmten Zeit verarbeitet werden und (4) Aktionen die beliebig oft nach einer jeweiligen bestimmten Zeit verarbeitet werden. Die Verarbeitung dieser Aktionen wird in Abbildung 6.7 dargestellt. Erster Schritt der Ausführung der dargestellten *run* Methode ist die Ermitt-

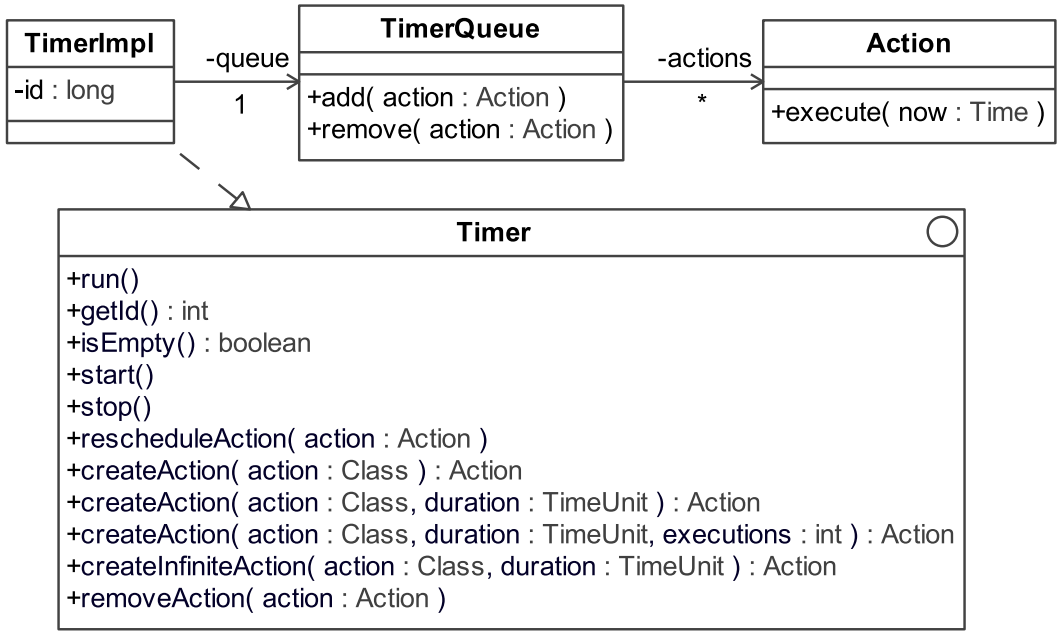



**Abbildung 6.6: Klassenübersicht der Komponente *Timer*.**

lung anstehender Aktionen. Zur Ermittlung dieser dient die *TimerQueue*. Diese erbt von der Java-Schnittstelle *Iterable*, womit ein Instanz, welche die Schnittstelle *Iterator* implementiert, zur Iteration über alle anstehenden Aufgaben erzeugbar ist. Mithilfe des erzeugten Iterators werden im Fall vom Aufruf der *run* Methode im Timer (siehe Abbildung 6.7) alle Aktionen ausgewählt und anschließend sequenziell, über den Aufruf der Methode *execute*, bearbeitet. Abgeschlossen wird mit der Entscheidung was anschließend mit dieser Aktion passiert. Je nach dem ob sie speziell, einmal, x-mal oder unbegrenzt ausgeführt werden soll, wird sie diese erneut in die TimerQueue einsortiert oder entsorgt. Dabei hat die Aktion selbst Möglichkeiten diese Entscheidung zu beeinflussen um sich selbst zu entfernen oder erneut zu einem späteren Zeitpunkt ausführen zu lassen. Die vordefinierten *createAction*-Signaturen in der Abbildung 6.6 helfen hier in erster Linie zur Erzeugung von Aktionen in häufig anzutreffenden Entwicklungssituationen.

Für die Erzeugung von Entitäten des Spielmodells ist innerhalb einer verteilten Architektur das Erstellen von eindeutigen Identifikatoren wichtig. Im Fall von datenbankzentrischen Anwendungen werden diese meist durch die Datenbank selbst vorgegeben. Im Fall der Ausführungsumgebung dient die Datenbank der nachrangigen Persistenz und Entitäten werden zeitlich verzögert

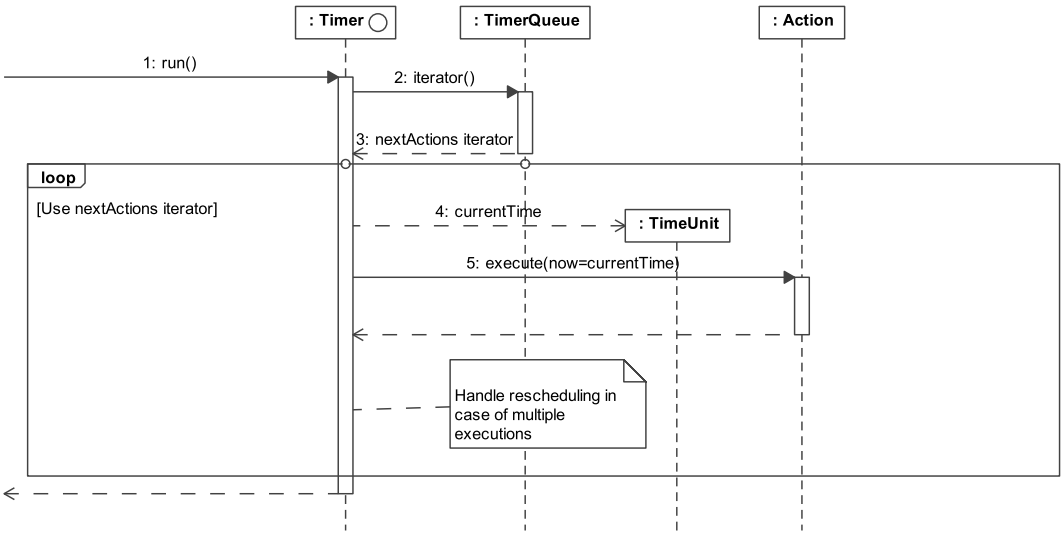



**Abbildung 6.7: Verarbeitung von Aktionen im *Timer*.**

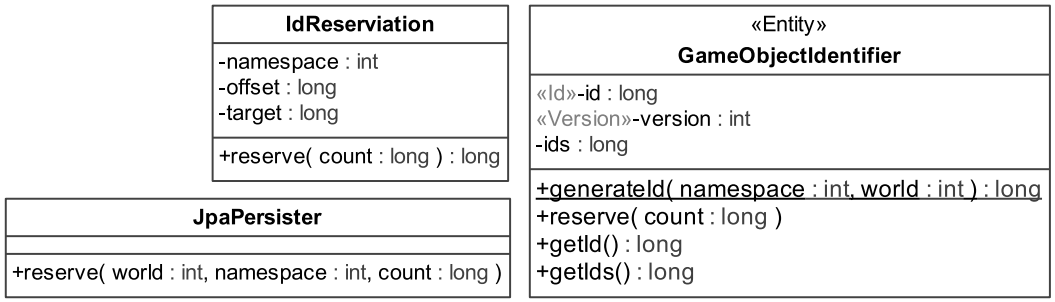



**Abbildung 6.8: Klassendiagramm zur Realisierung der Komponente *Id Generator*.**

in die Datenbank überführt. Für die eindeutige Kennung innerhalb der Ausführungsumgebung ist daher ein Service notwendig der Kennungen bereitstellt, die innerhalb der Systemgrenzen eindeutig sind.

Die Ausführungsumgebung verwaltet zur Vergabe von eindeutigen Identifikatoren für jeden Entitätstyp reservierte Identifikatoren-Kontingente und nutzt dafür die in Abbildung 6.8 dargestellten Klassen. Verwaltet werden die Kontingente über die Klasse *JpaPersister*, welcher innerhalb der Ausführungsumgebung als EJB realisiert ist. Immer wenn ein Identifikator für eine neue Entität benötigt wird, prüft die Ausführungsumgebung ob sie hierfür bereits Kontingente reserviert hat. Existiert ein reserviertes Kontingent und hat dieser verbleibende Identifikatoren, so wird von diesem ein Identifikator verwendet. Existiert dieser nicht oder ist das Kontingente erschöpft, so wird über

die Persistierungsmechanismen eine speziell verwaltete Entität vom Typ *GameObjectIdentifier* in der Datenbank aktualisiert. Diese Entität enthält einen Klassenidentifikator und eine eindeutige Spielweltkennung, sowie eine Versionsnummer und die bereits konsumierten Identifikatoren. Im Fall der Reservierung neuer Kontingente wird die jeweilige Entität vom Typ *GameObjectIdentifier* bezogen oder erstellt, der Identifikator inkrementiert, die Versionsnummer geändert und persistiert. Die Persistierung schlägt fehl, wenn Unstimmigkeiten mit der Versionsnummer auftreten. Diese Unstimmigkeiten können passieren, wenn parallele Zugriffe durch mehrere Instanzen einer Ausführungsumgebungen die Teile der selbe Spielwelt simulieren, aufeinander treffen. In diesem Fall kann der Vorgang wiederholt werden und ein neues Identifikator-Kontingent reserviert werden. Ist die Reservierung erfolgreich, kann ein neuer Identifikator aus dem resultierenden Kontingent gewählt und an die neue Instanz vergeben werden.

## 6.2 Kommunikationsinfrastruktur

Zur Interaktion mit dem *Game Model Container* ist eine Kommunikationsinfrastruktur auf der Serverseite notwendig. Diese wird mithilfe der Subsysteme *Server Message Handler* und *Request Handling* (siehe Abbildung 6.9), sowie der Komponente *Sessions* realisiert. Während die Komponente *Sessions* eine einfache Verwaltung von aktiven Verbindungen, sowie den damit im Zusammenhang stehenden Daten darstellt und hier nicht genauer betrachtet wird, wird im Folgenden detailiert die Funktionsweise des Datenaustausches dargestellt.

### 6.2.1 Nachrichtenverarbeitung

Als Basis für die Kommunikation zwischen Client und Server kommt die TCP- und UDP-basierten APIs in Java zum Einsatz. Hierfür arbeitet die Kommunikationsinfrastruktur der Ausführungsumgebung auf Basis eines Konstruktes, mit dem die Verwaltung von Verbindungen austauschbar ist. Im aktuellen Entwicklungsstand werden TCP-basierten Verbindungen auf Basis des Java NIO Socketservers unterstützt. Die Grundgedanken des Socketservers basieren auf den Ergebnissen der Ausarbeitung „Browserbasierte Echtzeit Simulation ‚Evo-

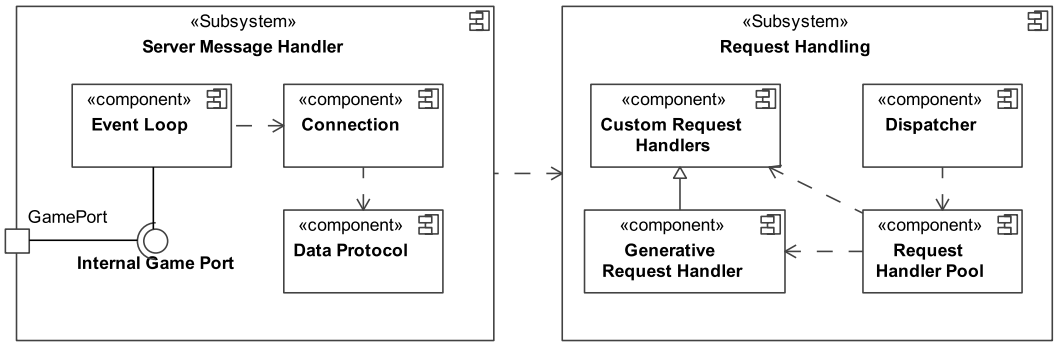



**Abbildung 6.9: Komponentenübersicht der Kommunikationsinfrastruktur.**

lution“ [8]. Die Implementierung wurde aufgriffen und als dedizierter JMX Service in einer Java EE Umgebung realisiert und erweitert. Abbildung 6.10 zeigt den Aufbau der Socketservers in einem Klassendiagramm.

Der Kern des Subsystems zur Nachrichtenverarbeitung ist der *SocketAdapter*, welcher als JMX MBean zur Verfügung steht. Dieser steuert den Lebenszyklus der untergeordneten Komponenten und stellt für diese, Repräsentiert durch die Implementierung der Schnittstelle *SocketController*, Funktionalitäten bereit. Er erzeugt, konfiguriert, startet und stoppt die Implementierung der Schnittstelle *SocketServer* und definiert eine Menge von Detektoren zur Behandlung von Transport- und Kommunikationsprotokollen zwischen dem Abbildungsformat der Ausführungsumgebung und dem TCP- oder UDP-Protokoll. Die Detektoren fungieren im Sinne einer Verarbeitungskette auf Basis des Pipes and Filters Patterns [19, S. 53]. Diese Verarbeitungskette ist notwendig, da zum Beispiel im Falle von Flash spezielle initiale Sicherheitsabfragen auf XML Basis zu beantworten sind oder im Falle von WebSockets ein Handshake-Prozess zu behandeln und im folgenden WebSocket-Dataframes zu Endpacken und vor dem Versand wieder zu Verpacken sind (siehe Kapitel 5.5.2, S. 106).

Das Detektionsverfahren arbeitet mit der Annahme, dass Datenpakete innerhalb der ersten vier Byte von möglichen anzutreffenden Alternativen abgegrenzt werden können. Im Fall von Flash startet das initiale Datenpaket mit `<policy-file-request/>`, im Fall eines WebSockets oder REST-Aufruf startet das initiale Datenpaket mit `GET`, `POST`, `PUT` oder `DELETE`. Aus diesen HTTP-basierten Header lässt sich im Folgenden ableiten, ob es sich dabei um

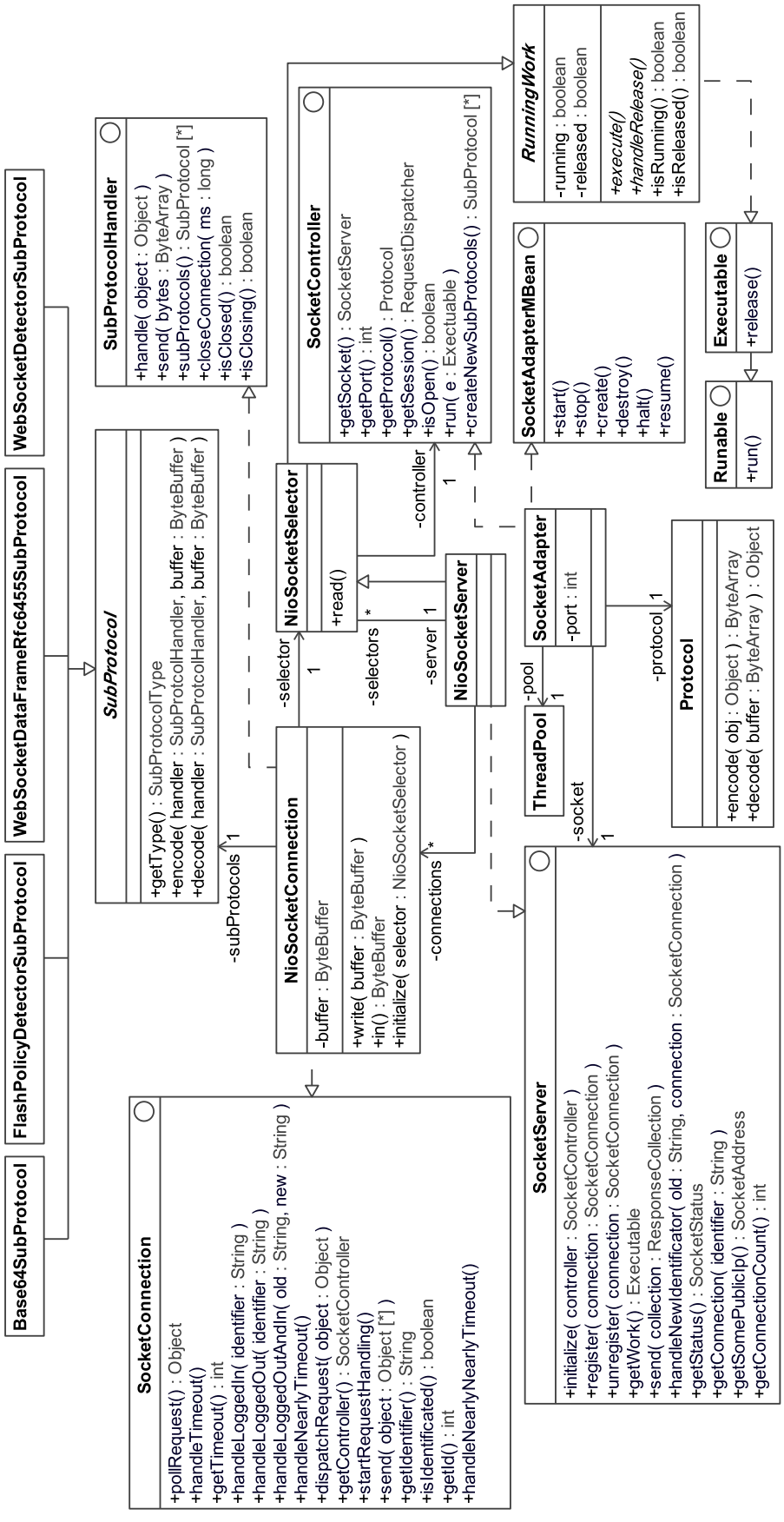



**Abbildung 6.10: Klassendiagramm zur Nachrichtenverarbeitung.**

ein REST-Aufruf oder um eine Initiierung eines WebSockets handelt.

Da diese Protokollerweiterungen initial, nach dem Aufbau der Verbindung zu erwarten sind, können die Detektoren in Folge dessen bei Abweichungen entfernt werden um die Effektivität der Datenverarbeitung nicht zu beeinträchtigen. Im Falle einer Detektion können notwendige Antworten, zum Beispiel ein Policy-Response als Antwort für das initiale Datenpaket in Flash, an den Client gesendet werden und die Datenverarbeitungskette auf neue Anforderungen angepasst werden. Dies ist zum Beispiel bei WebSockets notwendig, da die Datenpakete mit einem einfachen Maskierungsverfahren in einem speziellen Dataframe übertragen werden.

Die Detektoren, im folgenden Subprotokoll genannt, sind in einer verketteten Liste organisiert und werden beginnend beim ersten Element bearbeitet. Dem Subprotokoll wird ein Datenpaket mit einem Marker für den Offset übergeben und soll, ab der markierten Position, analysiert oder modifiziert werden. Als Rückgabewert wird ein Datenpaket erwartet, dass dem Eingabewert entsprechen kann, jedoch nicht muss. Sollte das Datenpaket keine Folgedaten enthalten, wird die Verarbeitung beendet. Sollte für die Analyse eine unzureichende Menge an Daten verfügbar sein, kann das Subprotokoll eine *BufferUnderflowException* auslösen um die Verarbeitung abzubrechen und zu einem späteren Zeitpunkt, wenn weitere Daten verfügbar sind, wieder aufzunehmen. Stellt das Subprotokoll während der Verarbeitung fest, dass er für die Analyse nicht zuständig ist, soll es die für die Analyse gelesenen Daten als ungelesen markieren und sich selbst aus der verketteten Liste entfernen.

Der *SocketAdapter* erzeugt neben der Liste von Subprotokollen eine Instanz des Abbildungsformates zur Serialisierung und Deserialisiserung von Kommunikationsobjekten. Jedem *SocketAdapter* wird eine feste Menge an Klassen mitgeteilt, die er empfangen und versenden darf, welche sich aus den transportierbaren Entitäten des Spielmodells, den abgeleiteten Kommunikationsobjekten, den manuell definierten Kommunikationsobjekten und den durch die Ausführungsumgebung vorgegebenen Kommunikationsobjekten ergibt.

Für die Behandlung der Clientverbindungen und die Verarbeitungen des Datenverkehrs wird eine Socketserver Implementierung verwendet, die vom *SocketAdapter* unabhängig ist: der *NioSocketServer*. Dieser verwendet eine Implementierung auf Basis des Java NIO Socketservers. Diese Ereignis-basierte

Verarbeitung von Socketverbindungen löst beim Empfangen von Daten und bei Initiierung neuer Verbindungen Ereignisse aus. Eine Ereignisschleife, die in einem eigenen Prozess ausgeführt wird, behandelt alle anstehenden Ereignisse über die Methode *execute*. Will der Server Daten an verbundene Clients versenden, löst er ein Ereignis aus, dass die Ereignisschleife veranlasst Daten zu versenden. Ereignis-basierte Socketverbindungen ermöglicht es ressourcenarm Verbindungen zu verwalten und das Gesamtsystem nicht durch den Socketserver zu beeinflussen [8, S. 18ff]. Gerade bei Prozess-basierten Implementierungen eines Socketservers kann es bei hohen Verbindungszahlen passieren, dass die datenverarbeitenden Prozesse verhältnismäßig geringe Rechenzeiten im Prozessor erhalten und damit von den Verbindungsprozessen überlagert werden [8, S. 18ff]. Ein Socketserver mit nur einem Prozess dagegen wäre bei moderner Rechenhardware ein Flaschenhals, weswegen die Implementierung der Ausführungsumgebung erlaubt beliebig viele Ereignisschleifen, mittels Instanzen der Klasse *NioSocketSelector*, je instanziiertem Spielmodell zu starten.

Die eigentlichen Clientverbindungen, welche durch den *NioSocketServer* verwaltet werden, sind in der Klasse *NioSocketConnection* beschrieben. Diese Klasse implementiert die Schnittstelle *SocketConnection* und beschreibt wie Daten empfangen, verarbeitet und versendet werden. Dabei kommen zur Verarbeitung die im folgenden beschriebenen Komponenten in Kapitel 6.2.2 (S. 138) und Kapitel 6.2.3 (S. 153) zum Einsatz.

### 6.2.2 Abbildungsformat

Unter der Annahme, dass Client und Server synchron publiziert werden, ist es nicht notwendig, dass anhand des Datenstromes Interpretationen durchgeführt werden um zu erkennen, welches Attribut wo steht, welcher Datentyp verwendet wird und wie Werte zu finden sind. Sind diese Interpretationen nicht notwendig, können Strukturdaten, wie Bezeichner und Datentypen, entfernt werden. Dieses Vorgehen bei der Abbildung setzt voraus, dass Client und Server eine exakte Vorstellung von Struktur und Aufbau der Datenpakete haben.

Für das Abbildungsformat wird eine Datenrepräsentation auf binärer Ebene verwendet. Dieses Format muss es erlauben Nachrichten eindeutig zu identifizieren und Parameter korrekt zu reproduzieren. Die dafür notwendigen Infor-

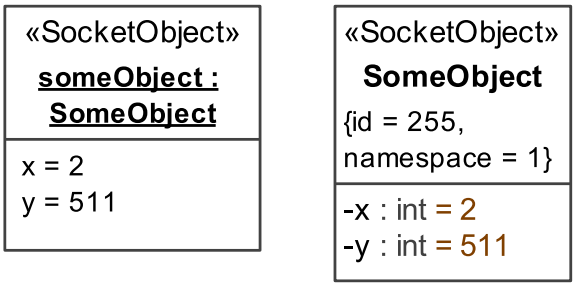



**Abbildung 6.11: Kommunikationsobjekt mit zwei ganzzahligen Attributen. Eine Implementierung ist in Anhang J (S. 253) in Quellcode J.2 zu finden.**

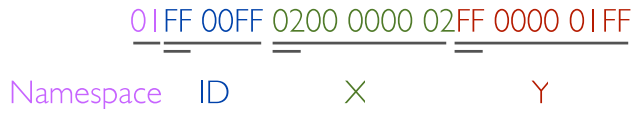



**Abbildung 6.12: Struktur des Kommunikationsobjektes aus Abbildung 6.11.**

mationen werden aus den transportierbaren Entitätstypen abgeleitet.

Das Abbildungsformat setzt sich aus vier Regelsätzen zusammen: Identifikation von Kommunikationsobjekten mittels Objektbezeichner, Basisattribut- und Reihenabbildung, sowie Kommunikationssequenzen. Diese Regelsätze müssen sowohl in der Server-Implementierung, als auch den Client-Implementierungen identisch sein. Dabei gilt, je geringer der Umfang des Grundregelsatzes ist, desto effizienter ist die Implementierung einer neuen Clienttechnologie möglich. Darüber hinaus gilt, jedes Datenpaket beginnt mit dem Identifikator des transportierten Kommunikationsobjektes, gefolgt von einer durch das Kommunikationsobjekt definierten Menge an Attributsegmenten.

Um Kommunikationsobjekte eindeutig identifizieren zu können, jedoch nicht ihren Bezeichner als Identifkationselement nutzen zu müssen, wird jede transportierbare Entitätstypen mit einem numerischen Identifkator versehen. Mithilfe dieses Identifikators wird bestimmt, wie der folgende Datenstrom zu interpretieren ist und erlaubt die Rekonstruktion des Kommunikationsobjektes ohne Transport von Strukturdaten.

Der in dem Kommunikationsobjekt eindeutig definierte Identifikator *id* und Namensraum *namespace* ergeben den Objektbezeichner. Die Angabe von Iden-

```
1 NUMERIC = "1" | "2" | "3" | "4" | "5" |
2           "6" | "7" | "8" | "9" | "0";
3 LETTER = "A" | "B" | "C" | "D" | "E" | "F" ;
4 VALUE = NUMERIC | LETTER ;
5 NAMESPACE = VALUE VALUE ;
6 LOW_PART = VALUE VALUE ;
7 HIGH_PART = VALUE VALUE ;
8 IDENTIFIER = NAMESPACE LOW_PART HIGH_PART LOW_PART ;
```

**Quellcode 6.1: Spezifikation des Objektbezeichners (EBNF)**

tifikator und Namensraum ist optional. Werden die Werte nicht angegeben, werden sie zur Laufzeit bestimmt. Soll der Identifikator und/oder Namensraum definiert werden, wird diese Informationen mit Hilfe einer Annotation direkt an der Klasse angegeben. Der Namensraum kann einen Wert zwischen 0 und 256 annehmen, wobei der Wertebereich zwischen 128 und 256 für systemnahe Dienste reserviert ist. Der Identifikator kann einen Wert zwischen 0 und $2^{16}$ annehmen. Daraus ergeben sich maximal $2^{24}$ unterschiedliche Kommunikationsobjekte je Protokollinstanz – den reservierten Namensraumbereich ausgenommen bleiben $2^{23}$ mögliche Kommunikationsobjekte.

Abbildung 6.11 zeigt ein Kommunikationsobjekt mit zwei ganzzahligen Parametern, Abbildung 6.12 zeigt die daraus resultierende Abbildung in Hexadezimaldarstellung. Dieser Datenstrom beginnt mit dem Namensraum (Bit 0..7), gefolgt von der Identifikationsnummer (Bit 8..31). Die Identifikationsnummer wird in 24 Bit codiert, wobei 8 Bit eine Prüfsumme der folgenden 16 Bit darstellen. Quellcode 6.1 zeigt wie der Objektbezeichner aufgebaut ist.

Die ersten 32 Bit definieren wie die folgenden Daten zu interpretieren sind. Als Interpretationsvorlage dienen die Informationen der Klasse des Kommunikationsobjektes. Jedes Attribut wird in einem exakt berechenbaren Bereich des Datenstroms abgebildet. Im Resultat kann solange eindeutig dekodiert werden, wie Client und Server identische Informationen zu den Kommunikationsobjekten zur Verfügung stehen.

*SomeObject* definiert die Attribute $x$ und $y$ mit den Basisdatentyp Integer. Das Abbildungsformat sortiert die Attribute in alphabetischer Reihenfolge und platziert sie als Attributsegmente im Anschluss an den Objektbezeichner. Je nach Datentyp werden Werte unterschiedlich im Datenstrom repräsentiert und verbrauchen eine konstantes Volumen. Quellcode 6.1 liefert hierfür einen Über-

| *Datentyp* | *Beispiel* | *Beispiel (Hex)* | *Beschreibung* |
|---|---|---|---|
| byte | 127 | 7F | vorzeichenbehaftete Repräsentation in 8 Bit |
| short | 32.767 | 7FFF | vorzeichenbehaftete Repräsentation in 16 Bit |
| int | 1.234.567 | 87 0012 D687 | vorzeichenbehaftetes Repräsentation in 40 Bit, 8 Bit Prüfsumme, 32 Bit für Integer |
| long | Darstellung als double, mit einer 100 % Genauigkeit in Höhe der Mantisse (252) | | |
| float | 1.234 | 3FE3 8DCA | 32 Bit Darstellung für Gleitkommazahl nach IEEE 754 |
| double | 1.234 | 3F8D C276 B8CC 3958 | 64 Bit Darstellung für Gleitkommazahl nach IEEE 754 |
| char | A | 41 | UTF-8 Bytekette |
| boolean | true | 01 | Darstellung als byte, true = 0x01, false = 0x00 |

**Tabelle 6.1: Beispieldaten und Merkmale der Basisdatentypen. Alle Werte werden nach Big-Endian Reihenfolge abgebildet.**

blick. Die Basisdatentypen werden, insofern möglich, nach gängigen Standards abgebildet, jeweils mit der höchsten Wertigkeit zuerst (Big-Endian). Die Implementierung des Abbildungsformates erlaubt es weitere Basisdatentypen zu ergänzen. Das Ergänzen eines weiteren Basisdatentyps zieht jedoch eine Anpassung in allen unterstützten Clientsprachen mit sich.

Das Abbildungsformat bildet mehrdimensionale Reihen für die genannten Datentypen in einem eigenen Attributsegment ab, welches rekursiv in den deklarierten Datentypen aufgelöst wird. Dieses spezielle Attributsegment beginnt mit einer Anzahl für die erste Dimension der Reihung gefolgt von den Werten. Abbildung 6.13 zeigt ein Kommunikationsobjekt mit dem Attribut *list* vom Typ einer ganzzahligen Reihung. Im Datenstrom wird erneut der Objektbezeichner abgebildet, gefolgt von der Anzahl der Elemente der Reihung und abschließend die einzelnen Werte der Reihung. Abbildung 6.14 zeigt den Datenstrom.

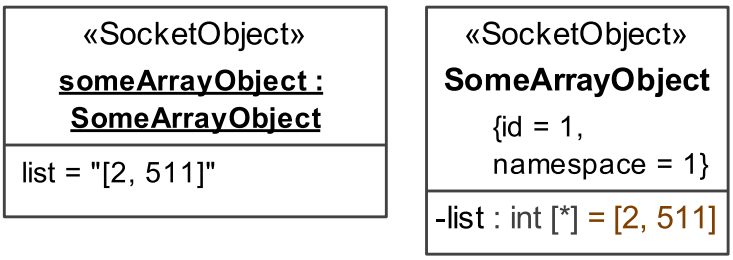



**Abbildung 6.13: Kommunikationsobjekt mit einer Reihung mit ganzzahligem Typ. Eine Implementierung ist in Anhang J (S. 253) in Quellcode J.3 zu finden.**


**Abbildung 6.14: Struktur des Kommunikationsobjektes aus Abbildung 6.13.**

Alle komplexe Datentypen werden rekursiv bis auf ihre Basisdatentypen aufgelöst, insofern sie als transportierbar markiert sind. Komplexe Datentypen werden in einem Attributsegment abgebildet, welches für sich betrachtet mit einem Objektbezeichner beginnt und in Folge die alphabetisch sortierten Attributsegmente dieses Kommunikationsobjektes abbildet. Der daraus gebildete Baum kann eine beliebige Tiefe, mit beliebig vielen Abhängigkeiten besitzen. Komplexe Datentypen können, wie Basisdatentypen, als Wert in einer Reihung existieren, womit ein und mehrdimensionale Reihungen aus komplexen Datentypen abbildbar sind. Eine führende Anzahlkennziffer definiert wie oft im folgenden der Datentyp auszulesen ist. Der Objektbezeichner definiert den eigentlichen Datentyp und die Interpretationsvorschrift womit Spezialisierungen im Sinne von Vererbungsstrukturen abbildbar bleiben.

In Abbildung 6.15 wird das Kommunikationsobjekt *SomeParent* definiert, welches als Datentyp für das Attribut *ref* das Kommunikationsobjekt *SomeObject* aus Abbildung 6.11 verwendet. Der daraus resultierende Datenstrom enthält den Objektbezeichner für *SomeParent* (0..31) gefolgt von dem Attributsegment für die Instanz von *SomeObject*. Dieses Segment für sich betrachtet beginnt wieder mit einem Objektbezeichner (32..63) gefolgt von den Integer Attributen $x$ und $y$ (64..103 und 104..144). Abbildung 6.16 zeigt diesen Da-

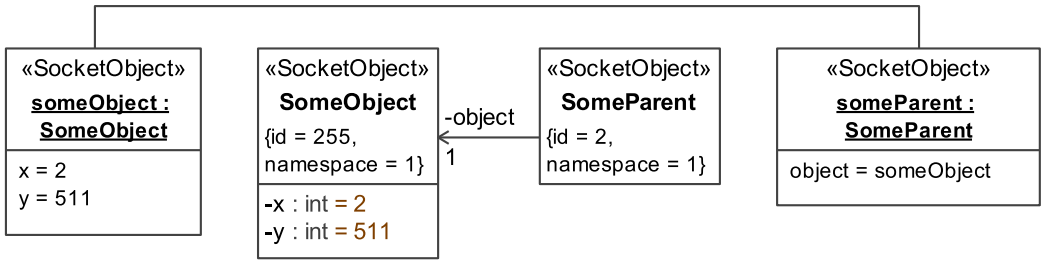



**Abbildung 6.15: Kommunikationsobjekt mit einem komplexen Datentyp. Eine Implementierung ist in Anhang J (S. 253) in Quellcode J.4 zu finden.**

```
0102 0002 01FF 00FF 0200 0000 02FF 0000 01FF
                  ref (x, y)
```

**Abbildung 6.16: Struktur des Kommunikationsobjektes aus Abbildung 6.15.**

tenstrom.

Sonderrollen in dem beschriebenen binären Repräsentationsverfahren nehmen Zeichenketten (*String*) und Aufzählungstypen (*Enum*) ein. Zeichenketten werden nicht als Kommunikationsobjekte betrachtet. Die Daten werden in Form eines Bytearrays abgebildet. Dieses startet mit der Länge der Bytekette, gefolgt von den Binärwerte jedes Zeichens encodiert mittels UTF-8. Aufzählungstypen dagegen werden mit dem Basisdatentyp Integer abgebildet. Als Wert kommt die Ordnungszahl des zu übertragenden Elements zum Einsatz.

Eine Herausforderung bei der Serialisierung von Kommunikationsobjekten ohne Übertragung von Strukturdaten stellen Attribute mit variablen Datentyp dar. In Java sind dies Attribute vom Typ *Object* oder generische Datenstrukturen mit der Typisierung „?“, welche zur Laufzeit ebenso dem Typ *Object* entsprechen. Während alle komplexen Datentypen durch ihre eindeutigen Objektbezeichner abbildbar sind, würde das Abbildungsformat bei Instanzen der Basisdatentypen in Java einen Fehler erzeugen. Wird z. B. eine Ganzzahl *int* auf ein Attribut vom Typ *Object* zugewiesen, transformiert Java den Basisdatentyp in eine Instanz vom Typ *Integer* in der die Ganzzahl gekapselt ist. Das Protokoll verarbeitet *Integer* und *int* identisch als ganzzahligen Basisdatentyp. Da in diesem Fall die Metadaten des Attributes nicht ausreichend exakt sind

| *Datentyp* | *ID* |
|---|---|
| Integer | 1 |
| String | 2 |
| Double | 3 |
| Byte | 4 |
| Boolean | 5 |
| Long | 6 |
| Short | 7 |
| Char | 8 |
| Float | 9 |
| Object[] | 10 |
| Enum | 11 |
| Void | 12 |
| komplexes Objekt | 13 |

**Tabelle 6.2: Identifikationsnummern für Basisdatentypen, Enums, komplexe Objekte und Objekt Reihen.**

um dies zu rekonstruieren, benötigt dieses Abbildungsformat zusätzliche Informationen für Attribute vom Typ *Object*. Die Informationen werden in 8 Bit Segment vor dem Attribut kodiert und definieren, wie die folgenden Bits zu lesen sind: entweder als einer der beschriebenen Basisdatentypen, als komplexen Datentyp oder als Reihe dieser. Quellcode 6.2 nennt die Identifikationsnummern der verschiedenen Typen. Ein Beispiel dieses Anwendungsfalles ist in Abbildung 6.17 und in Abbildung 6.18 dargestellt.

Ein weiteres Element der Spezifikation dieses Abbildungsformates bezieht sich auf Kommunikationssequenzen. Bezüglich den Anforderungen, ist es not-

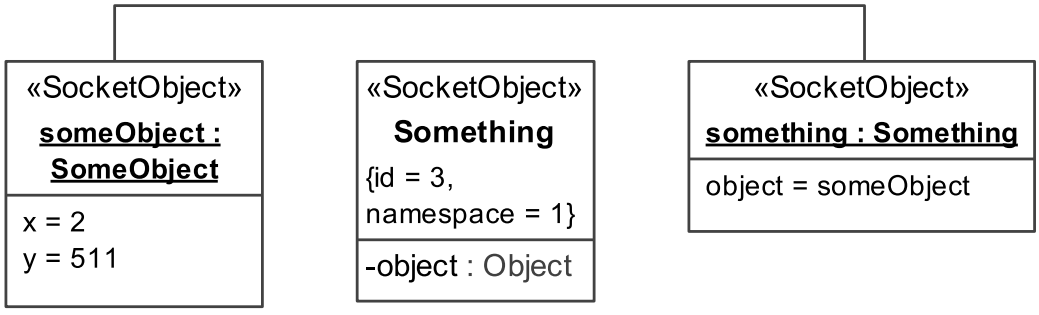



**Abbildung 6.17: Kommunikationsobjekt mit einem variablem Datentyp. Eine Implementierung ist in Anhang J (S. 253) in Quellcode J.5 zu finden.**


**Abbildung 6.18: Struktur des Kommunikationsobjektes aus Abbildung 6.17.**

wendig, innerhalb eines Kommunikationskanals zu wissen, welche Antwort zu welcher Anfrage gehört um eine eindeutige Behandlung auszuführen. Dessen Bedeutung steigt vor allem in Anwendungsfällen, in denen Pakete asynchron von beiden Seiten an die jeweils andere verteilt werden können. Eine solche Sequenz kann durch eine Identifikationsnummer realisiert werden, welche in der Anfrage und Antwort identisch ist. Sendet der Client eine Anfrage, dann muss der Server alle aus dieser Anfrage entstehenden Antworten mit dem selben Sequenzidentifikator versehen, damit der Client zugehörige Prozesse auflösen kann. Diese Information könnte mittels einem Attribut und Vererbungshierarchien bei Kommunikationsobjekten realisiert werden, im Rahmen weiterführender Mechaniken der Ausführungsumgebung wurden diese jedoch in die Spezifikation des Abbildungsformates übernommen. Die Ausführungsumgebung sieht hierfür eine generalisierte Klasse vor: *SocketData*. Kommunikationsobjekte könnnen diese spezialisieren um die Sequenzbezifferung zu aktivieren. Der Sequenzidentifikator wird im Abbildungsformat, falls das Kommunikationsobjekt die Klasse *SocketData* spezialisiert, nach dem Objektbezeichner abgebildet. Das Abbildungsformat behandelt den Sequenzidentifikator nicht als eigenständiges Attribut, da es nachträglich manipulierbar sein soll und dafür die Position im Datenstrom effizient, ohne Analyse des Paketes, berechenbar sein muss. Ein Anwendungsfall sind Antwortobjekte bezüglich einer Anfrage, welche an mehrere Empfänger gerichtet sind. Während der antwortauslösende Client eine Anfragebehandlung den korrekten Identifikator für die Sequenz benötigt, erhalten alle anderen Clients eine Antwort mit neutralisierten Identifikator. Die Neutralisierung ist notwendig, da Sequenzidentifikatoren nur im jeweiligen Kommunikationskanal garantiert eindeutig sind. Die Übetragung eines fremden Sequenzidentifikators kann zu unvorhersehbaren Effekten führen.

Der Sequenzidentifikator wird im Abildungsformat in 16 Bit abgebildet. Dies ist ausreichend für $2^{16} - 1$ unabhängige Anfrageströme, der Identifikator 0 gilt als sequenzloses Kommunikationsobjekt. Ist der maximale Definitionsbe-

reich überschritten, soll die Sequenznummer erneut bei 1 beginnen. Ein Client soll zudem solange die Behandlungsroutine der Anfrage mit dem Identifikator $n$ verhalten, bis der Sequenzidentifikator den Definitionsbereich überschritten hat und bei $n - 1$ angekommen ist. Die Protokollimplementierung sieht eine modifizierbare Sequenzidentifikatorlänge vor, um Anwendungen mit hohen Sequenzaufkommen eine fehlerfreie Behandlung zu ermöglichen. Sollte ein Client ausschließlich sequenzbehaftete Kommunikationsobjekte versenden, würde bei einem Intervall von einer Millisekunde und einer durchschnittlich abgebildeten Kommunikationsobjektgröße von 48 Bit (32 Bit Objektbezeichner und 16 Bit Sequenzidentifikator) ein Datenstrom von $5,8594$ kB/s Rohdaten entstehen, die der Server innerhalb von $2^{16} - 1$ Millisekunden, bzw. ca. $65,5$ Sekunden beantworten müsste, bevor die Behandlungsroutine des Sequenzidentifikators erneut bei 1 beginnt wird. In einem realen Anwendungsfall kommen sequenzbehaftete Kommunikationsobjekte bei nutzerbasierten Clientinteraktionen zum Einsatz, z. B. wenn ein Formular abgesendet wird, wenn Daten ausgegeben werden oder eine Spielfigur bewegt wird. Diese Clientinteraktionen sind zumeist je Sekunde, anstatt je Millisekunde, zu erwarten und ermöglichen dem Server die Antwort innerhalb von ca. einer Stunde zuzustellen bevor der Sequenzidentifikator ungültig wird.

Damit Fehler in gewollt oder ungewollt kompromittierten Datenströmen erkannt werden können – zum Beispiel wenn nicht kompatible Client- und Server-Versionen über die bereitgestellte Schnittstelle kommunizieren – gibt es in der Protokollspezifikation zwei Prüfsummen. Beide arbeiten mit einem einfachen Modulo. Die Prüfsummen für sich sichern den Datenstrom nicht ausreichend. Das Abbildungsformat selbst definiert jedoch durch seinen Grundaufbau weitere Prüfelemente. Durch eine bitgenaue Vorgabe und der resultierenden Schablone für Datenströme ist keine Abweichung in Länge und Struktur möglich. Jedes Kommunikationsobjekt definiert damit eine Grundsammlung an Strukturregeln die Fehlerhafte Datenpakete identifizieren können. Zum Beispiel definiert Abbildung 6.11 folgende Rahmenbedingungen:

1. Der vorzeichenfreie ganzzahlige Wert der Bits 8..15 müssen dem Modulo des vorzeichenfreien ganzzahligen Werts der Bits 16..31 mit 28 entsprechen.

2. Die Bitfolge muss exakt $32 + 40 + 40 = 112$ Bits lang sein.

3. für den aus der Bitfolge 0..31 extrahierten Objektbezeichner muss ein Kommunikationsobjekt hinterlegt wurden sein.

4. Der vorzeichenbehafteten ganzzahlige Wert der Bits 32..39 müssen dem Modulo des vorzeichenbehafteten ganzzahligen Werts der Bits 40..71 mit 27 entsprechen.

5. Der vorzeichenbehafteten ganzzahlige Wert der Bits 72..79 müssen dem Modulo des vorzeichenbehafteten ganzzahligen Werts der Bits 80..111 mit 27 entsprechen.

6. Die diesem Datenstrom folgende Nachricht muss der 1. Regelung entsprechen.

Bei der Interpretation des Datenstrom müssen alle sechs Regelungen zutreffen - trifft eine nicht zu, gilt der Datenstrom als korrupt und wird verworfen. Je komplexer Kommunikationsobjekte sind, desto umfangreicher wird diese Menge and Regeln und desto geringer die Wahrscheinlichkeit der fehlerhaften Interpretation. Während diese Detektion von Fehlern die nachgelagerte Verarbeitung, insbesondere das Spielmodell, vor falschen Anfragen und Informationen sichert, sorgt es jedoch für den Verlust aller bereits empfangenen Daten. Da das Datenpakete keine Start und Endinformation enthalten, ist es nicht möglich zu erkennen, wann das Ende der korrupten Nachricht erreicht ist. In der Praxis hat sich dieser Nachteil nie zu einer Schwierigkeit entwickelt. Zum einen weil kaum mehr als ein Kommunikationsobjekt, bzw. Anfrage, zur selben Zeit verarbeitet wird und zum anderen weil die Spezifikation des übergeordneten TCP Protokolls bereits die Reihenfolge der Pakete und die korrekte Auslieferung sicherstellt. Ergänzende Sicherungsmaßnahmen zur Detektion von defekten Datenpaketen wären Redundant.

Das metadatenfreie Abbildungsformat liefert die Basis zur kompakten Repräsentation von Daten ohne direkten Sprachbezug. Die Umsetzung des Abbildungsformates in einer Java Anwendung verlangt neben einer kompakten Datenstromerzeugung nach einer performanten Umsetzung.

Betrachtet man die Ausführungszeiten der Serialisierungsprozesse verbreiteter Nachrichtenformate (siehe Kapitel 3.5, S. 77), gewinnt dieser Faktor an

Bedeutung. Je mehr Zeit die Deserialisierung von Daten benötigt - desto länger ist die Antwortzeit der Applikation. Das Ziel für eine Ausführungsumgebung für Spielmodelle muss eine zeitnahe Reaktion bei Clientanfragen sein, bei der Rechenzeit nicht der Interpration von Datenströmen zufällt.

Während eine Implementierung des Abbildungsformates auf Basis der Java Reflection API für den Zugriff auf Attribute zeitintensiv ist, wurde für die Umsetzung des Serialisierungsalgorithmus die unter Kapitel 6.1.1 (S. 126) beschriebenen Verfahren zur Generierung von Bytecode zur Laufzeit angewendet. Die Implementierung des Abbildungsformates besteht aus einem Pool an Grundfunktionalität die die binäre Abbildung von Basisdatentypen ermöglicht und einem Codegenerator, der die Strukturdaten von Kommunikationsobjekten einmalig verarbeitet und in Form von implementierten Ableitungen der Klasse *Converter* bereitstellt. Dieser generierte Code wird anschließend in Bytecode überführt, über einen speziellen Klassenlader in die Ausführungsumgebung injiziert und zur Serialisierung und Deserialisierung von Kommunikationsobjekten verwendet.

Durch den Einsatz von generierten Verfahren zur Abbildung kann auf Identifikationsprozesse für Datentypen verzichtet werden. Attribute können direkt, ohne Umweg über die Reflection API, aus dem Kommunikationsobjekt bezogen werden. Dadurch sind Optimierungsprozesse, wie Just-In-Time-Kompilierung in Java, im Rahmen der Ausführung der Implementierung des Abbildungsformates möglich. Abbildung 6.19 zeigt den grundlegenden Klassenaufbau der Implementierung des Abbildungsformates.

Die Klasse *Protocol* stellt eine allgemeine Implementierung zur Steuerung des Lebenslaufes eines Abbildungsformates bereit. Mit dieser Klasse lassen sich Kommunikationsobjekte serialisieren und deserialisieren. Bei der Erzeugung einer neuen Instanz mittels Aufruf von *newInstance* in über die Reflection API wird der Instanz eine Implementierung der Schnittstelle *ProtocolHandler* übergeben, sowie eine Liste der unterstützten Typen von Kommunikationsobjekten. Wie die Implementierung aus den unterstützten Klassen Informationen ableitet und welche Datenrepräsentation verwendet wird ist auf dieser Abstraktionsebene unbekannt und der Implementierung überlassen. Auf diese Art lassen sich JSON und XML basierte Abbildungsformate, anstatt dem eigenen Abbildungsformat, verwenden. Die Klasse *ByteBufferWrapper* dient als

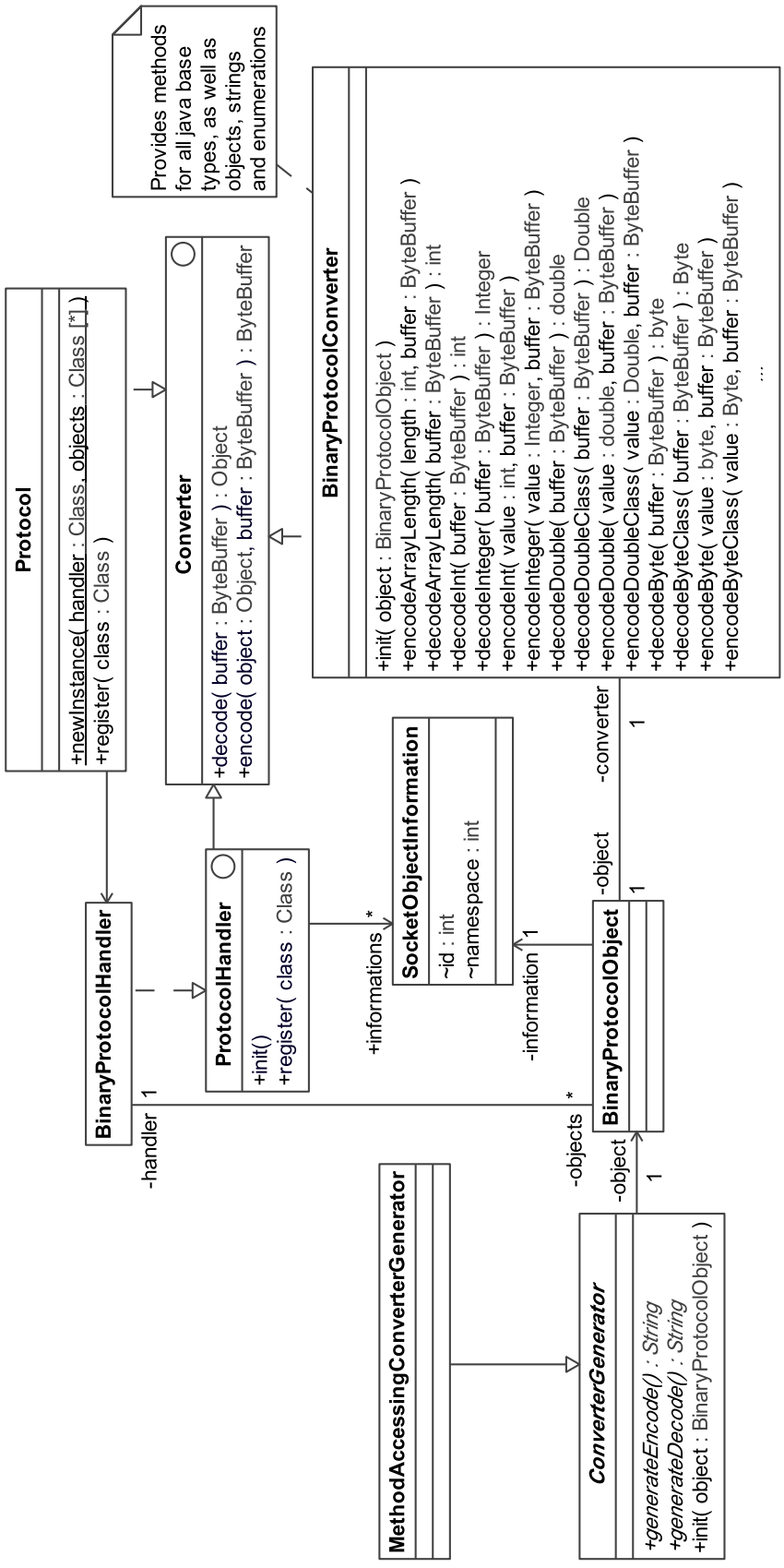



**Abbildung 6.19: Klassendiagramm der Implementierung des Abbildungsformates in Java.**

```
1  public class SomeObject2134512 extends
    BinaryProtocolConverter {
2    public void encode(Object original, ByteBufferWrapper
    buffer) {
3      SomeObject object = (SomeObject) original;
4      this.encodeInt(object.getX(), buffer);
5      this.encodeInt(object.getY(), buffer);
6    }
7    public Object decode(ByteBufferWrapper buffer) {
8      SomeObject object = new SomeObject();
9      object.setX(this.decodeInt(buffer));
10     object.setY(this.decodeInt(buffer));
11     return object;
12   }
13 }
```

**Quellcode 6.2: Generierte Konvertierungsklasse.**

Zugriffswrapper auf *ByteBuffer*-Instanzen und ermöglicht den kontrollierten Eingriff in Lese- und Schreibprozesse der jeweiligen Abbildungsformate. Zum Beispiel lassen sich auf diesen Weg Logausgaben produzieren die die unleserlichen Binärströme leserlich darstellen.

Der *BinaryProtocolHandler*, die Implementierung für das zuvor beschriebene Verfahren für das Abbildungsformat, verwendet für die Markierung von übertragbaren Kommunikationsobjekten und die Definition des Objektbezeichners die Annotation *@SocketObject*. Zusätzliche Informationen zu Attributen werden darüber hinaus mittels der Reflection API gesammelt. Alle gesammelten Informationen werden in einem festen Ablaufplan in Java Quellcode abgebildet und zur Laufzeit mit dem Werkzeug Javassist in Bytecode überführt. Quellcode 6.2 zeigt den generierten Code für das in Abbildung 6.11 definierte Kommunikationsobjekt.

Die bei der Vorverarbeitung entstehenden Beschreibungen zur Konvertierung werden der *BinaryProtocolObject*-Instanz zugeordnet und für zukünftige Serialisierungen verwendet. Soll ein Kommunikationsobjekt serialisiert werden, wird dies mittels *Protocol*-Instanz angestoßen. Diese reicht die Anfrage an die verwendete *ProtocolHandler* Implementierung. Hier wird für die Instanz das zugehörige *BinaryProtocolObject* bestimmt, und der assoziierte *BinaryProtocolConverter* aufgerufen. Der Ablauf ist in Form eines Aktivitätsdiagramms in Abbildung 6.20 dargestellt. Eine Instanz von *SomeObject* (siehe Abbildung

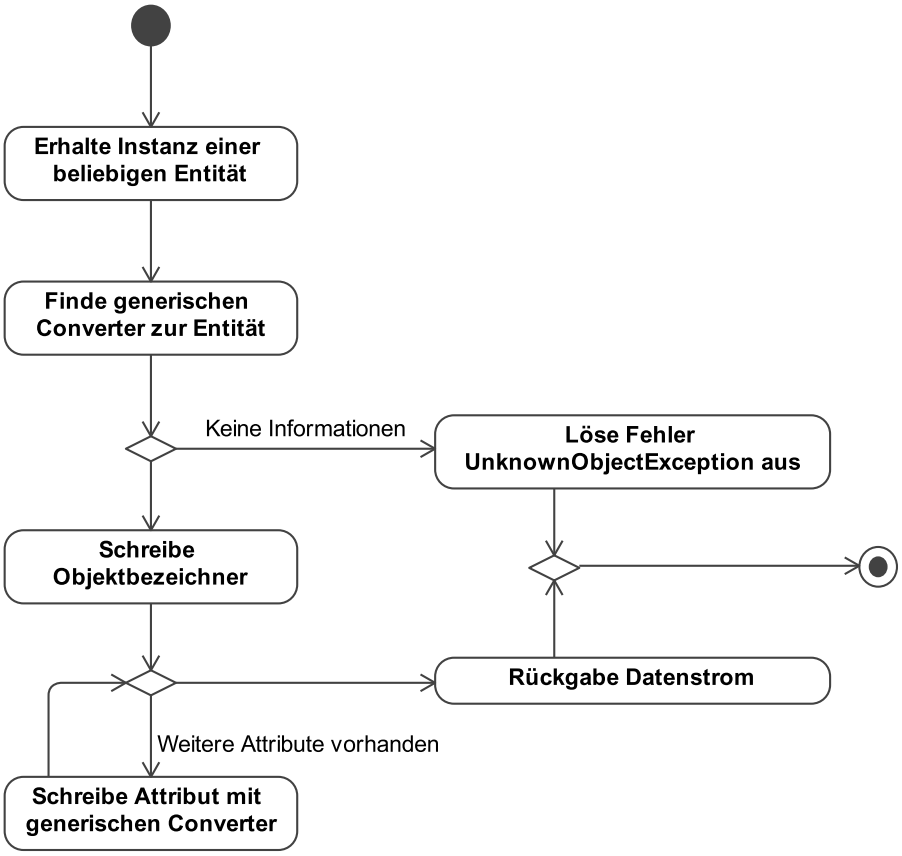



**Abbildung 6.20: Ablauf der Serialisierung eines Kommunikationsobjektes.**

6.11) wird an das *Protocol* gesendet, dieser erzeugt einen Buffer für das Ergebnis und reicht den Aufruf an den verwendeten *ProtocolHandler* weiter. Dieser sucht für die Instanz die passenden *BinaryProtocolObject* Instanz und reicht die Serialisierung an diese weiter. Wird keine Instanz gefunden soll ein Fehler zurück gegeben werden. Das *BinaryProtocolObject* serialisiert das Kommunikationsobjekt mit der generierten Konvertierungsklasse.

Der Deserialisierungsprozess wird ebenso über die *Protocol*-Instanz gestartet, verwendet als Input einen Datenstrom, repräsentiert durch eine Instanz der *ByteArrayWrapper*-Klasse. Der Aufruf wird an die unterliegende *ProtocolHandler* Implementierung weitergeleitet. Diese liest die Informationen für den Objektbezeichner aus den ersten 32 Bit und prüft dessen Korrektheit. Ist diese Korrekt, wird versucht anhand dieser Informationen eine passende Instanz der Klasse *BinaryProtocolObject* zu finden. Mit dieser Instanz kann anschließend das eigentliche Kommunikationsobjekt erzeugt werden. Abbildung 6.21 zeigt

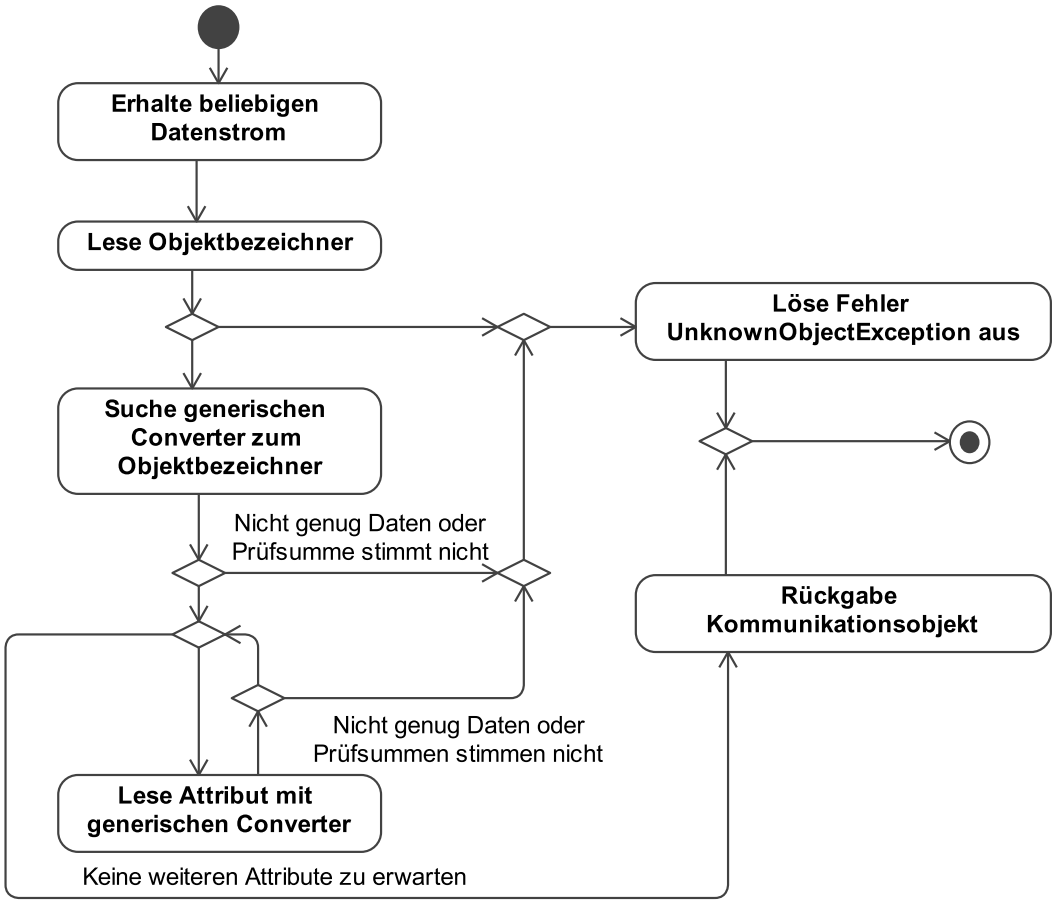



**Abbildung 6.21: Ablauf der Deserialisierung eines Kommunikationsobjektes.**

den Abblauf in einem Aktivitätsdiagramm.

Das resultierende Abbildungsformat kann abschließend in Anlehnung an die Betrachtung in Kapitel 3.1.1 (S. 32) mithilfe des Kommunikationsobjektes aus Abbildung 3.2 analysiert werden. Dabei werden erneut Nutz- und Strukturdaten gegenübergestellt um zu analysieren, wieviel Volumen eines Datenpaketes den eigentlichen Nutzdaten zufällt. Die im Anhang H (S. 249) dargestellte Analyse ergibt die Abbildung 6.22 visualisierte Verteilung zwischen Nutz- und Strukturdaten. Das Abbildungsformat erreicht mit dem verwendeten Beispiel 69 % Anteil an Nutzdaten im Verhältnis zum gesamten Datenpaket.

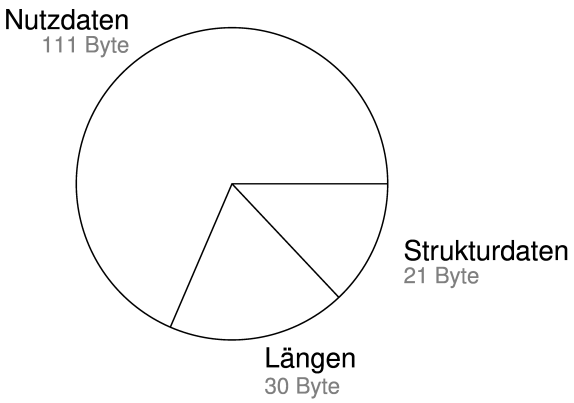



**Abbildung 6.22: Verhältnis von Nutz- und Strukturdaten im Datenstrom des Abbildungsformates der Ausführungsumgebung bei Anwendung auf das Beispiel aus Abbildung 3.2.**

### 6.2.3 Anfrageverarbeitung

Für die Verarbeitung von Anfragen innerhalb der Ausführungsumgebung ergeben sich je Instanz eines Spielmodells eine abgegrenzte Menge an Verarbeitungsanweisungen die bei dem Eintreffen von bestimmten Kommunikationsobjekten ausgeführt werden sollen. Sobald ein Kommunikationsobjekt vollständig durch den *Server Message Handler* vom Client empfangen wurde, wird dieses an den *Dispatcher* des Subsystems *Request Handling* übergeben. Die im diesem Zusammenhang verwendeten Klassen sind im Diagramm in Abbildung 6.23 dargestellt.

Zur Verarbeitung werden aus dem *Request Handler Pool* anhand des Types des empfangenen Kommunikationsobjektes registrierte Verarbeitungsanweisungen vom Typ *RequestHandler* gesucht. Die einfachste Form sind die sogenannten *Custom Request Handler*.

Eine einfache Verarbeitungsanweisung spezialisiert die abstrake Klasse *RequestHandler*. Hierfür muss sie die Methoden *validate* und *handle* überschreiben, welche die zwei Schritte für eine Verarbeitungsanweise darstellt. Für die Verarbeitung wird die spezialisierte Anweisung instanziiert und initiiert. Im Rahmen der Initiierung erhält die Instanz einen Verweis auf das eingegangene Kommunikationsobjekt und den Kontext. Anschließend wird die Methode *validate* aufgerufen, um zu prüfen ob das empfangene Kommunikationsobjekt von dieser Verarbeitungsanweisung behandelt werden kann. Abschließend, und

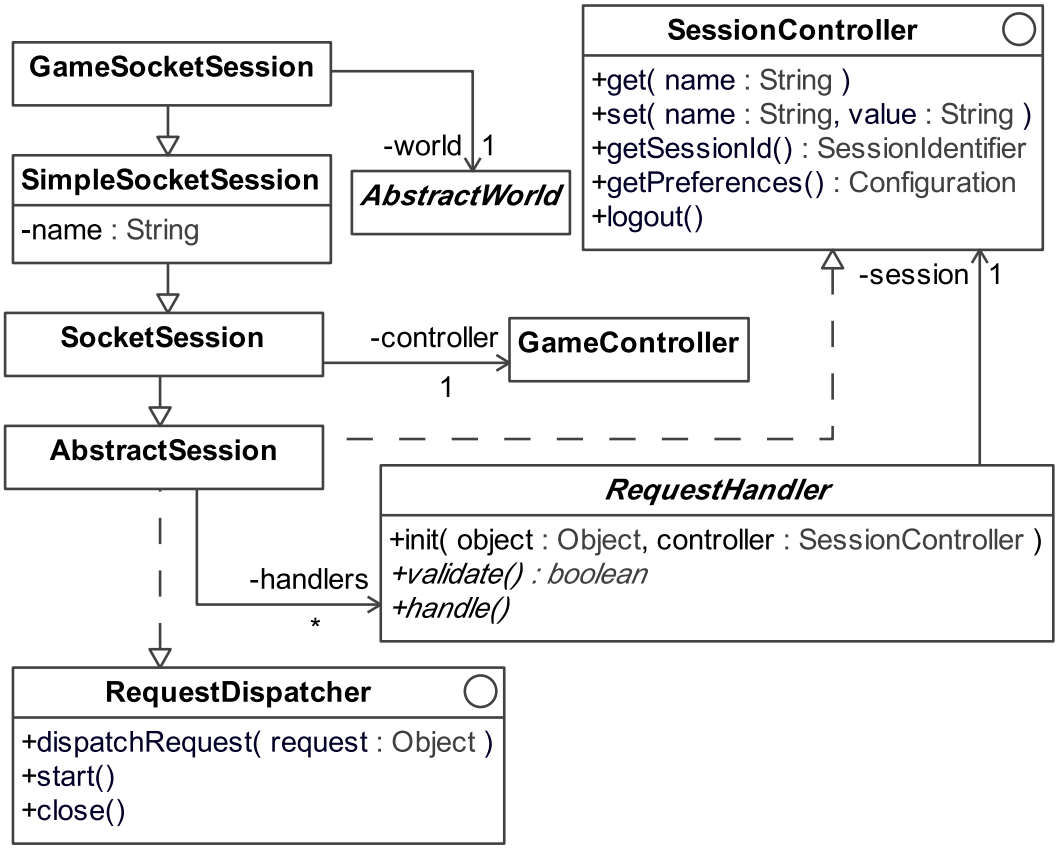



**Abbildung 6.23: Klassendiagramm zur Anfrageverarbeitung mit *RequestDispatcher* und Implementierung, sowie abstraktem *RequestHandler* als Generalisierung für Behandlungsroutinen.**

nur wenn *validate* erfolgreich aufgerufen wurde, wird die Methode *handle* ausgeführt, welche die eigentlichen Anweisungen zur Bearbeitung des Kommunikationsobjektes enthält.

Die Kopplung zwischen Kommunikationsobjekt und Verwarbeitungsanweisung wird mithilfe der Annotation *@UseHandler* am Kommunikationsobjekt vorgenommen. Mit dieser Annotation wird für ein Kommunikationsobjekt der Verweis auf die Klasse mit Verarbeitungsanweisungen festgelegt. Die verwiesene Klasse muss vom Typ *RequestHandler* sein. Eine Darstellung zum Verarbeitungsprozess, welcher im Rahmen von empfangenen Daten durchgeführt wird, ist in Kapitel 6.5.3 (S. 175) in Kombinationt mit beteiligten Komponenten dargestellt.

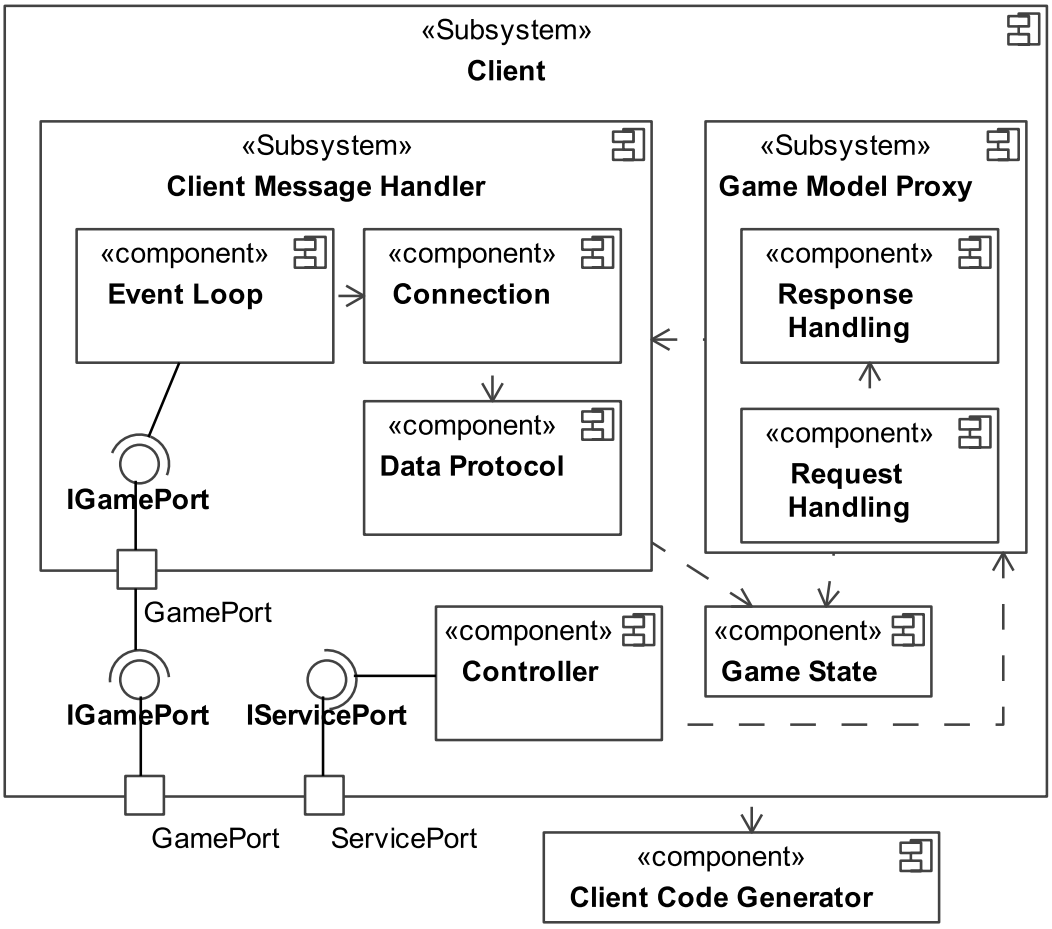



**Abbildung 6.24: Komponentenübersicht vom *Client*.**

## 6.3 Abstraktion der Kommunikation im Client

Zur Interaktion mit dem *Game Model Container* ist zusätzlich eine Kommunikationsinfrastruktur auf der Clientseite notwendig, die mit der Kommunikationsinfrastruktur auf der Serverseite interagiert. Diese Infrastruktur wird mithilfe der Subsysteme *Client Message Handler* und *Game Model Proxy*, dargestellt in Abbildung 6.24, realisiert und im Folgenden die Komponenten beschrieben.

### 6.3.1 Nachrichtenverarbeitung

Als Gegenstück zum *Server Message Handler* fungiert der *Client Message Handler*. Dieser muss in unterstützen Client Sprache implementiert werden

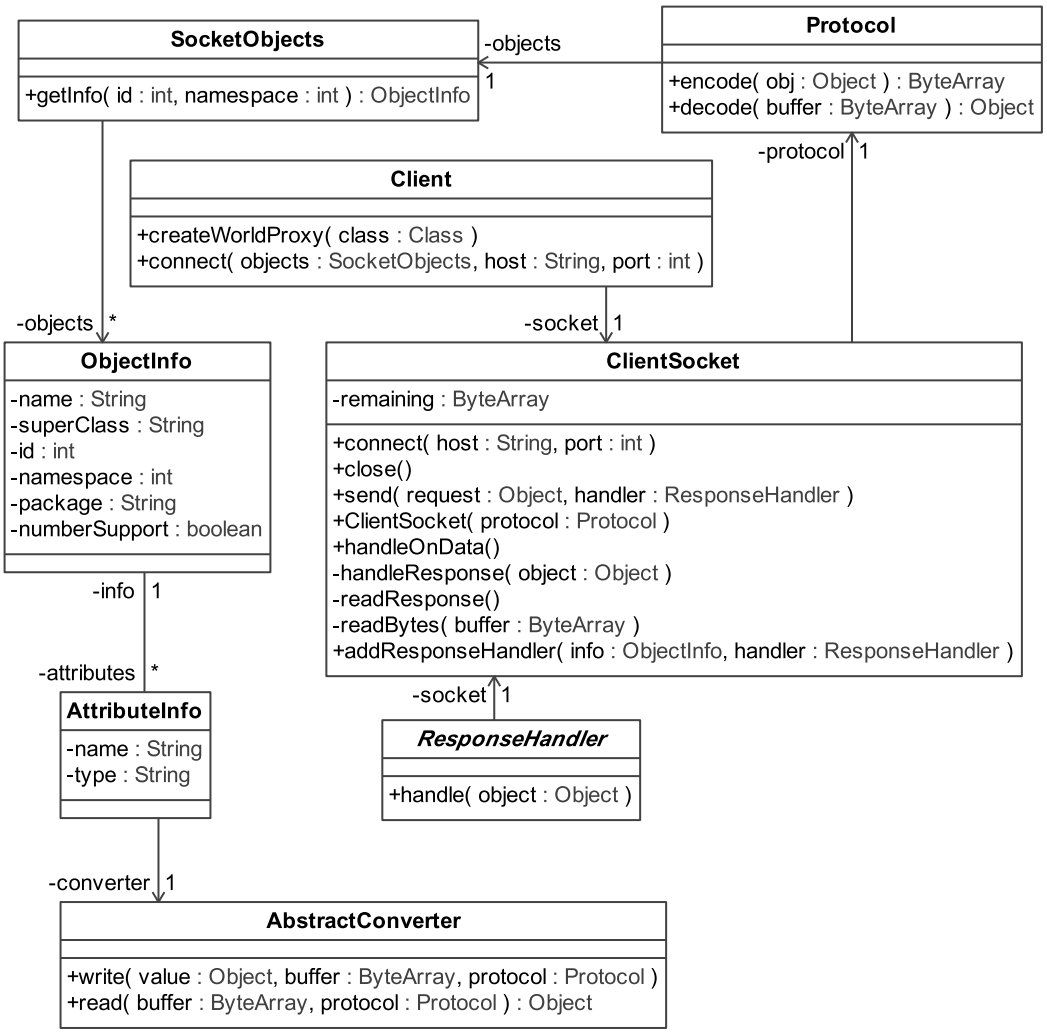



**Abbildung 6.25: Klassendiagramm der im Client zu realisierenden Logik zur Anbindung der Komponente *Server Message Handler*.**

und Baut eine Socket-Verbindung zum Server auf. Unterstützt werden aktuell ActionScript, Java und JavaScript als Client Sprachen. Je nach Client Sprache variiert die Art dieser Verbindung. Im Fall von Flash kommt ein TCP-Socket zum Einsatz, bei dem initial ein XML-Handshake ausgetauscht wird. Davon abweichend kommt im Fall von JavaScript ein WebSocket zum Einsatz, welcher einen eigenen Handshake auf Basis eines HTTP-Headers ausführt und anschließend sogenannte Dataframes über ein TCP-Socket austauscht. Im Fall von Java ist keine zusätzliche Anforderung zu berücksichtigen.

Die Abbildung 6.25 stellt die notwendigen Klasse zur Realisierung der Komponente *Event Loop*, *Data Protocol* und *Connection* des Subsystems *Client*

*Message Handler* dar. Einstiegspunkt ist die Klasse *Client*. Dessen Instanzierung und anschließender Aufruf von *connect* – bei dem eine Sammlung von zuverwendenden Kommunikationsobjekten, sowie zusätzlich der Host und Port übergeben wird – erzeugt eine Verbindung zu der aktiven Instanz des Spielmodells.

Damit die Instanz der Klasse *Client* und die damit verbundene Klasse *ClientSocket* die eigentliche Kommunikation und den Datenaustausch mit dem Server realisieren kann, sind Informationen bezüglich der Kommunikationsobjekte, den Absetzen von Anfragen und der Behandlung von Antworten interessant.

Informationen zu Kommunikationsobjekte können in vielen Sprachen, insbesondere JavaScript und ActionScript, nicht wie in Java typensicher und mit einer API erwirtschaftet werden. Protokollbedingt ist es jedoch erforderlich, dass diese Informationen identisch zu den im Server verwendeten sind (siehe Kapitel 6.2.2, S. 138). Während z. B. Java Fließ- und Ganzzahlen in *float*, *double*, *byte*, *short*, *int* und *long* differenziert, gibt es in JavaScript ein Datentyp für alles: *Number* [29, S. 20]. ActionScript 3 bietet dagegen zusätzlich zum Datentyp *Number* spezifische Typen wie *int* und *uint* trotz dessen Kompatibilität zum ECMAScript Standard (Version 3) [4]. Damit diese Sprachbarriere durchbrochen wird, müssen im Client die Informationen je Entität hinterlegt werden.

Realisiert wird die Hinterlegung von Informationen mithilfe einer Instanz der Klasse *ObjectInfo*. Diese enthält alle Informationen, die eine Entität beschreiben würde und hilft dabei, sie wie im Server zu interpretieren. Neben dem eindeutigen Objektbezeichner enthält sie Informationen zu Attributen und ihren sprachunabhängigen Typen. Die durch *ObjectInfo* und *AttributInfo* bereitgestellten Informationen eleminieren die Sprachbarrieren, erfordern jedoch die Pflege der Informationen auf Clientseite. Die zu einer Klasse zu hinterlegenden Instanzen von *ObjectInfo* müssen an ein öffentliches, unveränderliches (insofern möglich), statisches Attribut des entsprechenden Kommunikationsobjektes bereitgestellt werden.

Das Absetzen von Anfragen wird über die Methode *send(request[, handler])* realisiert. Diese Methode nimmt eine Instanz eines Kommunikationsobjektes als Anfrage entgegen, stößt intern die Serialisierungsprozesse an und versendet

das Ergebnis über die API des TCP-Sockets. Optional kann eine Anfrageverarbeitung hinterlegt werden, welche im Fall des Eintreffens einer Antwort zu diesem Kommunikationsobjekt zur Verarbeitung aufgerufen wird. Dies fordert die Möglichkeit einer Sequenzbildung in Kommunikationsobjekten (siehe Kapitel 6.2.2, S. 138).

Die Verarbeitung von Antworten koppelt sich an Objektbezeichner und ggf. Sequenzbezeichner. Ein einfacher Anwendungsfall ist die Erzeugung einer Antwortverarbeitung für ein spezielles eintreffendes Kommunikationsobjekt. Hierfür wird in der Instanz der Klasse *ClientSocket* mit der Methode *addResponseHandler* das Tupel aus *ObjectInfo* und *ResponseHandler* hinterlegt. Während *ObjectInfo* über das Kommunikationsobjekt bezogen werden kann, muss der *ResponseHandler* spezialisiert werden, eine Methode *handle* mit Verarbeitungsinstruktionen implementiert werden und instanziiert an den *ClientSocket* übergeben werden. Dieser nutzt die Instanz von *ObjectInfo* und dessen eindeutigen Objektbezeichner als Ereignisskennung im *ClientSocket*. Trifft ein Datenpaket ein, welches deserialisiert einem Kommunikationsobjekt mit hinterlegten Objektbezeichner entspricht, wird die bereitgelegte Instanz von *ResponseHandler* und dessen methode *handle* aufgerufen um die Verarbeitung der Antwort anzustoßen.

### 6.3.2 Client / Server Synchronisation

Die Thematik zur Vereinfachung der Entwicklung adressiert keine eigene Komponente, jedoch ein Vorgehen innerhalb der Verarbeitungslogik bei eintreffenden Kommunikationsobjekten.

Ein üblicher Anwendungsfall bei der Implementierung von Clientlogik ist das Erfragen oder zyklische Empfangen von Entitäten des Servers und die darauf folgende Aktualisierung des lokal im Client verwalteten Zustands. Verarbeitungslogik dieser Art ist charaktersisiert durch die Zuweisung von Attributen des empfangenen Kommunikationsobjektes auf die Attribute der lokal verwalteten Instanz. Quellcode 6.3 zeigt diesen Anwendungsfall in einem Beispiel mithilfe von Kommunikationsobjekten.

Da in den meisten Fällen Attribute aktualisiert werden, lässt sich der Aufwand für die Aktualisierung durch eine Iteration über alle erwartbaren Attribute beschreiben, unter der Voraussetzung das das Originalobjekt bekannt

```
1  var players = new Array();
2  var clientSocket = new ClientSocket("localhost", 12345);
3  var createPlayerRequest = new CreatePlayerRequest();
4  createPlayerRequest.setPlayerName("Testname");
5  clientSocket.send(createPlayerRequest, function(result) {
6    // Result has PlayerResponse Type
7    var player = new Player();
8    player.setId(result.getPlayerId());
9    player.setPlayerName(result.getPlayerName());
10   player.setHealth(result.getHealth());
11   players.push(player);
12 });
```

**Quellcode 6.3: Verarbeitung der Antwort einer Anfrage: der Inhalt der Antwort wird verwendet um eine neue Instanz der Klasse *Player* auf Clientseite zu erzeugen.**

und identifizierbar ist. Ist die Instanz nicht bekannt, jedoch identifizierbar, muss dieses erzeugt werden und so hinterlegt werden, dass Aufräumprozesse der Ausführungsumgebung nicht daran gehindert sind, die Instanz aus dem Speicher zu beseitigen.

Die Instanzidentifikation kann gelöst werden, indem man an den zu versendenden Kommunikationsobjekt das Attribut markiert, dass für diese Entität als Identifikator dient. Diese Information wird mit Hilfe der *@SocketObjectIdentifier* hinterlegt und kann am Getter des Identifikationsattributs ergänzt werden. Die Wiederauffindbarkeit unter Voraussetzung der nicht Beeinflussung von Aufräumprozessen in der Ausführungsumgebung (Garbage Collecting) kann mittels schwache Referenz gelöst werden. Im Gegensatz zu üblichen starken Referenzen hält eine schwache Referenz den Garbage Collector nicht davon ab, eine Instanz zu entfernen wenn diese Referenz noch existiert. Verwaltet die Kommunikationsinfrastruktur alle Instanzen, von denen auszugehen ist, dass sie aktualisiert werden sollen (existiert eine *@SocketObjectIdentifier* Annotation an einem Attribut), dann kann sie dies in einem Verzeichnis mit schwachen Referenzen zu den Instanzen lösen, dem Instanzverzeichnis. Trifft darauf hin ein weiteres Kommunikationsobjekt mit gleichem Identifikator ein, kann zuvor in dem Verzeichnis nachgeschlagen werden, ob die Referenz noch existiert, und die Instanz hinterlegt oder aktualisiert werden.

Während die *@SocketObject* Annotation das Kommunikationsobjekt identifiziert dient die *@SocketObjectIdentifier* Annotation zum Markieren von Attri-

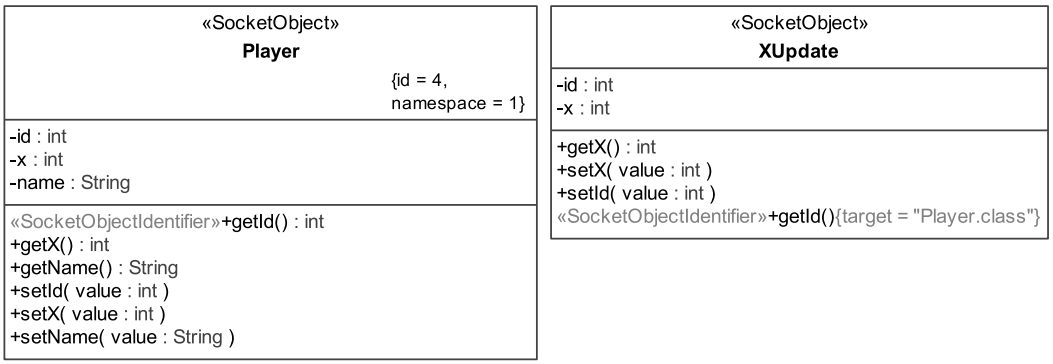



**Abbildung 6.26: Beispiel Kommunikationsobjekt *XUpdate*, welches das Attribut $x$ aus der Entität vom Typ *Player* aktualisiert. Eine Implementierung ist in Anhang J (S. 253) in Quellcode J.6 und in Quellcode J.7 zu finden.**

buten die die zugehörige Instanz identifizieren. Befindet sich die Annotation an dem zu identifizierenden Entitätstyp, muss der Ziel-Entitätstyp mittels *target*-Parameter nicht gesetzt werden, da sie implizit durch sich selbst definiert ist. Befindet sich die Annotation an einem Kommunikationsobjekt, welches sich auf einen anderen Entitätstyp bezieht, wird mit dem *target*-Parameter der jenige Entitätstyp identifiziert, welche zu aktualisieren ist. Abbildung 6.26 zeigt ein Beispiel eines Entitätstyps des Spielmodells welcher mit *@SocketObjectIdentifier* annotiert ist und ein Kommunikationsobjekt *XUpdate* welches eine Entität vom Typ *Player* des Spielmodells aktualisiert.

Empfängt der Client eine Instanz der Klasse *XUpdate* oder eine Instanz der Klasse *Player* aus Abbildung 6.26 versucht der Client einen Verweis auf eine existierende Instanz in dem Instanzverzeichnis zu finden. Existiert diese Instanz, wird jene aktualisiert indem alle Attribute des empfangenen Kommunikationsobjektes in die existierende Instanz überführt werden. Existiert keine Instanz, wird eine neue Instanz erzeugt oder die empfangene Instanz im Verzeichnis hinterlegt, falls die empfange Instanz dem repräsentierten Entitätstyp entspricht.

### 6.3.3 Plattformunabhängige entfernte Methodenaufrufe

Abbildungsformat (siehe Kapitel 6.2.2, S. 138) und Kommunikationsport (siehe Kapitel 6.2.1, S. 134) bilden die Basis für die Kommunikationsinfrastruktur

```
1 var world = new GameWorld("localhost", 12345);
2 var player;
3 world.newPlayer("Testname", function(result) {
4   player = result;
5   player.dance();
6 });
```

**Quellcode 6.4: JavaScript-Client mit methodenbasierter Serverinteraktion.**

des Spielservers, welche vom Client genutzt werden kann. Im Vergleich zur Formulierung von Key-Value-Pair (siehe Kapitel 3.4, S. 61) und dessen Austausch zwischen Client und Server, ermöglicht der Nachrichtenfluss mittels Kommunikationsobjekten eine typensichere Entwicklung ohne Notwendigkeit über Abbildungsformate nachzudenken. Darauf aufbauend ist ein weiteres Ziel der Ausführungsumgebung die Kommunikation soweit zu abstrahieren, dass Kommunikation nicht im Sinne des Austausches von Anfragen und Antworten in Form von Kommunikationsobjekten realisiert wird, sondern über den Aufruf von Methoden am Spielmodell mithilfe der im Modell verwendeten Entitäten.

Die Kommunikation mithilfe von Proxies und entfernten Methodenaufrufen ist in Java über den Standard Remote Method Invocation (RMI) [2] bekannt. Dieser bietet auf Basis einer austauschbaren Kommunikationsschicht die Möglichkeit über die Grenzen von Java VMs hinweg unidirektional Methoden aufzurufen und in die entgegengesetzte Richtung das Ergebnis zu transportieren. Aufgrund der Fokussierung von RMI auf Java-basierte Plattformen und der Art der Adressierung ist eine Abbildung der Mechanismen auf Java-fremde Plattformen nur eingeschränkt Möglich und damit im Rahmen der Ausführungsumgebung ungeeignet. Eine eigene Realisierung ist daher erforderlich.

Zentraler Mechanismus zur Realisierung von RMI ist das Proxy-Pattern. Bei einer entfernten Kommunikation mithilfe von Proxies werden Aufrufe von Methoden der implementierten Schnittstellen in der Implementierung der Verarbeitungslogik an einen Server weitergereicht, ausgeführt und das Ergebnis an den Aufrufenden zurückgegeben. Quellcode 6.4 zeigt hierfür ein Beispiel in JavaScript, wie eine Client Anwendung aussehen könnte:

In dem Quellcode 6.4 wird ein neue Proxy-Instanz für die Spielwelt erzeugt (Zeile 1) und an dieser Instanz die Methode *newPlayer* Aufgerufen um

```
1 var clientSocket = new ClientSocket("localhost", 12345);
2 var player;
3 var createPlayerRequest = new CreatePlayerRequest();
4 createPlayerRequest.setPlayerName("Testname");
5 clientSocket.send(createPlayerRequest, function(result) {
6   // Result has CreatePlayerResponse Type
7   var result = new Player();
8   result.setId(result,getPlayerId());
9   result.setPlayerName(result.getPlayerName());
10  player = result;
11  var danceRequest = new DanceRequest();
12  danceRequest.setPlayer(player.getId());
13  clientSocket.send(danceRequest);
14 });
```

**Quellcode 6.5: JavaScript-Client mit Kommunikationsobjekt basierter Serverinteraktion.**

einen Spieler zu erzeugen (Zeile 3). Da parallele Prozesse in JavaScript nicht möglich sind und blockierende Arbeitsabläufe nicht zur Verfügung stehen, ist es notwendig in JavaScript auf Callback-Methoden für die Ergebnisverarbeitung zu setzen. Callback-Methoden werden hier im direkten Anschluss zu den ursprünglichen Parameter angegeben, sind jedoch nicht obligatorisch. Ist der Spieler erzeugt, ruft der Client die Callback-Methode auf. Diese nutzt das Ergebnis um an dem Spieler die Methode *dance* auszuführen und die Instanz für spätere Verwendung bereitzulegen.

Quellcode 6.5 zeigt das selbe Beispiel auf Basis von Kommunikationsobjekten. Der Verbindungsaufbau (Zeile 1) ist identisch, die Erzeugung eines neuen Spielers (Zeile 3 bis 5) setzt die Instanziierung eines Kommunikationsobjektes voraus. Ebenso wie in Quellcode 6.4 muss in diesem Beispiel mit Callback-Methoden gearbeitet werden. In Zeile 7 bis 10 wird die Antwort verarbeitet und ein Entität vom Typ *Player* erzeugt, welches dann in Zeile 11 bis 13 dazu verwendet wird um das Tanzen zu starten.

Während ein Entwickler in Quellcode 6.5 Anfragen in speziellen Kommunikationsobjekten formuliert und die Antworten verarbeiten muss, kann bei einer methodenbasierten Interaktion auf diese zusätzlichen Instruktionen verzichtet werden. Zusätzlich ist die Kommunikation auf dieser Basis direkt aus dem Spielmodell abgeleitet und erfordert keine zusätzlichen Klassen um Anfragen zu erzeugen. Für eine Spielentwicklung ist Quellcode 6.4 bei zunehmender An-

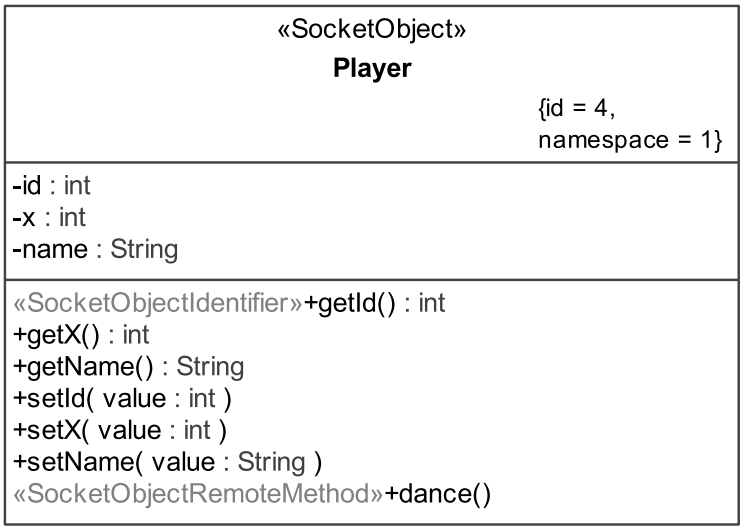



**Abbildung 6.27: Erweiterte Klasse aus Abbildung 6.26 zur Bereitstellung der Methode *dance* für das Beispiel in Abbildung 6.4. Eine Implementierung ist in Anhang J (S. 253) in Quellcode J.8 zu finden.**

wendungskomplexität einfacher zu erfassen, da erkennbar ist, welche Methode am Spielmodell im Server aufgerufen wird und keine Codeblöcke entstehen, in denen Attribute aus Kommunikationsobjekten auf lokale Entitäten (und umgekehrt) zugewiesen werden müssen.

Zur Realisierung dieses Ansatzes sind drei Schritte notwendig: die Instanzidentifizierung, die Methodenidentifizierung und Bezug der Instanz im Server, an der die Methode aufzurufen ist. Die Methodenidentifikation kann über das das jeweilige Kommunikationsobjekt realisiert werden. Für die Instanzidentifizierung muss im Kommunikationsobjekt ein Identifikator ergänzt, mit *@SocketObjectIdentifier* annotiert, mittels *target*-Parameter an den Entitätstyp im Spielmodell gekoppelt und mitgesendet werden. Für den Bezug der Instanz im Server wird über das Instanzverzeichnis (siehe Kapitel 6.3.2, S. 158) auf die reale Instanz zugegriffen.

Damit Methoden vom Client aufgerufen werden können, definiert die Ausführungsumgebung die Annotation *@SocketObjectRemoteMethod*, welche eine Methode als entfernt aufrufbar markiert. Abbildung 6.27 zeigt die Klasse aus Abbildung 6.26 und erweitert diese um die Methode *dance*, welche anschließend mittels Annotation zur Bereitstellung markiert wird.

### 6.3.4 Clientcode Generator

Kommunikationsobjekte im Server, sowie dessen Abbildung, passende Proxies und überführte Entitätstypen des Spielmodells in Clientsprache ermöglichen eine effiziente und kompakte Interaktion zwischen Server und Client. Damit dessen Implementierung konsistent und synchron mit dem in Java implementierten Spielmodell ist, greift die Ausführungsumgebung zur Bereitstellung der abgeleiteten Kommunikationsobjekte in Java und der Clientimplementierung auf einen Code Generator zurück.

Abbildung 6.28 zeigt die Transformation einer in Java implementierten Methode *foo* der Klasse *Bar* in dessen Kommunikationsobjekte, sowie die in der Clientsprache bereitgestellten Implementierung. Aus der Methodensignatur entsteht ein Kommunikationsobjekt für den Transport der Parameter und ein Kommunikationsobjekt für den Transport des Ergebnisses. Die Signatur der Methode *foo* weicht in JavaScript von ihrem Pendant in Java ab, da JavaScript keine blockierenden Methodenaufrufe anbietet. Zur Verarbeitung von zeitlich divergenten Abläufen werden in JavaScript sogenannte Callback-Handler verwendet, die der Signatur als Parameter vom Typ *Function* hinzugefügt werden und im Fall des Eintreffens der Antwort aufgerufen werden.

Für die Erzeugung von Klassen in alternativen Sprachen verwendet die GameEngine einen eigenen Generator, der die abstrakten Spracheigenschaften, dargestellt in Abbildung 6.29, nachbildet und je Zielsprache spezialisiert:

- **CodeElement**
  Abstrakter Code, wird durch die Methode *generate* Generiert.

- **CodeBlock**
  Ein Block an Anweisungen, besteht aus einer Menge von *CodeElement*.

- **Line**
  Einzelne Code-Zeile.

- **CodeBody**
  Ist ein *CodeBlock*, z. B. einer Methode, bestehend aus einem *CodeBlock*.

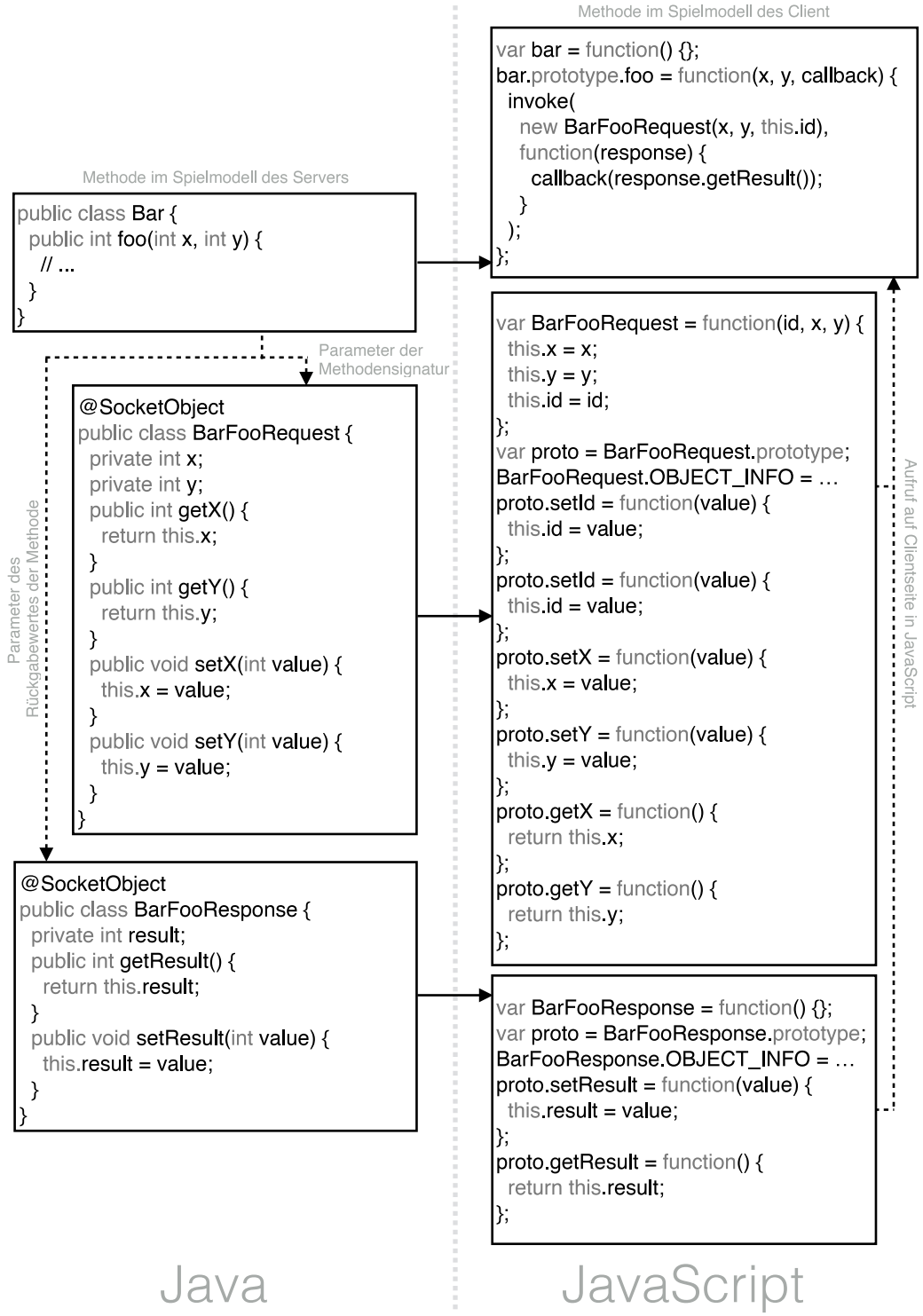



**Abbildung 6.28: Transformation von Methodensignaturen in Kommunikationsobjekte mit anschließender übersetzung in JavaScript.**

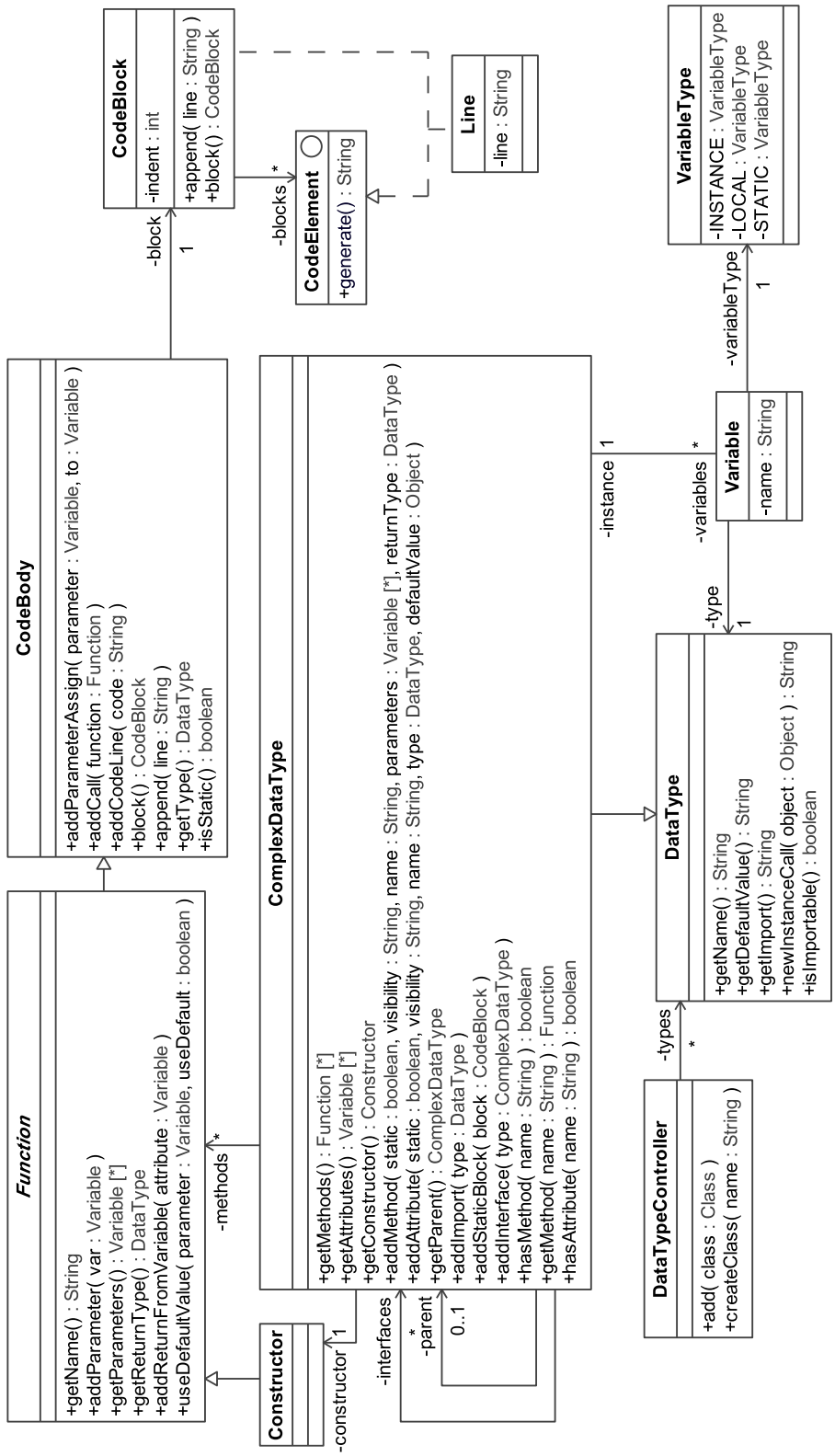



**Abbildung 6.29: Klassendiagramm über Sprachelemente zur Konstruktion der abstrakten Basis des Code Generators.**

- **Function**
  Beischreibt die Komponenten einer Methode, insbesondere Name, Parameter als Menge von *Variable*, Rückgabewert als *DateType* und Methodenkörper als *CodeBody*.

- **Constructor**
  Spezielle Funktion zur Erzeugung einer neuen Instanz.

- **Variable**
  Definiert eine Variable zur Verwendung als Parameter, Attribut oder lokale Variable.

- **DataType**
  Beschreibt einen Datentypen, i. A. Basisdatentypen.

- **ComplexDataType**
  Beschreibt komplexe Datentypen durch deren Methoden als Menge von *Function*, Attribute als Menge von *Variable*, Construktor als *Constructor*, sowie Name und Standardwert (z. B. *null*).

Basierend auf diesen abstrakten Eigenschaften werden anschließend Clientsprachen, z. B. JavaScript und ActionScript, spezialisiert. Die Spezialisierungen beschreiben wie z. B. eine Funktion mithilfe der definierten Elemente (Methoden, Variabeln, Construktor, usw.) in der jeweiligen Sprache zu formulieren ist.

## 6.4 Abstraktes Spielmodell

Aufbauend auf Kapitel 2 (S. 7) enthält die Ausführungsumgebung ein abstraktes Spielmodell zur Unterstützung der Entwicklung von browserbasierten MMOGs. Das Spielmodell stellt die Grundelemente Spielwelt, Spielobjekte und Aktionen in Form einer API für die Implementierung von Spielmodellen zur Verfügung.

Die primären Entitätstypen des abstrakten Spielmodells sind jene, welche bereits in Kapitel 2.6 (S. 14) als Bestandteile eines Spielmodells identifiziert

wurden. Dies umfasst eine abstrakte Spielwelt *AbstractWorld* sowie die abstrakten Spielobjekt *AbstractGameObject*. Spielwelt und Spielobjekte implementieren eine Schnittstelle welche die Behandlung von Ereignissen ermöglicht und spezialisieren eine Verarbeitungslogik dieser Ereignisbehandlung. Die Schnittstelle basiert auf dem Observer-Pattern erweitert dieses Pattern um Zeichenketten-basierte Filtermöglichkeiten, damit die Auswahl der zubearbeitenden Beobachter effizient möglich ist.

Die Implementierung des Observer-Patterns ermöglicht die Konfiguration einer Entität-behafteten oder Entitätstyp-behafteten Beobachterverarbeitung. Bei der Entität-behafteten Verarbeitung werden Ereignisse an einer Entität ausgelöst und Beobachter ausgeführt, die an dieser Entität auf Ereignisse warten. In der Entitätstyp-behafteten Verarbeitung werden Beobachter am Entitätstyp registriert. Wird ein Ereignis ausgelöst, werden alle Beobachter die auf Ereignisse dieses Entitätstyp warten informiert und erhalten im Rahmen der Verarbeitung die betroffene Entität als Parameter übergeben.

Abbildung 6.30 visualisiert das abstrakte Spielmodell in einem Klassendiagramm. Die Darstellung führt die verschiedenen Realisierungen der vorangegangenen Kapitel für das abstrakte Spielmodell zusammen. Das Klassendiagramm enthält die bereits genannten Klasse *AbstractWorld* und *AbstractGameObject*, sowie die Realisierung des Observer-Patterns und die bereits beschriebenen *Action*, welche durch den *Timer* und die *TimerQueue* verwaltet und ausgeführt wird:

- **GameElement**
  Schnittstelle zur Definition von Methodensignaturen für den Lebenszyklus eines Spielobjektes, sowie dessen Identifikator.

- **GameObject**
  Schnittstelle zur Definition von Methodensignaturen für Hilfsmethoden zum Erzeugen von Aktionen die im Kontext mit diesem Spielobjekt stehen. Die Schnittstelle spezialisiert die Schnittstelle *GameElement*.

- **AbstractGameObject**
  Abstrakter Entitätstyp als Basis zur Erzeugung von Spielobjekten bei der Entwicklung. Sie implementiert die durch die Schnittstellen *GameObject*

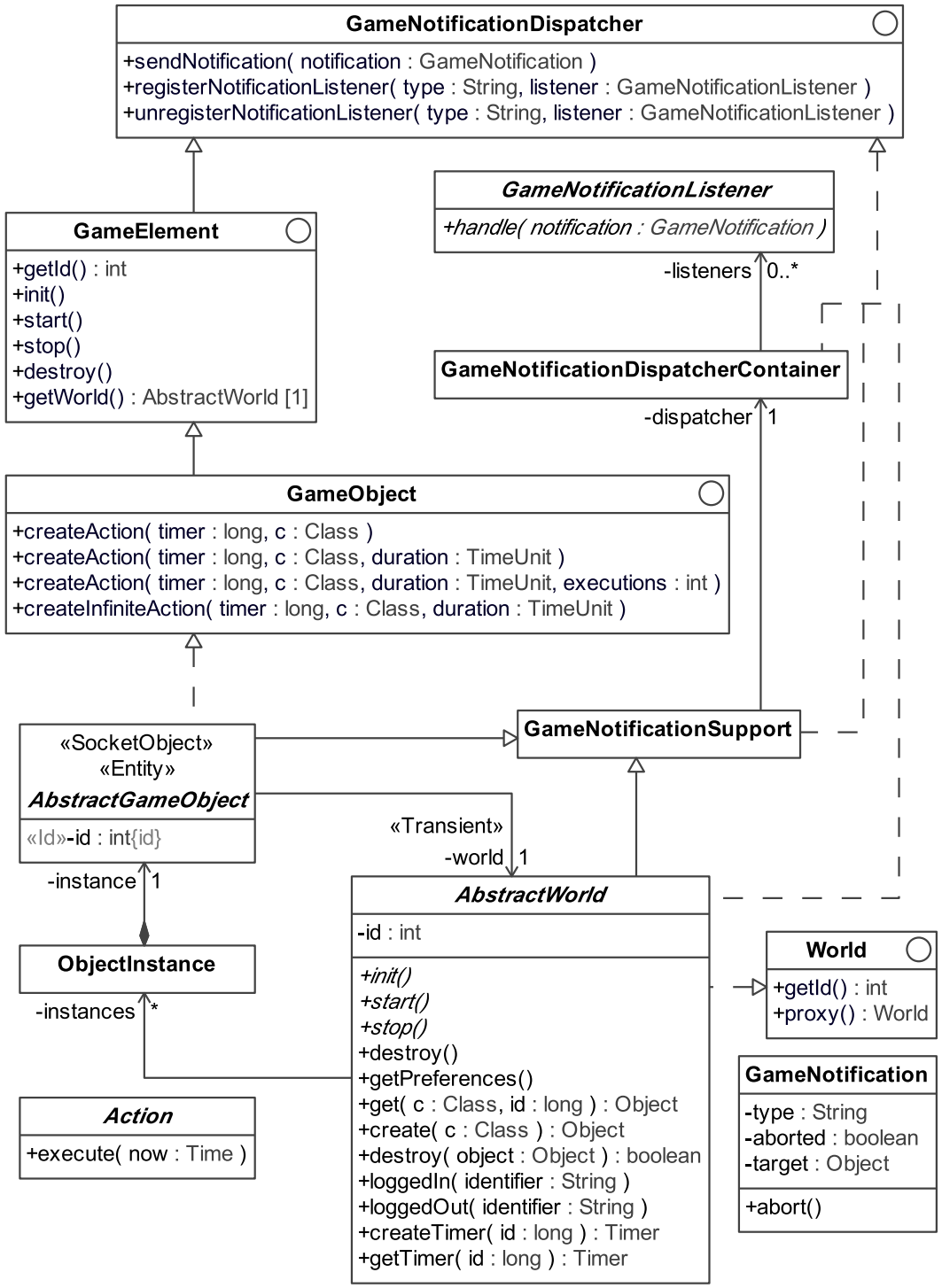



**Abbildung 6.30: Klassendiagramm des abstrakten Spielmodells zur Unterstützung der Entwicklung.**

und *GameElement* definierten Methoden. Der Entitätstyp ist für eine mögliche Kommunikation annotiert, Spezialisierungen müssen jedoch die *@SocketObject* Annotation ergänzen damit Entitäten dieses Typs versendet werden können.

- **GameNotificationDispatcher**
  Adaption der Schnittstelle *Observable* des Observer-Patterns. Die Signaturen zum Hinzufügen und Entfernen ermöglichen abweichend die Angabe eines Observer-Typisierers der die Auswahl von Beobachtern vereinfacht.

- **GameNotificationListener**
  Adaption der Schnittstelle *Observer* des Observer-Pattern.

- **GameNotification**
  Ereignis-Typ, welcher im Fall von Ereignissen an Beobachter übergeben wird. In der Regel kann dieser Typ spezialisiert werden um weitere Daten bezüglich des Ereignisses zu transportieren.

- **GameNotificationDispatcherContainer**
  Implementiert eine Verwaltung von registrierten Beobachtern und informiert die betroffenen Beobachter im Fall des Aufrufes von *sendNotification*.

- **GameNotificationSupport**
  Wrapper des *GameNotificationDispatcherContainer* zur Realisierung der Konfigurierbarkeit von Entität- und Entitätstyp-behafteter Ereignisverabeitung. Je nachdem welche Strategie verwendet wird, wird entweder je Entität oder je Entitätstyp ein Container erstellt.

- **World**
  Schnittstelle zur Definition von Methodensignaturen für die abstrakte Spielwelt.

- **AbstractWorld**
  Abstrakter Entitätstyp als Basis zur Erzeugung von Spielwelten. Die

Welt erfordert die Implementierung der abstrakten Methoden zum Lebenszyklus und bietet Möglichkeiten zum Erzeugen, Beziehen und Zerstören von Spielobjekten.

- **Action**
  Elemente des Timers, siehe Kapitel 6.1.4 (S. 131).

## 6.5 Laufzeitsicht

Die vorangegangenen Kapitel haben die Funktionsweisen der Komponenten der Ausführungsumgebung skizziert. Im Folgenden werden diese Ergebnisse zusammengeführt um deren Interaktion in drei Anwendungsfällen dargestellt. Dabei wird abgebildet, wie die Ausführungsumgebung startet, eine Client eine Verbindung zu einer Spielwelt aufbaut und eine Funktionalität der verbundenen Spielwelt aufgerufen. Zur Visualisierung der Zusammenhänge wird auf Sequenzdiagramme zurückgegriffen, welche den Vorgang abstrahieren. Eine vollständige Darstellung des Aufruf-Stacks gefährdet die Übersichtlichkeit der Visualisierung.

### 6.5.1 Start der Ausführungsumgebung

Wird die Ausführungsumgebung für Spielmodelle durch deren umgebene Ausführungsumgebung gestartet, z. B. durch eine Java EE Ausführungsumgebung, wird der Lebenszyklus der MBeans durch jene übergeordnete Ausführungsumgebung durchlaufen. Der Durchlauf beginnt nach der Instanzierung der Klasse *GameController* durch den Aufruf der Methode *create* und wird durch den Aufruf von *start* vollendet.

Im Rahmen des Aufrufs von *create*, siehe Abbildung 6.31, werden im *GameController* die durch ihn zu verwaltenden Spielwelten identifiziert. Dies kann zum einen durch die Suche nach Entitätstypen, welche mit *@GameWorld* annotiert sind, als auch durch Verarbeitung der Konfiguration der Ausführungsumgebung durchgeführt werden. Sind die Spielwelten identifiziert, werden diese in einem iterativen Prozess sequenziell instanziiert. Jede Spielwelt erhält hierbei ihre eigene Kommunikationsinfrastruktur in Form einer Instanz der Klasse *SocketAdapter*.

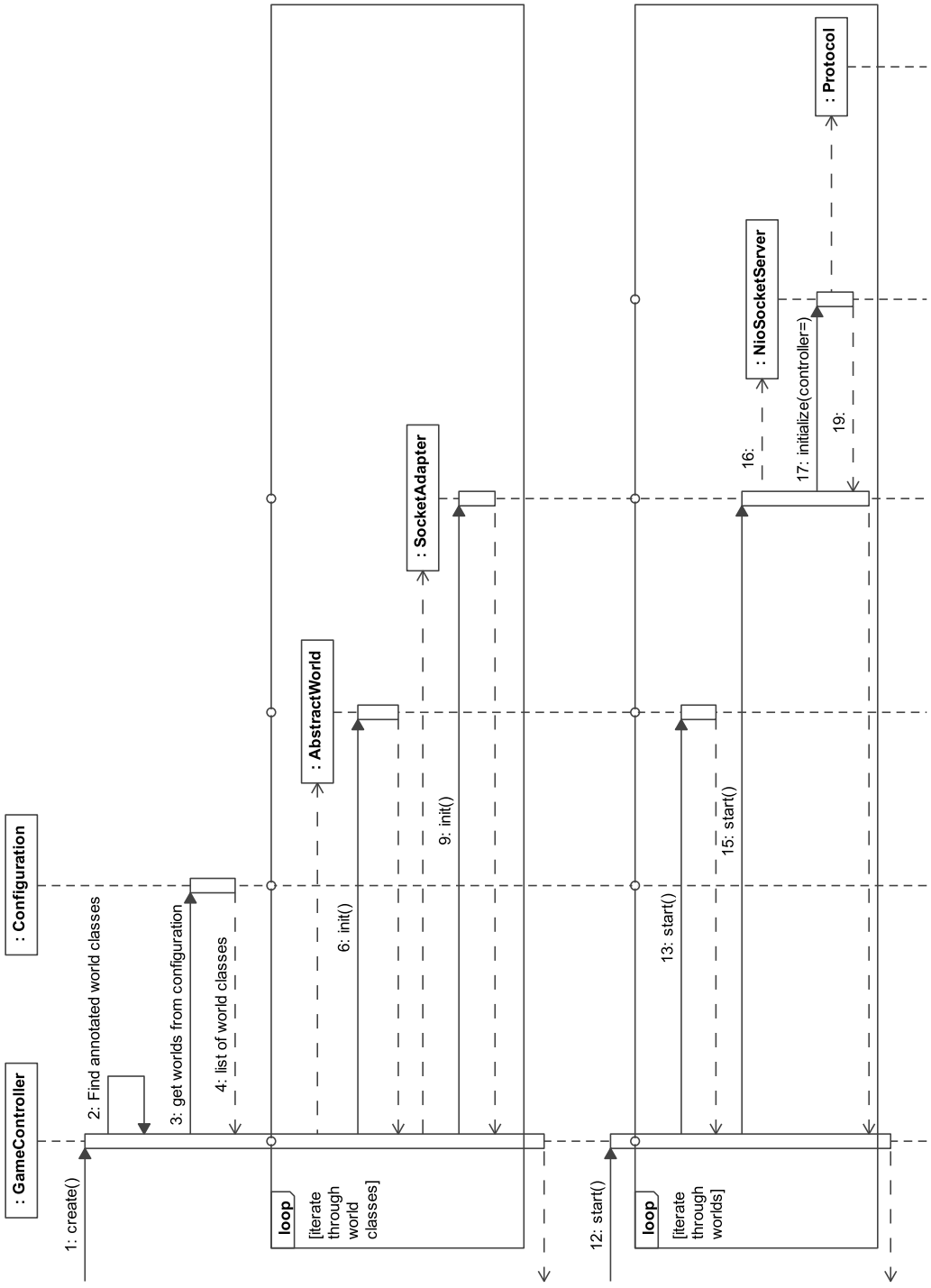



**Abbildung 6.31: Sequenzdiagramm zum Ablauf des Starts der Ausführungsumgebung.**

Sind alle Spielwelten und korrespondierende *SocketAdapter* durch den Aufruf von *create* erzeugt, werden diese im Anschluss durch den Aufruf von *start*, siehe Abbildung 6.31, bereitgestellt. Mit dem Aufruf der Methode *start* wird erneut über alle Spielwelten und Instanzen der Klasse *SocketAdapter* iteriert. Im Fall des *SocketAdapter* wird beim Aufruf von *start* die Implementierung der Schnittstelle *SocketServer*, z. B. der *NioSocketServer*, erzeugt und bereitgestellt, sowie das Abbildungsformat in Form einer Instanz der Klasse *Protocol* erzeugt.

### 6.5.2 Verbinden, Empfangen und Senden

Das Sequenzdiagramm in Abbildung 6.32 stellt den Prozess zum Verbindungsaufbau eines neuen Clients, sowie den Empfangen und Senden von Daten basierend auf der Kommunikationsinfrastruktur und dem Abbildungsformat dar. Die Einstiegsmethode zur Realisierung ist *execute* in der Klasse *NioSocketServer*, welche solange aufgerufen wird, wie die Ausführungsumgebung bereitgestellt wird. Der Prozess beginnt mit dem Beziehen von Ereignissen über eine Java API. Dieser Bezug stellt zu verarbeitende Datenpakete und neue Verbindungsanfragen in einer Liste bereit, welche im Folgenden in einem iterativen Vorgang bearbeitet werden kann. Je Iterationsschritt ist dabei zu prüfen, ob es sich bei dem Ereignis um einen Verbindungsaufbau, das Versenden von Datenpaketen oder ein empfangenes Datenpaket handelt.

Im Fall des Verbindungsaufbaus wird für die einzelne Verbindung eine neue Instanz der Klasse *NioSocketConnection* erstellt. Diese verwaltet einen exklusiven Dateneingangspuffer und lässt sich vom *SocketController*, welcher über den *NioSocketServer* bezogen wird, einen neuen *RequestDispatcher* bereitstellen.

Ist die *NioSocketConnection* und der *RequestDispatcher* erzeugt, können Daten empfangen und gesendet werden, basierend auf den im *NioSocketServer* entstehenden Ereignissen. Wird ein Empfangs-Ereignis vom *NioSocketServer* verarbeitet, wird die *NioSocketConnection* beauftragt, die Daten einzulesen, mit der Instanz von *Protocol* zu interpretieren und anschließen mittels *RequestDispatcher* zu verarbeiten. Abgeschlossen wird das Verarbeiten von empfangenen Daten durch den Versand von resultierenden Antworten.

Wird ein Senden-Ereignis vom *NioSocketServer* erfasst, welches immer dann erzeugt wird, wenn in der *NioSocketConnection* Kommunikationsobjekte zum

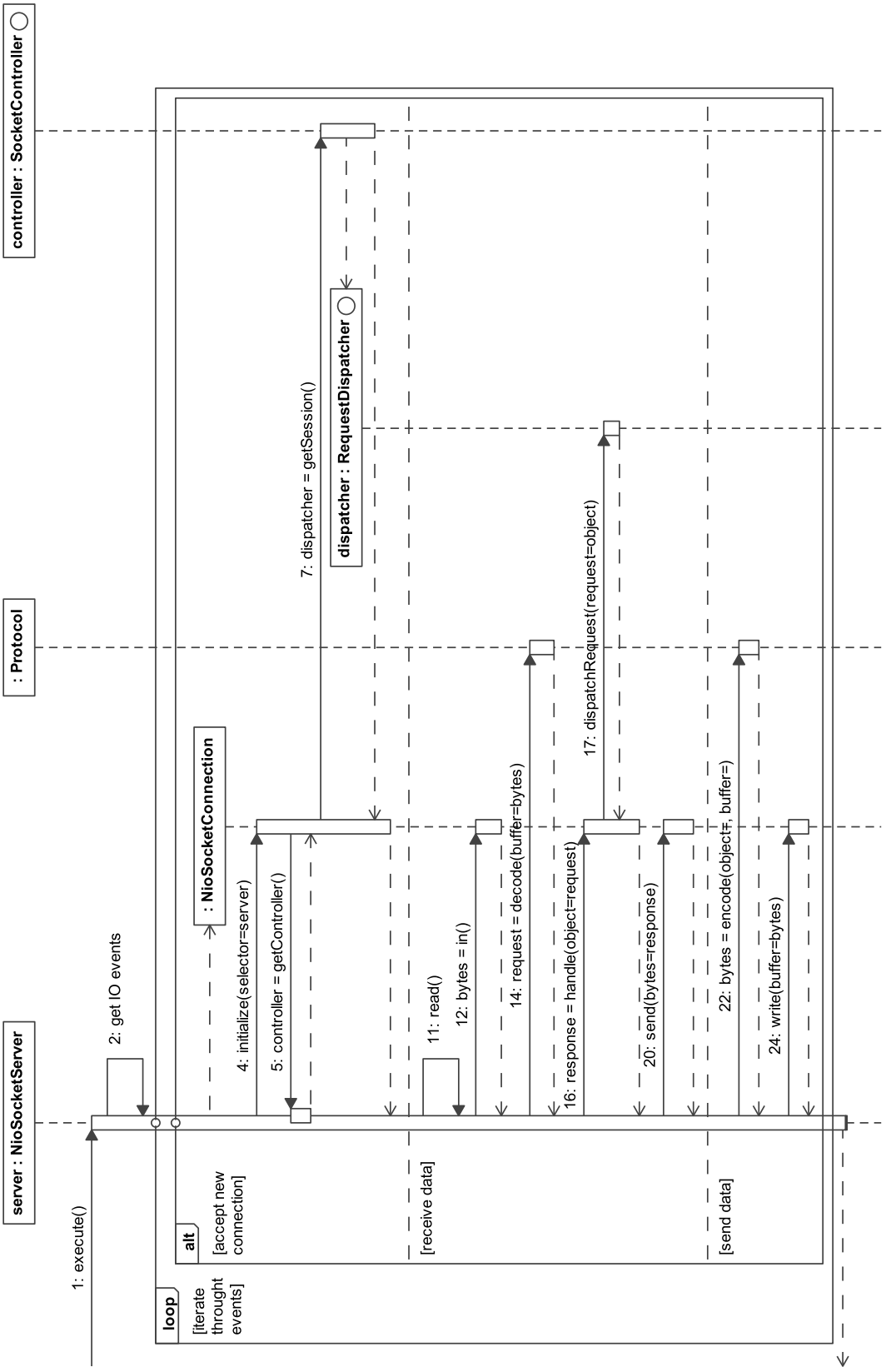



**Abbildung 6.32: Sequenzdiagramm zum Ablauf des Verbindungsaufbaus, Empfangen und Senden von Kommunikationsobjekten.**

versenden hinterlegt wurden, werden die zuversendenden Kommunikationsobjekte mit dem *Protocol* in einem Datenpaket abgebildet und anschließend an den Client übermittelt.

### 6.5.3 Aufruf einer Funktionalität

Die dritte Betrachtung der Laufzeitsicht durchläuft den Prozess zum Aufruf einer Funktionalität, dargestellt in Abbildung 6.33. Empfängt die *NioSocketConnection* Daten, werden diese, nach der initialen Verarbeitung durch die Komponente *Server Message Handler*, an die Komponente *Request Handling* übergeben und in einer Anfrageverarbeitungsroutine bearbeitet. Diese Bearbeitung interagiert mit der Instanz der Spielwelt über einen Proxy auf Spielwelt und Spielobjekte, Abbildung 6.33 demonstriert dies im Fall der Spielwelt, welcher nach erfolgreichem Aufruf den geänderten Zustand der Instanz persistiert.

## 6.6 Beispiel-Server mit der Ausführungsumgebung

In Anlehnung an die implementierten Beispiele in Kapitel 3 (S. 31) wird im Folgenden die Ausführungsumgebung zur Realisierung des minimale Beispiels aus Abbildung 2.2 genutzt. Abbildung 6.34 zeigt die am Modell erfolgten Anpassungen, wie die ergänzten Vererbungen und die notwendigen Annotationen, welche im Klassendiagramm mittels Stereotypen deklariert sind. Hierfür wurde die Spielwelt *BasicWorld* mit *@GameWorld* annotiert, um es als zentrales Weltelement zu markieren, und erbt von dem abstrakten Entitätstyp *AbstractGameWorld* um grundlegende Funktionalitäten zu übernehmen. Das Spielobjekt *Avatar* wird mit *@SocketObject* annotiert, um es übertragbar zu markieren, und erbt von dem abstrakten Entitätstyp *AbstractGameObject* um grundlegende Funktionalitäten als Spielobjekt zu erhalten. Der Entitätstyp *Position* wird innerhalb des Beispiels genutzt um eine Position zu beschreiben, ist selbst jedoch kein Spielobjekt. Da Entität dieses Typs an den Client transportierbar sein soll, erhält dieser Typ wie der *Avatar* die Annotation *@SocketObject*.

Zur Bereitstellung der fünf Funktionalitäten werden die korrespondierenden

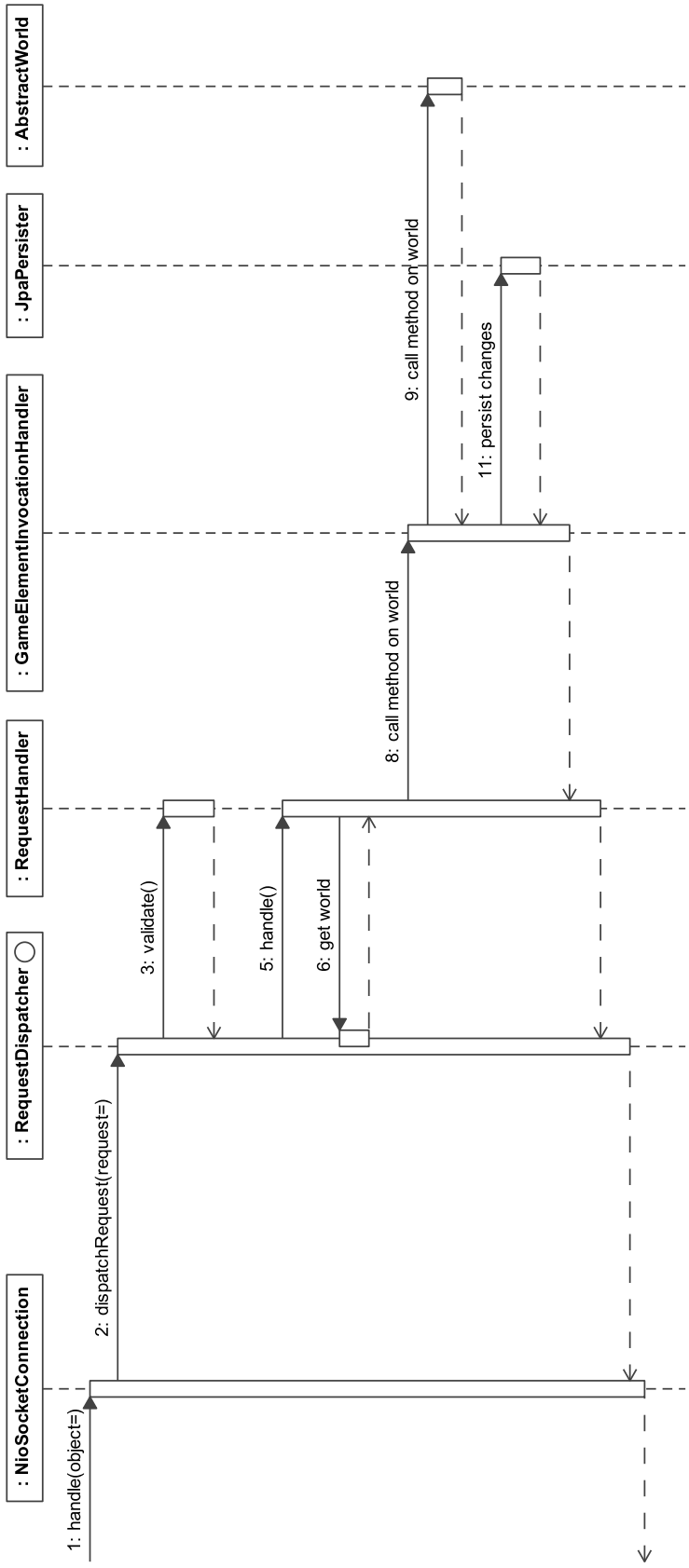



**Abbildung 6.33: Sequenzdiagramm zum Ablauf des Aufrufs einer Funktionalität.**

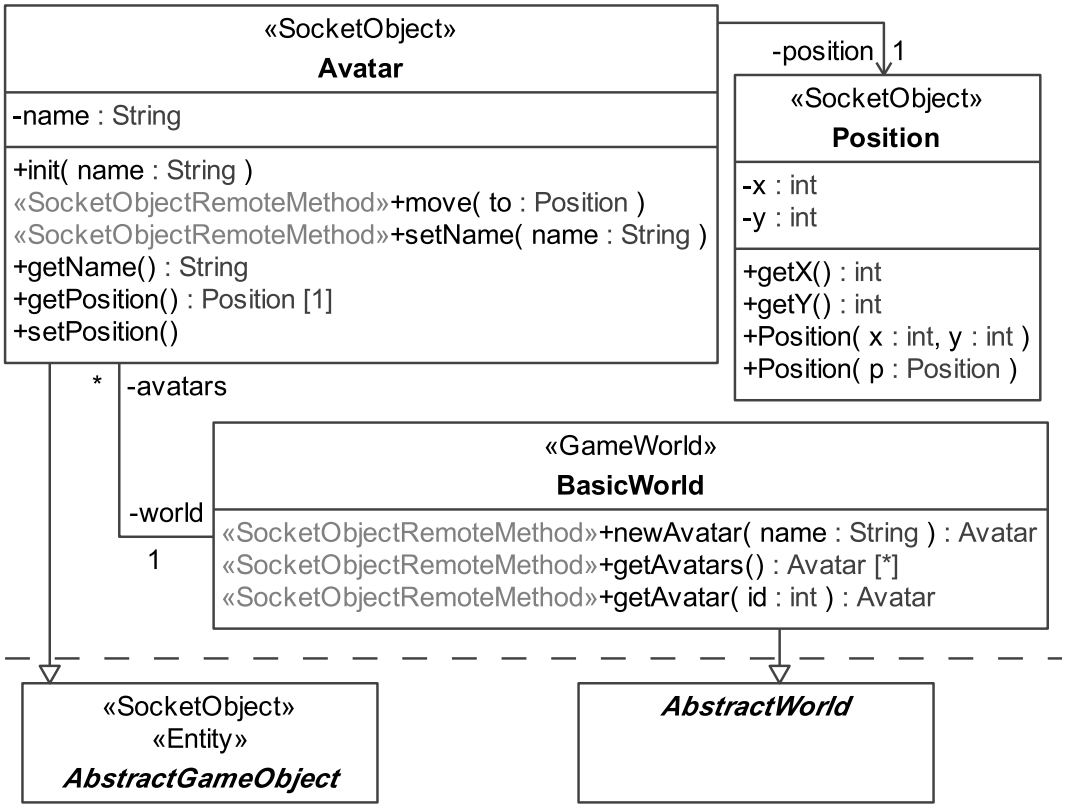



**Abbildung 6.34: Klassendiagramm für die Beispiel-Implementierung des minimalen Spielmodells aus Abbildung 2.2 in der Ausführungsumgebung.**

Methoden im Modell mit der Annotation *@SocketObjectRemoteMethod* markiert. Die bereitzustellenden Funktionalitäten leiten sich aus der Schnittstelle in Abbildung 2.8 ab: *newAvatar*, *getAvatar*, *getAvatars*, *moveAvatar* und *updateAvatar*.

Die zusätzlichen Vererbungen und Annotationen ignorierend, gleicht die Implementierung des Spielmodells derer welche in Kapitel 3.3.3 (S. 59), Kapitel 3.2.2 (S. 46) und Kapitel 3.4.1 (S. 61) zum Einsatz gekommen sind. Die Modifikationen begrenzen sich auf die gezeigten Anpassungen und sind ausreichend um das Spielmodell für die Ausführungsumgebung aufzubereiten.

# 7. Evaluation

Basierend auf dem Architekturentwurf (siehe Kapitel 5, S. 93) wurde die Ausführungsumgebung (siehe Kapitel 6, S. 123) realisiert und wird im Folgenden genauer betrachtet. Im Rahmen dieser Betrachtung werden verschiedene Spielprojekte, die mit der realisierten Ausführungsumgebung umgesetzt wurden, unter verschiedenen Aspekten bewertet. Die zur Verfügung stehenden Spielprojekte variieren im eingesetzten Entwicklungsstand der Ausführungsumgebung. Somit ergibt sich zum Beispiel, dass die Möglichkeit zur Verwendung von entfernten Methodenaufruf (siehe Kapitel 6.3.3, S. 160) nicht für alle Projekte verfügbar ist. Alternativ lässt sich jedoch eine Betrachtung der entstandenen Anfrageverarbeitungen ableiten, und deren Einsatzzweck betrachten.

Für die Analyse der Spielprojekte werden zwei Betrachtungen durchgeführt. Die erste führt eine allgemeine Betrachtung der Spielprojekte mittels Reverse Engineerings durch um diese anschließend einem Vergleich der intendierten und resultierenden Modelle zu unterziehen. Ziel ist es, eine Aussage zur These 1 zu erhalten. Die zweite Betrachtung wird hinsichtlich der These 4a durchgeführt. Zu erwarten ist dabei, dass sich die Menge der Anfrageverarbeitungen in einem realisierten Spielkonzept signifikant von der Menge der entfernten Methodenaufrufen in einem realisierten Spielkonzept unterscheidet. Diese kommt für Spielprojekte in betracht, wenn für die Realisierung des Konzepts die Anwendung von Anfrageverarbeitungen und entfernten Methodenaufrufen möglich ist. Ziel ist es, bezüglich der These 4a eine Betrachtung durchzuführen, ob eine Spielentwicklung ohne direkte Anwendung eines spezifischen Kommunikationsprotokoll realisierbar ist. Dabei wird angenommen, dass die Anwendung ohne Kommunikationsprotokoll möglich ist, wenn der unterschied signifikant ist und dieser signifikante Unterschied durch die höhere Menge an Funktionalitäten, die durch die Ausführungsumgebung automatisiert ableitet wird und vom Client mittels entfernten Methodenaufrufe genutzt wird, entsteht. Sind

in einem Spielkonzept ausschließlich manuelle Anfrageverarbeitungen möglich, kann alternativ betrachtet werden, ob die Menge an Anfrageverarbeitung zur Realisierung von Funktionalität auf einen signifikanten Anteil an Anfrageverarbeitung zur Realisierung von Funktionalität die das Spielmodell aufrufen, zurückzuführen sind. Die Ansätze für die Beispiel-Implementierung in Kapitel 3 (S. 31) zeigen, dass die Kommunikation zwischen Client und Server zur Bereitstellung eines Spielmodells insbesondere die Realisierung von Funktionsaufrufen auf Entitäten des Spielmodells darstellen. Es ist daher davon auszugehen, dass diese, wenn sie in einem Konzept manuell implementiert werden, auf Modellaufrufe zurückzuführen sind. Ergänzend ist davon auszugehen, dass bei ausreichender Komplexität zusätzliche Aufrufe entstehen, welche die in Kapitel 2.8.3 (S. 26) skizzierten Dienste adressieren. Ist in einem Konzept mit manueller Anfrageverarbeitung ein signifikanter Anteil dieser Verarbeitungen auf Funktionsaufrufe auf Entitäten des Spielmodells zurückführbar, dann kann angenommen werden, dass diese Verarbeitungen ebenso mittels entfernten Methodenaufrufen realisierbar wären.

Für die Analyse des Aufwands eines Spielkonzeptes, welches in mehreren alternativen Strategien realisiert ist, wird betrachtet, ob der Entwicklungsaufwand in der Realisierung einer Funktionalität mit der Ausführungsumgebung sich signifikant von dem Entwicklungsaufwand in der Realisierung einer Funktionalität mit alternativen Strategien unterscheidet. Es ist davon auszugehen, dass insbesondere durch die hohe Abstraktion der Ausführungsumgebung, als auch bezüglich der Vermeidung des Nachrichtenaustausch im herkömmlichen Sinn, der Mehraufwand in der Implementierung eines Spiels mit der Ausführungsumgebung minimal ausfällt. Im Fall der Beispiel-Implementierung lag das Verhältnis bei durchschnittlich 1:3,01. Mit dieser Betrachtung wird ein Rückschluss auf die These 2 intendiert.

Kapitel 3 (S. 31) gibt einen Einblick in Abbildungsformate, welche aktuell verfügbar sind und im Umfeld von Spielprojekten angewendet werden. Die These 4b zielt dabei auf einen Architekturanwendungsfall, in dem in der Regel Client und Server synchron entwickelt und publiziert werden, und somit der Transport von zusätzlichen Strukturinformationen obsolet ist. Die Ausführungsumgebung realisiert ein solches Abbildungsformat und nutzt dieses zum Austausch von Nachrichten. Im Rahmen der Betrachtung wird geprüft, ob

sich das Datenvolumen siginifikant vom Datenvolumen standardisierter Abbildungsformante (wie XML und JSON) unterscheidet, sowie die Ausführungszeit zur Serialisierung und Deserialisierung sich nicht signifikant von der Ausführungszeit effizienter Verfahren (wie ProtoBuf und JOS) unterscheidet.

Die These 3 lässt sich implizit überprüfen, insofern die Spielkonzepte ausführbar sind, insbesondere dem JBoss. Architekturbedingt ist hierfür keine weiterführende Betrachtung oder Analyse notwendig.

Im Folgenden wird ein detailierter Einblick in die realisierten Spielkonzepte gegeben, bevor auf die jeweiligen Analysen eingegangen wird. Der Einblick in die Spielkonzepte greift dabei einzelne Merkmale auf, welche im Folgenden zur Betrachtung herangezogen werden. Die folgenden Analysen betrachten die eigentliche Realisierung, sowie den Aufwand bei der Anwendung der Ausführungsumgebung und das Abbildungsformat im Vergleich zu alternativen Formaten im Detail.

## 7.1 Spielentwicklung

Die resultierende Ausführungsumgebung aus Kapitel 6 (S. 123) findet in unterschiedlichen Spielprojekten Anwendung. Darunter fällt das initial skizzierte Spielkonzept Viechers, der Demonstrator Stickman, sowie vier Studentenprojekte, welche in einem kooridinierten Versuchsaufbau im Rahmen von Lehrveranstaltungen entstanden.

Die Projekte differenzieren sich in Komplexität und Zeitpunkt der Anwendung der Ausführungsumgebung, sowie realisierenden Entwicklern und verwendeter Client Technologie. Im Folgenden werden die Projekte kurz vorgestellt und hinsichtlich relevanter Merkmale betrachtet, die in den folgenden Analysen genutzt werden können. Zu diesen Merkmalen zählen Entwicklungsstand der Ausführungsumgebung und Anzahl der Spielobjekte innerhalb der Spielwelt, sowie notwendige Kommunikationsobjekte, Anfrageverarbeitungen, entfernte Methodenaufrufe und ein Eindruck in den Umfang des Projektes in Form von LLoC des Spielmodells. Zusätzlich werden die Spiele im Sinn eines Reverse Engineerings in ein Klassendiagramm ohne Attributen und Methodensignaturen überführt um einen Einblick in die Generalisierungen und Assoziationen des realisierten Spielmodells im Bezug zu einem EGM zu erhalten.

### 7.1.1 Viechers

Viechers, wie in der Einleitung bereits skizziert, ist ein evolutionsbasiertes Aufbaustrategie MMOG bei dem der Spiele eine Menge von Einheiten in einer isometrischen dargestellten Landschaft koordiniert und deren Evolution kontrolliert. Durch den Bezug von Rohstoffen, Vermehrung und Expansion steigert der Spieler seinen Einfluss hin zu einer dominanten Rasse auf Pangea. Die in Herden zusammengefassten Einheiten können Forschen, Kämpfen, Verteidigen und Gebäude bauen um die Entwicklung der eigenen Herde voranzubringen. Die Spielwelt verwendet eine zweidimensionale begrenzte Karte und erlaubt die kontinuierlich stetige Positionierung von Entitäten. Jede Instanz einer Spielwelt bietet ausreichend Fläche um mehreren tausend Herden Möglichkeiten zur Expansions zu bieten.

Einen Einblick in das Spielmodell dieses Projektes liefert das Klassendiagramm in Abbildung 7.1. Dieses Diagramm abstrahiert die implementierten 2.300 Entitätstypen um die Verständlichkeit zu erhalten und reduziert die Darstellung auf wesentliche Strukturen und Abhängigkeiten. Ziel der Darstellung ist es die Zusammenhänge abzubilden, sowie die im Sinne des abstrakten Spielmodells definierten Elemente Spielwelt, Spielobjekte, Spielaktionen, Zeitgeber und Ereignisse. Die Implementierung des Modells verwendet zusätzlich Schnittstellen zu Elementen wie Haufen *Bunch*, Einheiten *Unit*, Herden *Herd* und Gebäuden *Building* um deren Verwaltung über Proxies zu realisieren. Die Erzeugung von Proxies aus Klassen stand im Rahmen dieser Implementierung nicht zur Verfügung. Einige wichtige Klassen des Diagramms sind:

- **Mutation**
  Beschreibt eine Mutation, z. B. ein Fuß oder ein Variante eines Auges.

- **Mutations**
  Ansammlung von Mutationen, welche in Kombination einen Organismus ergeben.

- **Generation**
  Spezialisierung von *Mutations* welche in einer Generationssequenz einer Herde platziert werden.

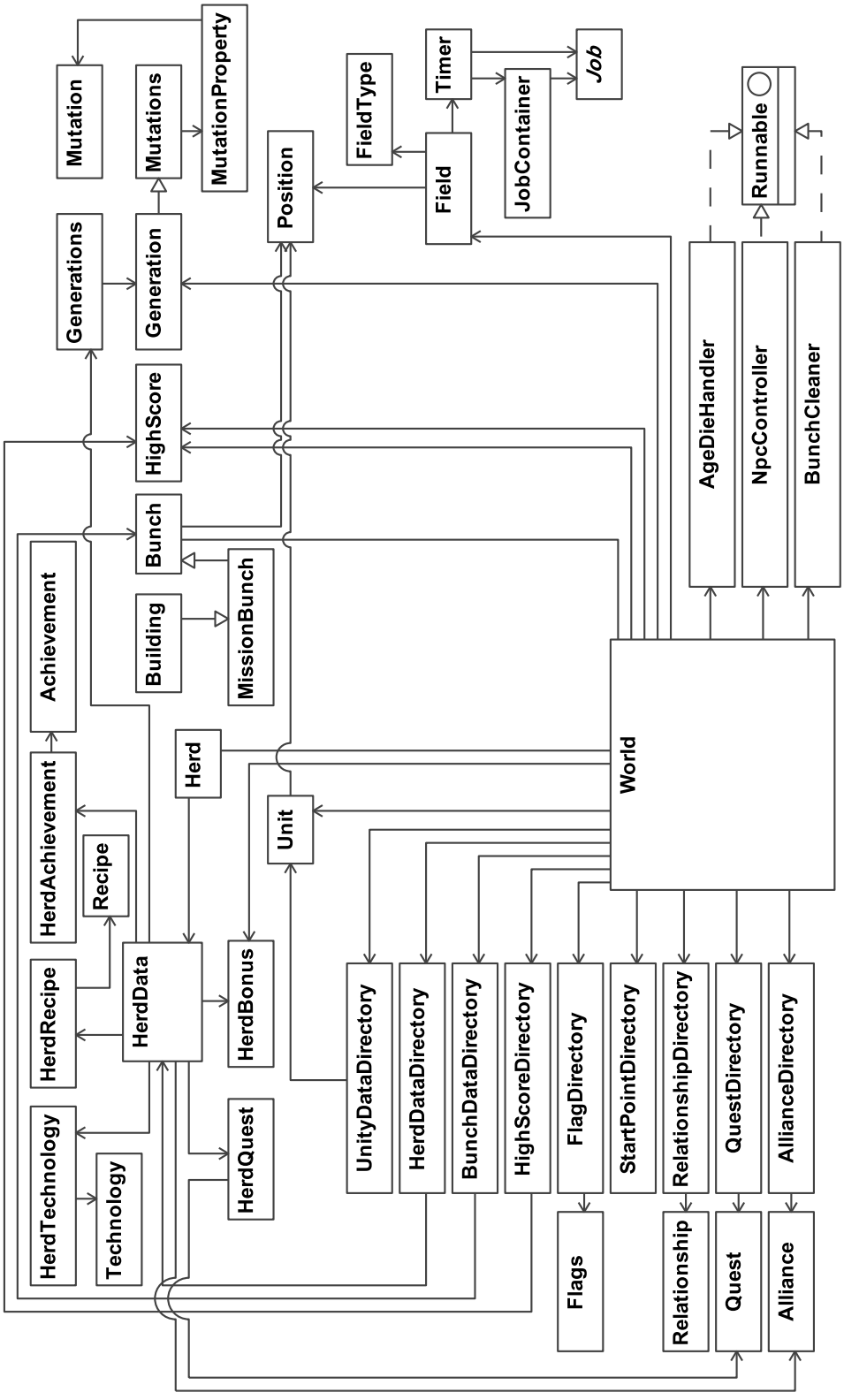



**Abbildung 7.1: Abstrahiertes Klassendiagramm des Spielmodells von Viechers. *Herd*, *Bunch*, *Mission*, *Building*, *Unit*, *Generation*, *Generations*, *Mutations*, *Field*, *Alliance*, *Quest* und *Relationship* stellen die 12 primären Spielobjekte des Modells dar.**

- **Generations**
  Sequenz von Generationen einer Herde. Stellt die Entwicklung vom Beginn bis zum aktuellen Zustand der Herde dar.

- **Unit**
  Beschreibt eine einzelne Einheit mit spezifischen Merkmalen wie Position, Geschlecht, Phänotypus und Name, insbesondere gehört jede Einheit einer Generation an.

- **Bunch**
  Einfachste Form eines Objektes in der Landschaft, z. B. ein Haufen von Rohstoffen. Ein Objekt vom Typ *Bunch* hat keinen Besitzer.

- **Mission**
  Spezialisierte Form des Entitätstyps *Bunch*, welcher die Möglichkeit bietet, dass Einheiten in ihr temporär Arbeiten durchführen können. Häufige Anwendung sind z. B. Missionen zum Sammeln von Rohstoffen. Befindet sich keine Einheit in einer Mission, wird das Objekt zerstört.

- **Building**
  Spezialisierte Form des Entitätstyps *Mission*, bei dem im Fall des Leerzustandes das Objekt in der Landschaft erhalten bleibt. Dieser Entitätstyp wird insbesondere für Gebäude wie z. B. Nester zum Wohnen und Lagern von Rohstoffen genutzt.

- **Herd und HerdData**
  Beschreibt die Herde eines Spielers und besteht aus einer Menge von Entitäten, wie *Mission* und *Building*, sowie Einheiten und einen Generationsverlauf. Zusätzlich kapselt die Herde, bzw. der Entitätstyp *HerdData* weitere Informationen bzgl. erforschter Technologien, Rezepte zum Bauen, bearbeitete Aufgaben und erreichte Errungenschaften. Die kontrollierbaren Einheiten der Herde sind nicht auf die Generationen der Herde begrenzt. Zum Beispiel ist das Einfangen von Einheiten anderer Spieler im Sinne des domestizierens möglich.

- **Field**
  Beschreibt ein Feld innerhalb der Welt mit einer Kantenlänge von 10x10

Meter. Ein Feld hat einen Typ, welchen die Flora und das Aussehen der Karte in Wüsten, Taigas, Wälder, Wiesen und Seen differenziert.

- **World**
  Beschreibt die Spielwelt unter Verwendung verschiedener Verzeichnisse die die Entitäten organisieren.

Viechers ist parallel zur Ausführungsumgebung entstanden und diente als Quelle für Anforderungen, sowie als Umgebung zur praktischen Anwendung der Ausführungsumgebung und deren Ansätze zur Unterstützung der Entwicklung. Das in Java implementierte Spielmodell arbeitet ohne Ableitung von Kommunikationsinfrastruktur aus dem Spielmodell, die in diesem Anwendungsfall anfallenden 306 Kommunikationsobjekte und 119 Anfrageverarbeitungen sind manuell implementiert. Genutzt wird in diesem Anwendungsfall die Nachrichtenverarbeitung (siehe Kapitel 6.2.1, S. 134), das Abbildungsformat (siehe Kapitel 6.2.2, S. 138) und die Anfrageverarbeitung (siehe Kapitel 6.2.3, S. 153), sowie eine erste Version eines abstrakten Spielmodells und der Clientcode-Generator (siehe Kapitel 6.3.4, S. 164) zur Erzeugung von Kommunikationsobjekten in ActionScript 3 für die Client-Implementierung, basierend auf den manuell in Java implementierten Anfragen und Antworten.

Das 30.705 LLoC große Spielmodell zeigt eine breite Vielfalt von Ereignissen und Aktionen im Sinne des abstrakten Spielmodells, sowie eine hohe Varianz an Anfragen und Kommunikationsobjekten die wiederkehrende Aufgaben identifizieren lassen und letztlich in die automatisierte Client / Server Synchronisation, sowie die plattformunabhängigen Methodenaufrufe eingeflossen sind. Das insgesammt ca. 150.000 LoC große Projekt wurde unter realen Bedingungen mit ca. 4.000 Nutzern erprobt um Abbildungsformat und Serverimplementierung in einer gängigen Ausführungsumgebung, dem JBoss 5, zu betreiben.

### 7.1.2 Stickman

Stickman ist ein 2D Jump'N'Run Adventure. Das Spiel demonstriert die Realisierung eines Projektes auf Basis der aktuellen Version der Umgebung unter Verwendung von automatisierter Client / Server Synchronisation und plattformunabhängigen entfernten Methodenaufrufen. Ziel des im Comic-Stil ge-

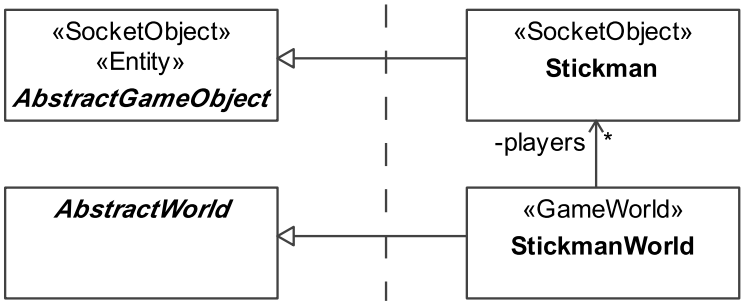



**Abbildung 7.2: Klassendiagramm des Spielmodells vom Demonstrator Stickman.**

haltenen Konzepts ist es, durch eine Landschaft zu laufen, Gegner zu finden und im Martial-Arts-Stil zu besiegen. Wie Viechers ist dieser Anwendungsfall Serverseitig in Java und Clientseitig in ActionScript 3 implementiert.

Das Spielmodell, siehe Abbildung 7.2, enthält zwei Entitätstypen, die *StickmanWorld* als Spielwelt und den Avatar als Spielobjekt, welche insgesammt 10 Funktionalitäten für den Client bereitstellt. Darunter zählen Funktionalitäten wie Bewegung, Schlag, Ausweichen, Springen, sowie weitere Manöver die der Avatar ausführen kann. Das mit 457 LLoC kompakt gehaltene Anwendungsbeispiel verwendet eine manuell bereitgestellte Anfrageverarbeitung zur Realisierung einer Authentifizierung und fünf zusätzliche Kommunikationsobjekte, welche zu Demonstrationszwecken darstellen, wie diese im Rahmen der Ausführungsumgebung in Kombination mit einem Spielmodell genutzt und bereitgestellt werden können.

### 7.1.3 Studentenprojekte

Zur Generierung weiterer Spielprojekte, wurde im Rahmen von Lehrveranstaltungen und praxisnaher Projektarbeit die Aufgabe zur Realisierung eines browserbasierten MMOGs an Studententeams gegeben.

Die initiale Aufgabe (siehe Aufgabe 1) der Studententeams von drei bis vier Personen bestand darin, ohne Kenntnis über Funktionsweise und Anwendung der Ausführungsumgebung, eine unabhängige Spielidee im Sinne eines klassischen Entwicklungsprozesses (siehe Kapitel 2.7, S. 18) zur entwerfen und in einer Ideenskizze festzuhalten. Die Skizze sollte anschließend in einer Anwendungsfall- (Aufgabe 2) und Anforderungsanalyse (Aufgabe 3) aus Sicht

einer Softwareentwicklung betrachtet werden und folgend in ein Designdokument (Aufgabe 4) zur Beschreibung der Entätiten eines Spielmodells überführt werden. Das Ergebnis war abschließend mithilfe der Ausführungsumgebung, sowie deren APIs zu implementieren (Aufgabe 5) und um einen Client zu ergänzen, welcher die realisierten Funktionen im Sinne eines Prototyps demonstriert. Für den gesamten Prozess stand ein Zeitraum von drei Monaten mit 180 Stunden Arbeitsaufwand zur Verfügung. Wichtig bei der Durchführung war, dass die Spielidee und Ergebnisse des Designprozesses ohne Kenntnis über die Ausführungsumgebung entstanden und nicht durch jene beeinflusst wurden.

**Aufgabe 1** *Erstelle eine Ideenskizze für ein MMO Browsergame. Beschreibe in der Skizze folgende Punkte:*

- ***Übersicht***
  *Die grundlegende Idee des Spiels, welchem Genre bedient es sich, wie soll die grafische Umsetzung aussehen, wer spielt gegen wen, was ist der MMO Gedanke.*

- ***Features***
  *Was sind die Kernelemente des Spiels - wodurch setzt es sich von anderen ab. Identifiziere einige „integral“ und „chrome“ Features.*

- ***Interface***
  *Eine kurze Beschreibung, wie der Spieler mit dem Spiel interagiert - z. B. wie wird das Spiel gesteuert, welche Bedienungselemente müssen vorhanden sein, wo könnten diese platziert werden.*

- ***Gameplay***
  *Beschreibe im Allgemeinen, wie das Spiel funktioniert und der Spielfluss entsteht. Wie beeinflussen die Feature das Gameplay. Achte auf Interaktionen mit anderen Spielern. (z. B. kann hier eine typische Situation im Spiel beschrieben werden)*

- ***Regeln***
  *Erstelle eine erste Liste an Regeln, die für die Spielwelt notwendig sind. Definiere Objekte, Abhängigkeiten und Beziehungen, sowie Regeln die das*

*Spiel beenden, Siegbedingungen Herausforderungen und Aktionsmöglichkeiten des Spielers.*

- ***Level Design***
  *Wie teilt sich das Spiel auf, gibt es überhaupt eine Aufteilung? Wo beginnt der Spieler im Spiel und wo soll er sich hinbewegen. Viele MMO Browsergames kommen mit einem Level aus, manche starten Spiele mit einem Tutorial Level, Level können ebenso Gebiete oder Zonen sein, die dynamisch betreten und verlassen werden können.*

**Aufgabe 2** *Modellieren Sie für die im Aufgabenblatt 1 entwickelte Spielidee Anwendungsfälle. Definieren Sie hierfür die Akteure und bestimmen Sie, welche Aktionen diese im Rahmen des Spiels ausführen können. Aktionen sollten aus Sicht des jeweiligen Akteurs formuliert werden. Aktionen könnten zum Beispiel sein:*

- *im Spiel anmelden*
- *einen neuen Avatar erstellen*
- *Anzeigen von Informationen*
- *Bewegen / Angreifen von Elementen*

*Erstelle hierfür eine UML 2.0 Use-Case Diagramm und beschreibe die Anwendungsfälle (in Kurzform). Zur Erstellung des Diagramms kann ein frei wählbarer Editor genutzt werden. (Falls keiner vorhanden ist, kann auf die Web-App http://draw.io zurückgegriffen werden).*

**Aufgabe 3** *Definieren Sie für die im Aufgabenblatt 1 entwickelte Spielidee und in Aufgabe 1 dieser Übung modellierten Anwendungsfälle (funktionale und nichtfunktionale, falls vorhanden) Anforderungen, die an das zu entwickelnde Spiel zu stellen sind. Anforderungen sollten primär aus dem Kontext des Spieles stammen, technische Aspekte zur Kommunikation, Synchronisiation und Timing sind im Rahmen des Team-Projektes nicht notwendig. Folgende Liste zeigt exemplarisch funktionale Anforderungen, die an ein Spiel gestellt werden könnten:*

- *dem Spieler stehen 42 Level zur Verfügung*

- *der Spieler soll nach 10 Minuten Inaktivität von der Spielwelt getrennt werden*
- *Gegenstände die nach einem Kampf vom besiegten fallen gelassen wurden, verschwinden nach 5 Minuten*
- *die Level des Spiels sind durch Portale miteinander verbunden*
- *der Übergang zwischen ist an Bedingung X geknüpft*
- *das Spiel besteht aus einer durchgängigen Karte*

*Anforderungen können aus den in der Ideenskizze formulierten Regeln des Spiels abgeleitet werden.*

**Aufgabe 4** *Modellieren Sie für die in Aufgabe 1 entwickelte Idee, in Aufgabe 2 modellierten Anwendungsfälle und in Aufgabe 3 abgeleiteten Anforderungen ein Klassendiagramm. Das Klassendiagramm sollte alle relevanten Entitätstypen des Spielmodells enthalten. Notwendige Methoden zur Modifikation des Spielmodells (z. B. move oder attack) und Attribute (z. B. position, health oder strenght) sollten in den Entitätstypen dargestellt werden. Beziehungen, deren Kardinalitäten und Bezeichnungen sollten ebenso visualisiert werden wie notwendige Vererbungen. Auf Getter- und Setter kann in der Darstellung für die Übersichtlichkeit verzichtet werden.*

**Aufgabe 5** *Implementieren Sie alle in Aufgabe 1 modellierten Entitätstypen des Spielmodells in Java Klassen. Benutzen Sie hierfür das Java-Projekt im Git-Repository ihres Team-Projekts. Berücksichtigen Sie die für die GameEngine notwendigen Abhängigkeiten zu AbstractGameWorld und AbstractGameObjects wie in der Vorlesung vorgestellt (siehe Level 5). Hinweise:*

- *Klassen müssen nur dann von AbstractGameObject erben, wenn diese eine aktive Rolle einnehmen (z. B. ein Spieler, ein Gegner oder ein Baum den man fällen kann), bzw. wenn die @SocketObjectRemoteMethod verwendet werden soll.*
- *Vektor-, Positions- oder Attributsklassen müssen zum Beispiel nicht von AbstractGameObject erben, da diese im Spielmodell nur als Datenobjekte verwendet werden.*

- *Alle Objekte die zum Client übertragen werden sollen, benötigen die @SocketObject Annotation*
- *Als Vorlage kann das Beispielprojekt aus der JavaScript Übung dienen (siehe MMOBG-Repository).*

Die Lehrveranstaltung wurde mit Inhalten zu Spielrelevanten Entwicklungsthemen wie Prozessen, Modellen und Architekturen begleitet. Dabei wurde Literatur aufgegriffen, welche im Kapitel 2 (S. 7) genutzt wurde, um Elemente eines Spiels zu definieren und das vorgehen zum Skizzieren eines Spielmodells erläutert. Folgend wurden Architekturen zur Erstellung von MMOGs betrachtet, insbesondere TCP-, REST- und WebSocket-basierte Server-Client Ansätze. Abschließend und mit Beginn der Aufgabe 5 wurde eine Einführung in die Anwendung der Ausführungsumgebung gegeben.

Die Projekte realisieren Anwendungen, welche zur Erprobung der Ausführungsumgebung in beliebigen Spielideen dient. Durch die Trennung von Designprozess und Implementierung, sowie dem Vermeiden von Kenntnissen über die Ausführungsumgebung, wurde sichergestellt, dass die Spielideen unabhängig entstehen und nicht auf mögliche Restriktionen der Ausführungsumgebung zurückzuführen sind. Durch die anschließende Realisierung der Spielidee in der Ausführungsumgebung, lässt sich zeigen, dass die Spielideen ohne zusätzliche Anpassungen an Konzept und Design in der Ausführungsumgebung realisierbar sind und eine Anwendung des generischen Ansatzes der Umgebung möglich ist.

### Terra Life Evolution

Terra Life Evolution ist ein RPG-Spiel in dem der Spieler aus einer Third-Person Perspektive eine Insellandschaft erkundet und dabei Erfahrungen sammelt um seinen Avatar zu entwickeln. Die Welt differenziert sich hierbei in Inseln, welche wiederum in Zonen unterteilt sind, welche für einen vordefinierten Bereich des Entwicklungsstandes des Avatars geeignet sind. Je weiter der Avatar entwickelt ist, desto anspruchsvollere Zonen können von ihm erkundet werden. Ein Einblick in die Ideenskizze des Projektes liefert Anhang K (S. 257).

Das in Java implementierte Spielmodell mit 771 LLoC ist in Abbildung 7.3 dargestellt. Es enthält eine Spielwelt als zentrales Element über die sich die

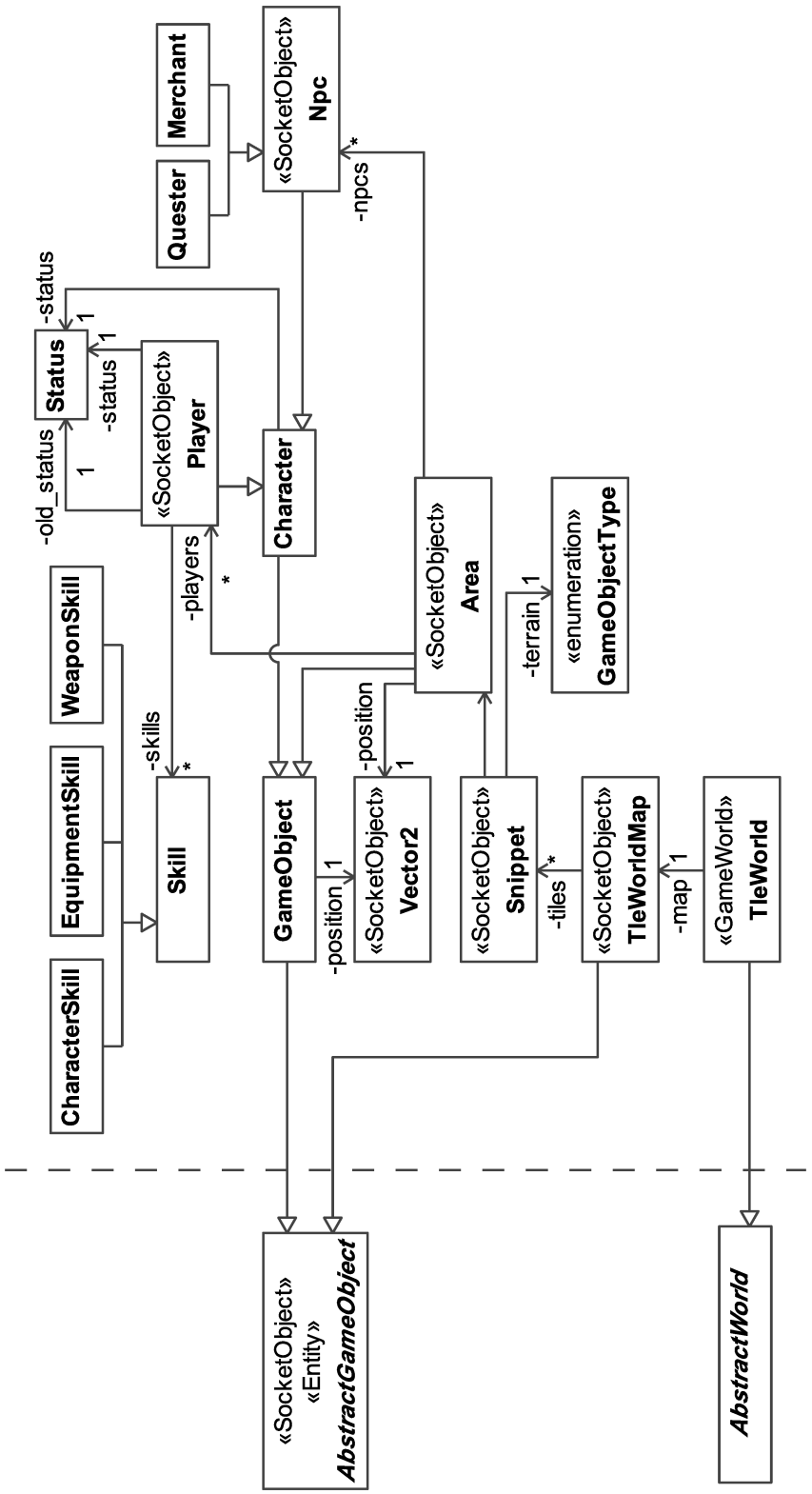



**Abbildung 7.3: Klassendiagramm des Spielmodells von Terra Life Evolution.**

Entitäten zur Abbildung der Karten (*TleWorldMap*, *Snippet* und *Area*) angliedern. Darüber hinaus verwendet das Model ein eigenes Spielobjekt für interagierende Entitäten mit Funktionalitäten, welche über klassische Datenkapseln hinaus gehen, und spezialisieren diese zu *NPC*, *Player* und Kartenelementen. Abweichend von der Skizze enthält die Realisierung kein Inventar oder Gegner, welche eine spezielle Form des *NPC*-Entitätstyps wären. Diese fehlenden Realisierungen konnten aus zeitlichen Gründen bis zum Ende des Projektes nicht implementiert werden.

Das Modell definiert in 8 Spielobjekten insgesamt 15 Funktionalitäten, welche für den in JavaScript implementierten Client bereitgestellt werden. Diese umfassen z. B. Funktionalitäten zum Bezug der aktuellen Insel, Bewegen des Avatars und Entwickeln des Avatars. Der Austausch von Informationen zwischen Client und Server konnte in diesem Projekt ohne Implementierungen von Anfrageverarbeitungen oder Kommunikationsobjekten realisiert werden. Sämtliche bereitgestellten Funktionalitäten wurden mittels Annotation der jeweiligen Modellmethode realisiert.

**Hack'N'Slay MMO**

Hack'N'Slay MMO beschreibt ein rundenbasiertes Areana-Konzept des Hack'N'Slay Genres, bei denen mehrere Wellen von computergesteuerten Gegner von einem Spielerteam zu bewältigen sind um ein zentrales Objekt, den Lebensbaum, zu beschützen. Die in separierten Kämpfe instanziieren jeweils den Lebensbaum, sowie Friedhöfe, von denen die Gegner heranstürmen, und bieten Möglichkeiten kurzfristige Boni einzusammeln, die den Spieler bei der Aufgabe unterstützen sollen. Ein Einblick in die Ideenskizze des Projektes liefert Anhang L (S. 263).

Das Spielmodell in Abbildung 7.4 zeigt eine Spielwelt welche verschiedene Spezialisierungen von *Entity* enthalten. Diese positionierbaren Elemente differenzieren sich in bewegliche und unbewegliche, sowie jene die vom Spieler und jene die von der Simulation gesteuert werden. Zusätzlich definiert das Modell die Aktionen zur Realisierung der verschiedenen Status, insbesondere den Wellen, welche hier mittels *Round1* bis *Round10* abgebildet sind.

Das Spielprojekt ist auf der Serverseite in Java und auf der Clientseite in JavaScript implementiert. Die 17 Spielobjekte des Modells mit einem Um-

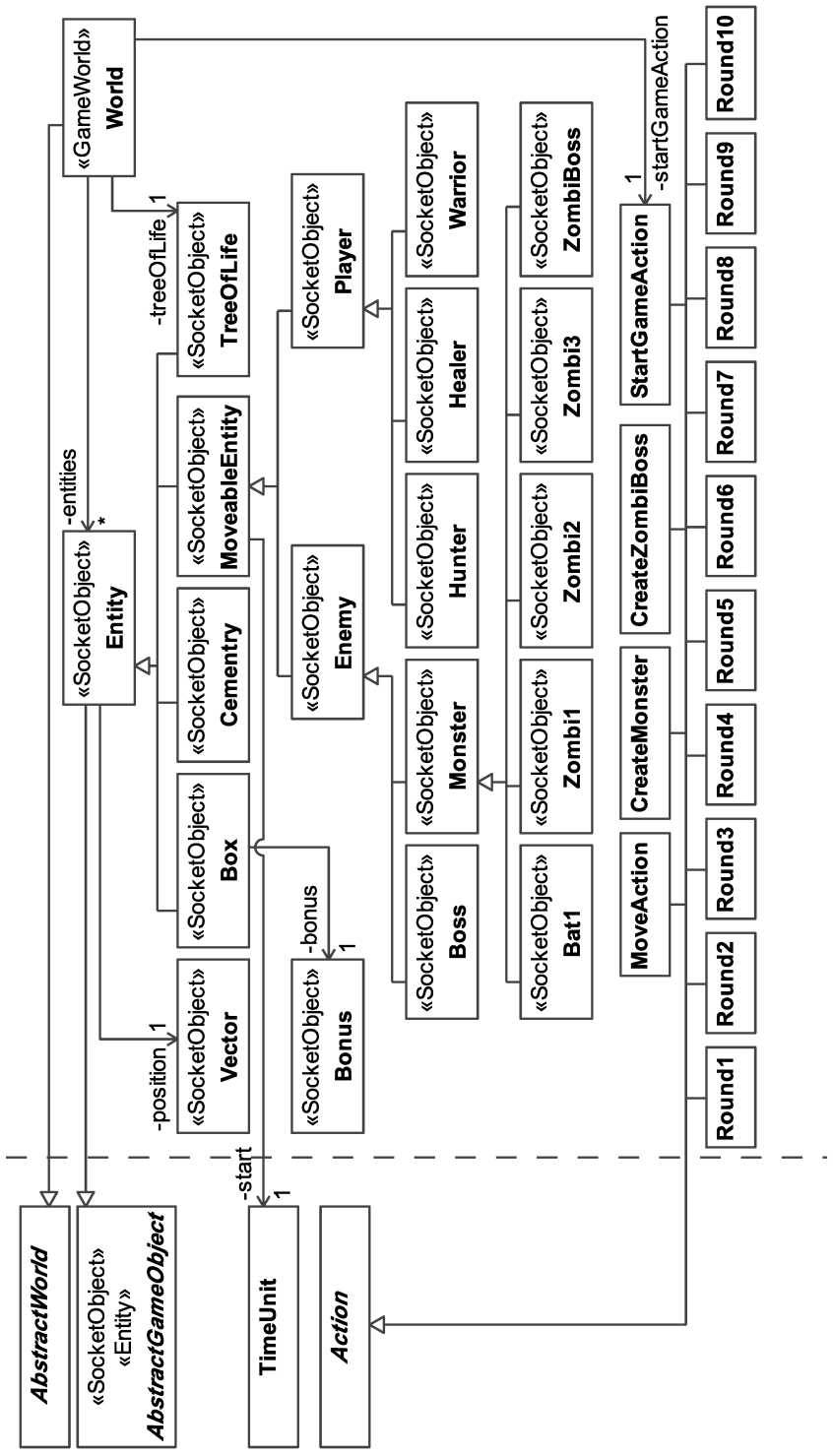



**Abbildung 7.4: Klassendiagramm des Spielmodells von Hack'N'Slay MMO.**

fang von 767 LLoC stellen insgesamt 13 verschiedene Funktionalitäten für den Client bereit. Diese umfassen z. B. Funktionalitäten zum Angreifen, Heilen, Bewegen, Beitreten eines Kampfes, sowie Bezug der aktuellen Kampf- und Rundeninformationen. Jede Runde, die von den Teams zu überstehen ist, sendet in einer zeitbasierten Taktung Gegner, welche mithilfe von Aktionen im Spielmodell realisiert wurden. Mit der Zielstellung zur Spielmodell-zentrischen Realisierung der Spielidee konnte das Projekt ohne Implementierungen von Anfrageverarbeitungen oder Kommunikationsobjekten realisiert werden.

**Kokosnussbikini**

Das Spielprojekt Kokosnussbikini beschreibt eine Survival-MMO-Idee in Anlehnung an Terraria [65] und Minecraft [53], bei der der Spieler in einer isometrischen Landschaft Rohstoffe abbauen, Tiere jagen, Monster bekämpfen und Objekte in der Welt konstruieren kann. Ein Einblick in die Ideenskizze des Projektes liefert Anhang M (S. 267).

Abbildung 7.5 zeigt die Entitätstypen dieses Spielmodells. Auf einer Welt definieren sich verschiedene Teilbereiche zur Verwaltung von Inventaren *Inventory*, den zugehörigen Gegenständen *Item*, Objekte auf der Karte *ResourceObject* und bewegliche Objekte wie Spieler *Player* und Gegner *Enemy*.

Das Spielprojekt ist auf der Serverseite in Java und auf der Clientseite in JavaScript implementiert und mithilfe der aktuellen Version der Ausführungsumgebung realisiert. Mit 13 Spielobjekten stellt das Modell mit einem Umfang von 845 LLoC 18 verschiedene Funktionalitäten für den Client bereit um die Möglichkeiten des Abbauens, Bewegens, Verwalten des Inventars, Bezug der Karte, Erzeugen von Objekten und dem Angreifen zu realisieren. Mithilfe von Aktionen realisiert das Konzept Prozesse die nach einer bestimmten Zeit finalisiert werden, z. B. das Abbauen wird durch eine Funktionalität gestartet und einige Sekunden später das Ergebnis an den Client übermittelt. Mit der Zielstellung zur Spielmodell-zentrischen Realisierung konnte die Spielidee ohne Implementierungen von Anfrageverarbeitungen oder Kommunikationsobjekten realisiert werden.

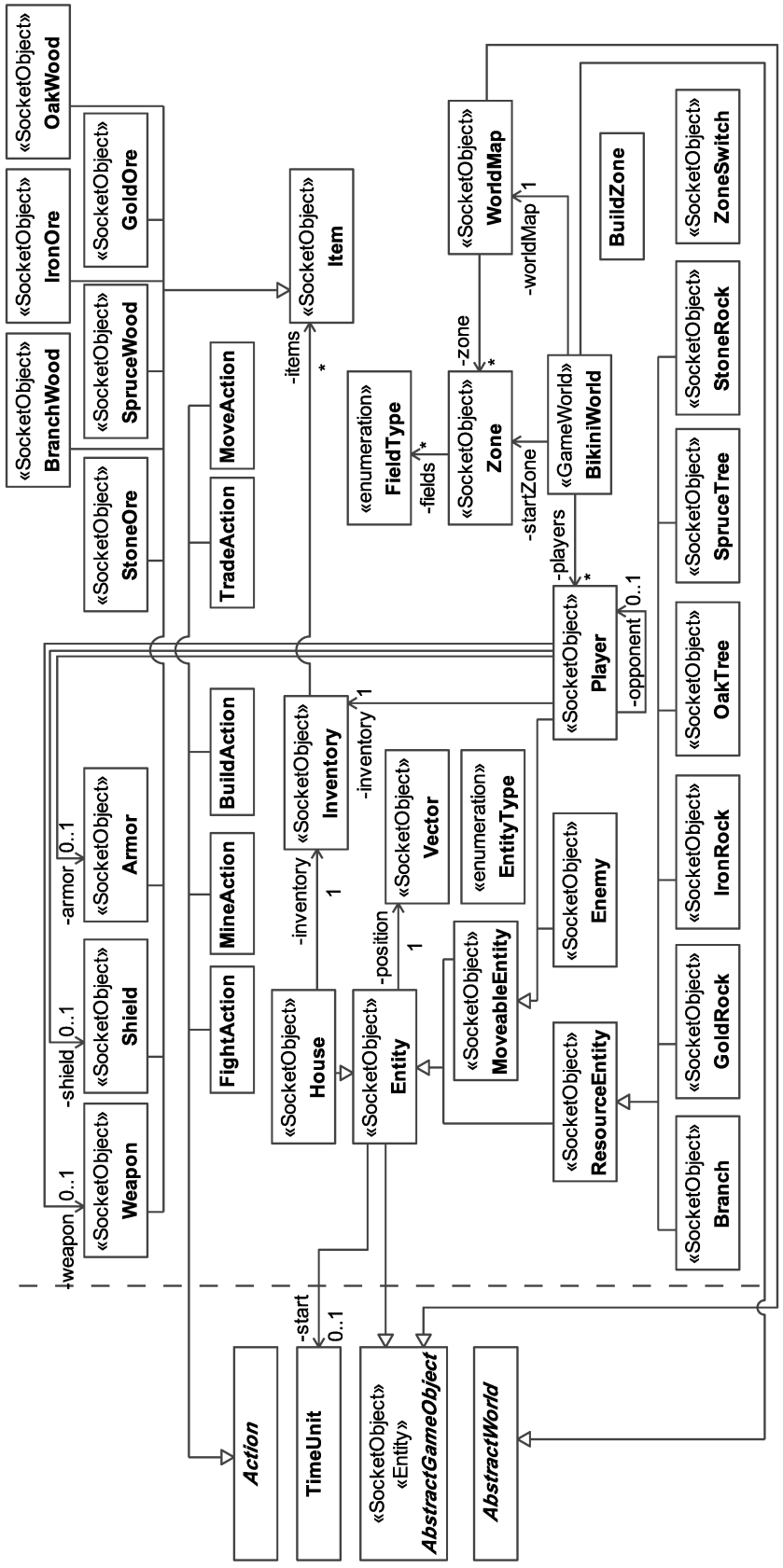



**Abbildung 7.5: Klassendiagramm des Spielmodells von Kokosunuss-bikini.**

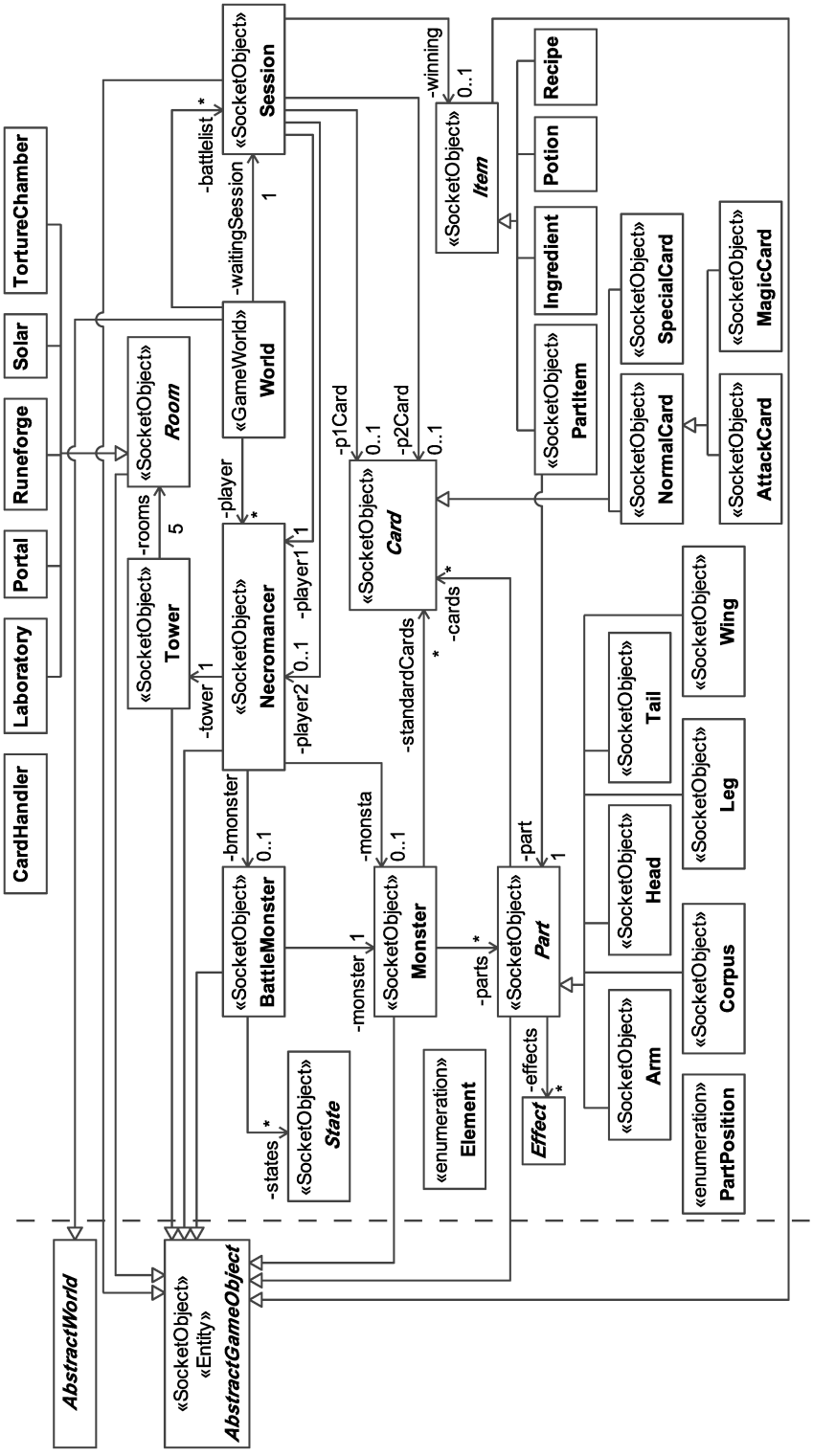



**Abbildung 7.6: Klassendiagramm des Spielmodells von MMO-Meischta.**

**MMO-Meischta**

Das Spielprojekt MMO-Meischta beschreibt ein Towerdefense- und Strategiespielidee, bei denen der Spieler als Necromancer seine sogenannten Igors beschäftigt um einen Turm auszubauen und sein eigenes Monster zu züchten, welches auf Beutezüge geschickt wird oder gegen andere Monster antreten kann. Der Kampf wurde im Stil eines Kartenspiels realisiert, wobei die zur verfügung stehenden Karten aus den am Monster anhängenden Körperteile resultieren und in einem rundenbasierten Ablauf im Kampf gegeneinander ausgespielt werden. Ein Einblick in die Ideenskizze des Projektes liefert Anhang N (S. 273).

Abbildung 7.6 zeigt das realisierte Spielmodell. Das Konzept dreht sich erneut um eine Spielwelt, in der die Elemente *Necromancer*, Turm *Tower* und dessen Räume *Room*, sowie das *Monster*, dessen Körperteile *Part* und Gegenstände *Item* organisiert werden. Die Entitätstypen erben direkt von *AbstractGameObject* und realisieren das Konzept aus der Ideenskizze.

Das Spielprojekt ist auf der Serverseite in Java und auf der Clientseite in JavaScript implementiert und mithilfe der aktuellen Version der Ausführungsumgebung realisiert. Das Modell aus Abbildung 7.6 hat einem Umfang von 904 LLoC und umfasst 18 Spielobjekte in einer Spielwelt. Dieses Spielmodell exponiert ingesammt 19 verschiedene Funktionalitäten um den Ausbau des Turmes, das Kämpfen und den Bezug von Informationen zu realisieren. Der Austausch von Informationen zwischen Client und Server konnte, wie in den zuvor vorgestellten Spielprojekten, ohne Implementierungen von Anfrageverarbeitungen oder Kommunikationsobjekten realisiert werden.

## 7.2 Analyse der Spielkonzeptrealisierungen

Die in Java, JavaScript und ActionScript realisierten Spielprojekte unterschieden sich in der Komplexität des Modells und dem Zeitpunkt der Realisierung bezüglich dem Entwicklungsstand der Ausführungsumgebung. Während Viechers parallel zur Ausführungsumgebung entstand und die Möglichkeit bot Eigenschaften der Ausführungsumgebung in einem realen Anwendungsfall zu Erproben, sowie Anforderungen durch das Spielprojekt in die Entwicklung der Ausführungsumgebung einfließen zu lassen, stellt das Projekt Stickman eine

| | *Spielobjekte* | *Kommunikationsobjekte* | *Anfrageverarbeitung* | *Entfernte Methodenaufrufe* | *Ereignisse* | *Aktionen* | *LLoC* | *PLoC* |
|---|---|---|---|---|---|---|---|---|
| Viechers | 12 | 306 | 119 | 0 | 80 | 32 | 30.705 | 45.088 |
| Stickman | 1 | 5 | 1 | 10 | 0 | 0 | 457 | 667 |
| BasicGame | 1 | 0 | 0 | 5 | 0 | 0 | 59 | 92 |
| Terra-Life-Evolution | 8 | 0 | 0 | 15 | 0 | 0 | 771 | 1.079 |
| Hack'N'Slay | 17 | 0 | 0 | 13 | 0 | 14 | 767 | 1.110 |
| Kokosnussbikini | 13 | 0 | 0 | 18 | 0 | 5 | 845 | 1.178 |
| MMO-Meischta | 18 | 0 | 0 | 19 | 0 | 0 | 904 | 1.254 |

**Tabelle 7.1: Vergleich der realisierten Spielprojekte bezüglich verwendeter Spielobjekte, Anfragen, Antworten, Ereignisse und Aktionen, sowie ein Einblick in den Umfang der Spielmodellimplementierung.**

ersten Versuch der Anwendung der finalen Ausgestaltung der Ausführungsumgebung dar. Die im Anschluss durchgeführten Stundentenprojekte wenden die Ausführungsumgebung abschließend an um die realisierten Konzepte in variablen Spielideen in unterschiedlichen Genre zu erproben. Für die Erprobung stand im Vordergrund, ob die Ausführungsumgebung auf beliebige Spielideen anwendbar ist und ob die Realisierung jener Ideen ohne Anwendung von spezifischen Kommunikationsprotokollen möglich ist. Die durch die Projekte entstandenen Spielmodelle wurden durch die Probanden implementiert und automatisiert in die jeweiligen Clientsprachen übersetzt.

Quellcode 7.1 führt die im vorangegangenen beschriebenen Spielprojekte zusammen und stellt deren Eigenschaften gegenüber. Während das Projekt Viechers alle Funktionalitäten mittels manuell implementierten Kommunikationsobjekten und Anfrageverarbeitungen realisiert und das Projekt Stickman die Möglichkeiten der Ausführungsumgebung ausschöpft, zeigen die Stundenprojekte das die Kommunikationsobjekte und Anfrageverarbeitungen vermeid-

bar sind.

### 7.2.1 Anwendung eines abstrakten Spielmodells

Die sechs Anwendungsfälle Viechers, Stickman, MMO-Meischta, Terra-Life-Evolution, Hack'N'Slay und Kokosnussbikini wurden im Sinn eines Reverse Engineerings nachträglich in ein Klassendiagramm überführt um den implementierten Stand mit dem intendierten Modell zu vergleichen. Zusätzlich kann diesen Diagrammen die Generalisierbarkeit der Spielmodelle auf ein abstrakten Spielmodells, dem EGM, entnommen werden. Die untersuchten Anwendungsfälle realiseren Konzepte in unterschiedlichen Spielgenre: Viechers klassifiziert sich als Aufbaustrategie, Stickman als Jump'N'Run, MMO-Meischta als Tower Defense, Terra-Life-Evolution als RPG, Hack'N'Slay als Hack'N'Slay und Kokosnussbikini als Survival. Die Anwendungsfälle lassen sich auf die Definition eines Spiels zurückführen: mehrere Symboliken, welche durch Spielobjekte beschrieben sind; eine „pretended reality", welche auf eine Spielwelt zurückführbar ist und in welcher die Spielobjekte agieren; sowie Aktionen und Ereignisse. Dabei differenzieren sich diese Entitätstypen in jene welche im Spielkonzept als Spielwelt, Spielobjekt, Spielaktionen und Spielereignisse agieren und jene welche als zur Beschreibung von Daten für jene dienen. In keinem Anwendungsfall ist die Notwendigkeit entstanden, über die abstrakten Entitätstypen von Spielwelt, Spielobjekten, Spielaktion und Spielereignis hinaus weitere Mechanismen oder Konzepte zu ergänzen.

### 7.2.2 Anfrageverarbeitung

Die vier Studentenprojekte haben einen vergleichbaren Aufwand mit durchschnittlich 822 LLoC und stellen durchschnittlich 16 Funktionalitäten zur Realisierung des Clients bereit. Da weder Kommunikationsobjekte noch Anfrageverarbeitung verwendet wurde, unterscheidet sich die Menge der Anfrageverarbeitungen sich signifikant von der Menge der entfernten Methodenaufrufen in einem realisierten Spielkonzept. Die Studentenprojekte konnten ausschließlich mit entfernten Methodenaufrufen realisiert werden. Davon Abweichend wurde im Demonstrator eine Anfrageverarbeitungen im Vergleich zu den zehn entfernten Aufrufen verwendet. Diese Anfrageverarbeitung demonstriert dessen

Einsatz und verweist auf eine Funktionalität im Spielmodell. Wie im Fall der Spielprojekte mit Studenten, kann von einem signifikanten Unterschied gesprochen werden, zumal die einzige Anfrageverarbeitung durch einen entfernten Methodenaufruf realisierbar ist.

Die 119 Anfrageverarbeitungen von Viechers eignen sich zur Betrachtung von Variationen jener Verarbeitungen in einem Spielprojekt mit mit hoher Komplexität. Hierfür wurde jede Anfrage hinsichtlich den folgenden Merkmalen Betrachtet:

- **Anfrageobjekt**
  Parameter die für die Anfrage verwendet werden, bzw. mit das Kommunikationsobjekt was die Parameteransammlung beschreibt.

- **Antwortobjekte**
  Antworten die durch die Anfrage erzeugt werden. Hierfür wurden die in der Behandlungsmethode möglichen Antworten betrachtet.

- **Modellaufruf**
  Ist die Verarbeitung auf einen Modellaufruf zurückführbar oder werden mehrere Verarbeitungsschritte, z. B. unter einbeziehung Modell-fremder Dienste, zur Erzeugung des Ergebnisses durchgeführt? Ignoriert wurden Transformationen, welche zur Aufbereitung der Antwort dienen, bzw. Instruktionen die unterschiedliche Antworten je nach Rückgabe des Modellaufrufs erzeugen.

- **Möglichkeit des Modellaufruf**
  Wenn die Anfrageverarbeitung eine komplexe Ansammlung von Verarbeitungsschritten ist, also kein einzelner Modellaufruf, wäre es möglich die Anfrage als Modellaufruf zu realisieren? Dies tritt insbesondere ein, wenn für die Anfrageverarbeitung keine externen Dienste genutzt werden und die Instruktionen der Anfrageverarbeitung eine Ansammlung von mehreren Modellaufrufen und daraus resultierenden Abläufe beschreibt.

Abbildung 7.7 zeigt eine Zusammenfassung dieser Identifikation von Merkmalen. Von den insgesammt 119 Anfrageverarbeitungen sind 91 % auf Modellaufrufe zurückzuführen, bzw. als jene realisierbar. Von den 108 Anfrageverarbeitungen mit Modellaufrufen sind 51 % auf Methoden-Signatur-Ähnliche

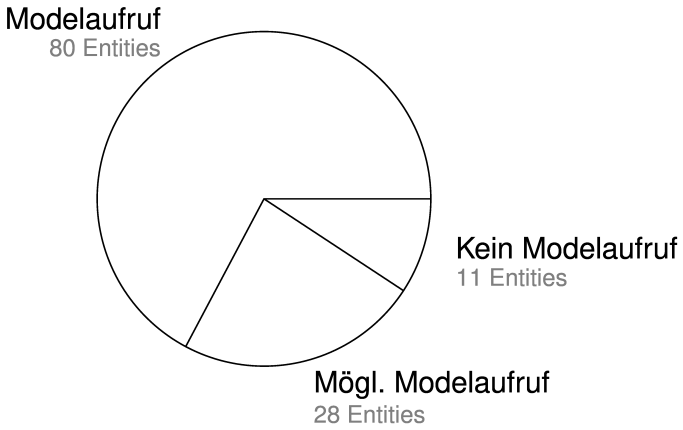



**Abbildung 7.7: Verhältnis von Anfrageverbeitungen in Viechers bezüglich Modellaufruf, möglicher Modellaufrufe und den verbleibenden.**

Strukturen zurückzuführen: Eine Identifikation der Verarbeitung, eine Menge an Parametern, ein Rückgabewert. Modellaufrufen die diese Eigenschaft nicht aufweisen haben mehrere Rückgabewerte, welche je nach Ergebnis des Modellaufrufs variieren, oder verteilen an andere Spieler Informationen die durch den Aufruf entstehen. Die verbleibenden 11 Anfrageverarbeitungen sind Instruktionen mit Kombinationen aus optionalen Modell-Aufrufen und Dienst-Aufrufen, z. B. der Aufruf von Chat-, Nachrichten-, Bezahl- und Authentifizierungsdiensten, die nicht in einem exklusiven Zusammenhang mit dem Spielmodell stehen. Im Anwendungsfall Viechers kann somit auf einen signifikanten Anteil an Anfrageverarbeitung zur Realisierung von Funktionalität die auf das Spielmodell zurückzuführen sind, geschlossen werden.

## 7.3 Analyse des Aufwands

Die Analyse des Aufwands dient zur Betrachtung des vermiedenen Overheads im Rahmen der realisierten Spielprojekte und wird in drei Fällen bewertet. Der erste Fall greift das Beispiel aus Kapitel 2 (S. 7) auf, ergänzt die Auswertung um die Implementierung aus Kapitel 6.6 (S. 175) und bewertet die Verbesserung bezüglich des vermiedenen Overheads zu den bereits betrachteten Architekturen. Der zweite Fall verwendet eines der Studentenprojekte und Analysiert dessen Ergebnis mit dem selben Ansatz wie es für die Bewertung

des Beispieles realisiert wurde. Der dritte Fall verwendet das Spielprojekt Viechers und führt auf der komplexen Implementierung eine grobe Betrachtung der Aufwandsverteilung durch.

Im Rahmen der Bewertung wurden acht Charakteristiken für die Implementierung eines Spielsmodells und der zugehörigen Infrastruktur zur Bereitstellung von Funktionalitäten definiert, sowie drei zusätzliche Charakteristiken welche den Betrieb eines Spiels unterstützen. Für die Bewertung wurde der Aufwand der für die Realisierung der acht Charakteristiken notwendig ist mittels Bewertung des Implementierungsaufwandes auf Basis von LLoC dessen Ermittlung im COCOMO II Kostenmodell beschrieben wird.

### 7.3.1 Beispiel-Implementierung

Wird das minimale Spielmodell aus Abbildung 2.7 mithilfe der entwickelten Ausführungsumgebung realisiert, sind neben der Implementierung des eigentlichen Spielmodells Konfigurationen an den bereitzustellenden Funktionalitäten notwendig. Diese können mittels Annotationen im Quellcode oder mittels Konfigurationsdatei definiert werden. Darüber hinaus ist es möglich, wie das Projekt Stickman demonstriert, zusätzliche Instruktionen im Sinne eines Controllers über sogenannte *RequestHandler* (siehe Kapitel 6.2.3, S. 153) zu definieren und individuelle Kommunikationsobjekte (siehe Kapitel 6.2.2, S. 138) zum Informationsaustausch zu verwenden. Im Fall des Beispiels sind diese zusätzlichen Elemente nicht notwendig. Das Spielmodell wird mittels Markierung öffentlicher Funktionalität und übertragbarer Entitätstypen für die Ausführungsumgebung aufbereitet.

Wird das Ergebnis mit dem selben Werkzeug zur Quantifizierung des Aufwandes aus Kapitel 3.6 (S. 78) analysiert, ergibt sich für die Bewertung der Strategie „Ausführungsumgebung“ das in Abbildung 7.8 visualsierte Ergebnis, welches im Vergleich zu den alternativen Strategien dargestellt wird. Begründet in dem Wegfall der Characteristiken API, Controller, Kommunikationsobjekte, Message Handler, Socketserver und Sitzungsverwaltung, ermöglicht die Anwendung der Ausführungsumgebung die Bereitstellung des Beispiels mit einem minimalen Mehraufwand.

Schwierigkeiten bereitet bei dieser Messung die Ermittlung der LLoC des Kostenmodell COCOMO II, welches im Fall der Verwendung von Annotatio-

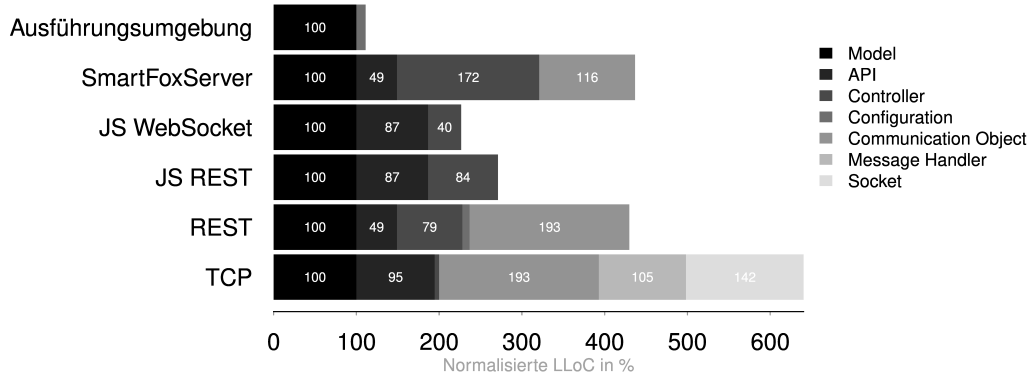



**Abbildung 7.8: Darstellung des Aufwands für die verschiedenen Strategien zur Realisierung des minimalen Beispiels.**

nen zur Konfiguration nur den Import der Annotation bewertet, nicht die Annotation selbst. Quellcode 7.2 stellt diese Werte, inkl. Strategie, Charakteristik und Funktionalität, gegenüber. Während im Fall der Quantifizierung von LLoC eine Funktionalität mit einem Aufwand (z. B. *BasicWorld.getAvatars*) von eins und alle folgenden (z. B. *BasicWorld.getAvatar* und *BasicWorld.newAvatar*) mit null bewertet werden, wird bei der Quantifizierung mittels LoC der Aufwand für die Konfiguration wie intendiert mit initial zwei und folgend mit eins angegeben. Für die Betrachtung ist diese unklare Messung nicht kritisch. Wird in dem verwendeten Beispiel auf die Quantifizierung mittels LoC zurückgegriffen, ergibt sich ein Ergebnis wie in Abbildung 7.9 dargestellt, welches eine vergleichbares Ergebnis über die kumulative Entwicklung des Aufwands je Funktionalität präsentiert. Das Endergebnis bewegt sich bei ca. 12 % Mehraufwand, wenn man die zusätzlichen Konfigurationselemente mit dem Spielmodell vergleicht. Im Fall des Beispiels ist damit der Overhead für API, Controller, Kommunikationsobjekte, Message Handler, Socketserver und Sitzungsverwaltung vermeidbar.

### 7.3.2 MMO-Meischta

In Analogie zum minimalen Beispiel wird die Analyse zur Bewertung des Entwicklungsaufwandes auf das Spielprojekt MMO-Meischta als unabhängiges Spielprojekt angewendet. Das Projekt hat im Vergleich die meisten bereitgestellten Funktionalitäten und erlaubt damit einen Einblick in die Auf-

| *Entität* | *Strat.* | *Funktionalität* | *Char.* | *LoC* | *LLoC* |
|---|---|---|---|---|---|
| Avatar | gameengine | init | model | 16 | 12 |
| Avatar | gameengine | init | model | 6 | 4 |
| Avatar | gameengine | UpdateAvatar | configuration | 2 | 1 |
| Avatar | gameengine | MoveAvatar | configuration | 1 | 0 |
| Avatar | gameengine | init | configuration | 2 | 1 |
| BasicWorld | gameengine | init | model | 16 | 9 |
| BasicWorld | gameengine | init | configuration | 4 | 2 |
| BasicWorld | gameengine | init | model | 12 | 9 |
| BasicWorld | gameengine | GetAvatars | configuration | 2 | 1 |
| BasicWorld | gameengine | NewAvatar | configuration | 1 | 0 |
| BasicWorld | gameengine | GetAvatar | configuration | 1 | 0 |
| Position | gameengine | init | model | 27 | 19 |
| Position | gameengine | init | configuration | 2 | 1 |

**Tabelle 7.2: Messwerte der Beispielimplementierung unter Verwendung der Ausführungsumgebung.**

wandsentwicklung bei deren Realisierung. Diese Betrachtung wird abweichend zum Beispiel in den Java-basierten Strategien für die Serverimplementierung durchgeführt, nicht in den JavaScript-basierten Strategien.

Zur Erstellung der alternativen Strategien wurde die aus dem Studentenprojekt resultierende Implementierung aufgegriffen, Abhängigkeiten zur API der Ausführungsumgebung beseitigt und in Anlehnung an das minimale Beispiel die Strategien Java TCP, Java REST und SmartFoxServer unter Verwendung des Spielmodells implementiert. Dabei ist die Client-Implementierung von untergeordnetem Interesse und wird im folgenden, wie im Fall des minimalen Beispiels, nicht betrachtet.

Für die Bewertung des Aufwandes wird das selbe Werkzeug zur Ermittlung von LoC und LLoC verwendet, sowie der Definitionsbereich für die Bewertung übernommen und an die Funktionalitäten und Strategien dieser Betrachtung angepasst. Stategien $S$ umfassen: Java TCP (1), Java REST (4) und SmartFoxServer (5), sowie eine spezielle Strategien für mehrfach verwendete Java-Implementierungen „All“ und die Ausführungsumgebung (8); für die Charakteristiken $C$ werden die bereits herausgestellten Merkmale unverändert verwendet: Model (1), API (2), Controller (3), Kommunikationsobjekte (4), Message Handler (5), Konfiguration (7) und Socket (8); und für die Funktionalitäten $F$ werden die bereitzustellenden Methodensignaturen des Spielmodells

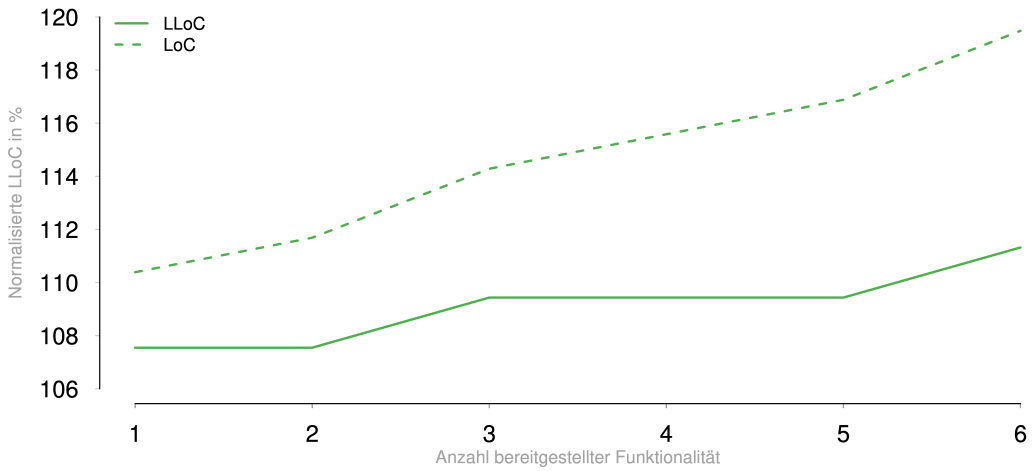



**Abbildung 7.9: Vergleich der Bewertung des Aufwands für die Realisierung Ausführungumgebung mittels LLoC und LoC.**

verwendet. Im Fall des Projektes MMO-Meischta sind dies 19 verschiedene Funktionalitäten. Strategie, Charakteristik und Funktionalität werden folgend numerisch angegeben und durch deren Mengen definiert:

$$s \in S, c \in C, f \in F \tag{7.1}$$

$$S := \{1, 4, 5, 6, 8\} \tag{7.2}$$

$$C := \{1, 2, 3, 4, 5, 6, 7, 8\} \tag{7.3}$$

$$F := \{1, 2, ..., 19\} \tag{7.4}$$

Als Ergebnis der Messung für das Spielprojekt MMO-Meischta, welche im Kapitel Q (S. 283) zu finden ist, kann jede Zeile $z \in Z$ der Implementierung für die Ermittlung des Aufwands $b_1(s, c, f)$ und Klassifizierung bzgl. der Strategie, Charakteristik und Funktionalität bewertet werden. Hierfür sollen die Funktionen $Strategy(x) \in S$, $Characteristic(x) \in C$ und $Functionalty(x) \in F$, zur Klassifizierung, und die Bewertungsfunktion $Countable(x) \in [0, 1]$, zur Entscheidung ob eine Zeile zählbar im Sinne des COCOMO II LLoC Modells ist, dienen. Die Anzahl der daraus resultierenden Zeilen sei wie folgt definiert:

$$
\begin{aligned}
b_1(s,c,f) = |\{x | x \in Z \wedge \\
&\text{Strategy}(x) = s \wedge \\
&\text{Characteristic}(x) = c \wedge \\
&\text{Functionalty}(x) = f \wedge \\
&\text{Countable}(x) = 1\}|
\end{aligned} \tag{7.5}
$$

Für die Betrachtung des Endergebnisses wird der gemeinsam genutzte Aufwand den jeweiligen Strategien je nach Verwendung zugerechnet:

$$
b_2(s,c,f) = \begin{cases}
b_1(s,c,f) + a_1(6,c,f) & \text{wenn } s = 1 \\
b_1(s,c,f) + a_1(6,c,f) & \text{wenn } s = 4 \\
b_1(s,c,f) + a_1(6,c,f) & \text{wenn } s = 5 \text{und } c < 4 \\
b_1(s,c,f) & \text{sonst}
\end{cases} \tag{7.6}
$$

Sowie, für deren Vergleich, anschließend der mit $b_2(s,c,f)$ ermittelbare Aufwand gegen das Spielmodell normalisiert:

$$
b_3(s,c,f) = \frac{b_2(s,c,f)}{b_2(s,1,f)} \tag{7.7}
$$

Abbildung 7.10 zeigt, die Ergebnisse verwendend, die Strategie Java TCP, Java REST, SmartFoxServer und Ausführungsumgebung mit normalisierten Gesamtaufwänden im finalen Zustand der Implementierung im Vergleich, welche sich aus $b_5(s)$ ergeben:

$$
b_4(s,c) = \sum_{f \in F} b_3(s,c,f) \tag{7.8}
$$

$$
b_5(s) = \sum_{c \in C} b_4(s,c) \tag{7.9}
$$

Während die Ausführungsumgebung durch den realisierten Abstraktions-

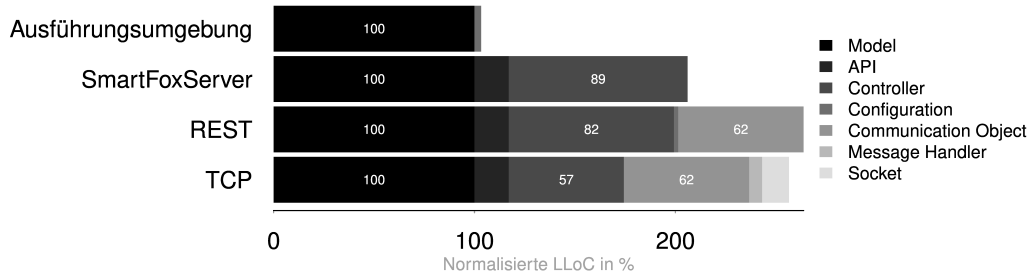



**Abbildung 7.10: Darstellung der normierten Aufwände für die Realisierungen des Spielmodells MMO-Meischta in unterschiedlichen Strategien.**

grad die Vermeidung des Overheads demonstriert, zeigt die Auswertung zusätzlich, dass in diesem Anwendungsfall ein erneut signifikanter Overhead ($c \in [2, 8]$) bei der Verwendung der alternativen Strategien entsteht. Der Overhead für die alternativen Strategien ist im Durchschnitt bei einem Verhältnis von 1:1,422. Im Idealfall liegt dieser Aufwand im Fall der SmartFoxServer Strategie bei 1:1,061 im analysierten Anwendungsfall.

Für die durchgeführte Betrachtung ist vom weiteren Interesse die kumulative Entwicklung des Overheads je bereitgestellte Funktionalität. Abbildung 7.11 zeigt, wie sich das im Fall des Spielprojektes MMO-Meischta für die 19 Funktionalitäten darstellt. Jede Funktionalität bedeutet für den Anwendungsfall einen zusätzlichen, stetig steigenden Overhead welcher im Fall der Ausführungsumgebung nicht vermeidbar, jedoch auf ein Verhältnis zwischen Spielmodell ($c = [2, 8]$) und Overhead ($c \in [2, 8]$) von 1:0,033 reduziert wurde. Die Abbildung 7.11 zeigt in Anlehnung an Kapitel 3.6 (S. 78) erneut ein Anstieg des Overheads in der Implementierung, womit Vorschrift 7.11 gilt:

$$b_6(s, f) = \sum_{c \in C} b_3(s, c, f) + b_6(s, f - 1) \tag{7.10}$$

$$b_6(s, f) > b_6(s, f - 1) \quad \text{wenn } s > 1 \tag{7.11}$$

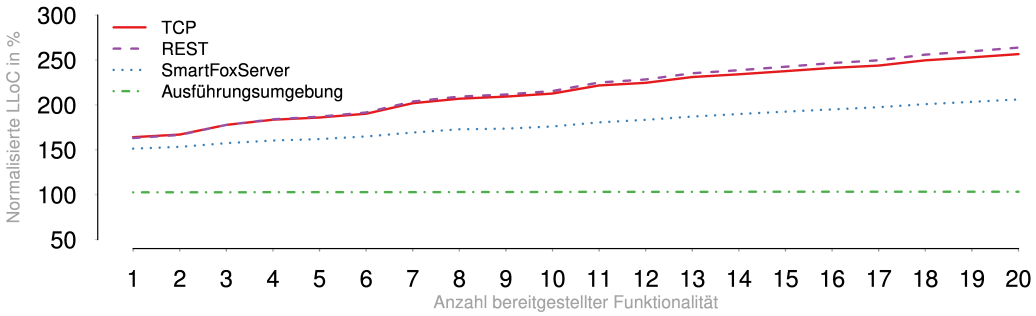



**Abbildung 7.11: Darstellung der Aufwandsentwicklung für das Spielprojekt MMO-Meischta je bereitgestellte Funktionalität.**

### 7.3.3 Viechers

Die dritte Aufwandsbetrachtung verwendet das Spielkonzept Viechers. Entgegen des Verfahrens im Fall des Beispiels und des Spielkonzepts MMO-Meischta ist es nicht möglich eine detailierte Code-Analyse durchzuführen, da der Umfang der Implementierung und der Aufwand zur Analyse nicht im Verhältnis steht. Abweichend wird im Folgenden ein Blick auf die Aufwandsverteilung der Implementierung auf einem abstrakten Niveau durchgeführt.

Diese Betrachtung partitioniert die Implementierung hinsichtlich ihrer Charakteristiken (siehe Kapitel 3.6, S. 78) und stellt sie anschließend gegenüber. Die Zuordnung der Implementierung wird mit dem selben Analysewerkzeug durchgeführt, jedoch aufgrund des Umfangs der Implementierung auf einem groben Level durchgeführt. Die Zuweisung der Charakteristiken wird auf Basis der Ordnerstruktur, welche im Fall von Java auf die Programmpakete zurückzuführen sind, durchgeführt. Zum Beispiel werden Implementierungen im Ordner *requests* und *responses* als Kommunikationsobjekt und Implementierungen im Ordner *handlers* als Teil des Controllers charakterisiert. Ziel dieser Betrachtung ist es eine Darstellung der Anteilsverteilung zu erzeugen, welche eine Aussage über die Partitionierung der Implementierung zulassen.

Die Auswertung kann in Anlehnung an Kapitel 3.6 (S. 78) mithilfe der folgenden Funktion $v_1(s, c, k)$ durchgeführt werden. Abweichend enthält diese keine Betrachtung je Funktionalität. Alternativ wird an dieser Stelle hinsichtlich Client- und Server-Implementierung differenziert.

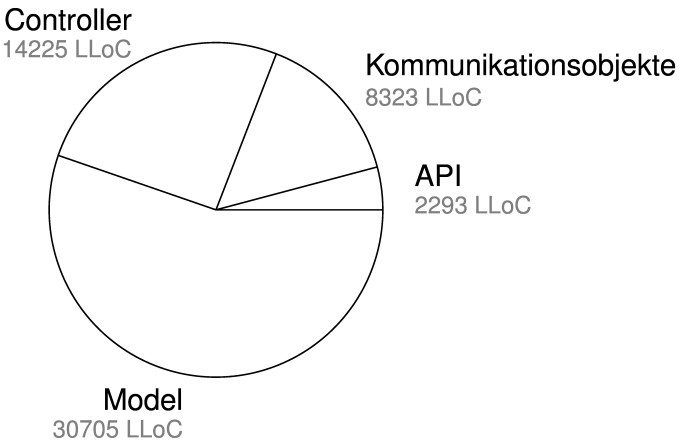



**Abbildung 7.12: Verhältnis des mittels $v_1(8, c, 2)$ für $c \in C$ ermittelten Aufwands der Server-Implementierung in Viechers.**

$$
\begin{aligned}
v_1(s, c, k) = |\{x | x \in Z \wedge \\
&\text{Strategy}(x) = s \wedge \\
&\text{Characteristic}(x) = c \wedge \\
&\text{Component}(x) = k \wedge \\
&\text{Countable}(x) = 1\}|
\end{aligned}
\tag{7.12}
$$

Abweichend von der Betrachtung in Kapitel 3.6 (S. 78) ist der Definitionsbereich von $s$, $c$ und $k$ der zu analysierenden Merkmale je Zeile wie folgt bestimmt: Stategien $S$ begrenzt sich auf die Ausführungsumgebung (8); für die Charakteristiken $C$ werden die bereits herausgestellten Merkmale Model (1), API (2), Controller (3), Kommunikationsobjekte (4), Message Handler (5), Konfiguration (7) und Socket (8), sowie das zusätzliche Merkmal View (9) verwendet; und für die Komponente $K$ werden die Implementierungen für Client (1) und Server (2) genutzt.

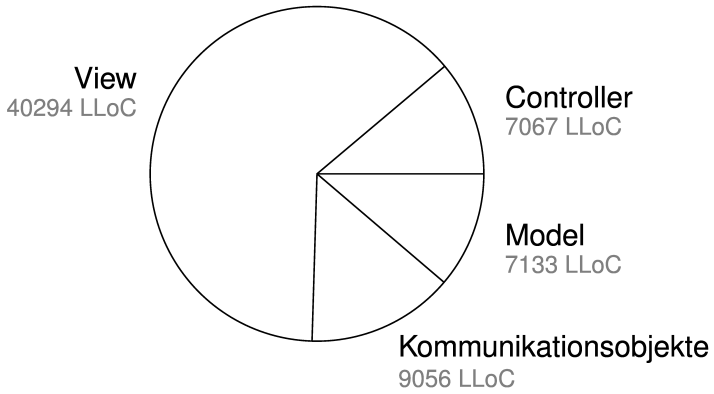



**Abbildung 7.13: Verhältnis des mittels $v_1(8, c, 1)$ für $c \in C$ ermittelten Aufwands der Client-Implementierung in Viechers.**

$$S := \{8\} \tag{7.13}$$

$$C := \{1, 2, 3, 4, 5, 6, 7, 8, 9\} \tag{7.14}$$

$$K := \{1, 2\} \tag{7.15}$$

$$s \in S, c \in C, k \in K \tag{7.16}$$

Abbildung 7.12 zeigt das Ergebnis der Betrachtung für die Implementierung des Servers. Für die Bereitstellung der Funktionalitäten für den Client sind zzgl. zum Model 24.841 LLoC notwendig, welches zum Model in einem Verhältnis von 1:0,809 stehen. Abbildung 7.13 zeigt das Ergebnis der Betrachtung für die Implementierung des Clients. Zur Realisierung des Views, der Verwaltung des lokalen Zustands und den Abbildern der Kommunikationsobjekte sind zusätzlich 23.256 LLoC notwendig. Diese stehen in einem Verhältnis von 1:0,577 zur Implementierung des Views.

## 7.4 Analyse des Abbildungsformates

Das Abbildungsformat, welches in Kapitel 5.5.3 (S. 107) vorgestellt und in Kapitel 6.2.2 (S. 138) beschrieben wird, soll im Folgenden mit den in Kapitel 3.1 (S. 31) beschriebenen Abbildungsformaten verglichen werden. Dabei wird

zum einen auf das Datenvolumen und Verhältnis zwischen Nutz- und Metadaten eingegangen, als auch die Geschwindigkeit zur Abbildung von Instanzen in Datenströme, sowie umgekehrt.

### 7.4.1 Nutz- und Strukturdaten Verhältnis

Die Betrachtung des Nutz- und Strukturdaten Verhältnisses bezieht sich auf die Analysen des Kommunikationsobjektes aus Abbildung 3.2. Dieses Beschreibt eine Instanz der Klasse *TestObject*, welches verschiedene Ganz- und Fließkommazahlen, sowie eine Reihung vom Typ Zeichenkette und eine Reihung vom Typ *Fruit* enthält. Diese Instanz wurde mit den Abbildungsformaten XML, JSON und JOS, sowie ProtoBuf, SmartFoxServer und dem der konstruierten Ausführungsumgebung abgebildet und in den Anhang C (S. 239), Anhang D (S. 241), Anhang E (S. 243), Anhang G (S. 247), Anhang F (S. 245) und Anhang H (S. 249) dargestellt. Die Formate XML, JSON und JOS, sowie SmartFoxServer betrachtend, konnte festgestellt werden, dass mit jedem Kommunikationsobjekt ein nicht unerheblicher Anteil an Strukturdaten transportiert wird, welche zur Beschreibung der transportierten Daten genutzt werden. Dieser Overhead ist insbesondere bei Architekturen mit offenen Schnittstellen zu unabhängigen Drittystemen notwendig. Für das Szenario einer Spielentwicklung wurde im Rahmen der Ausführungsumgebung ein Abbildungsformat vorgestellt, welches diesen Overhead auf ein minimum reduziert, wie es z. B. im Fall des Abbildungsformates ProtoBuf realisiert wurde, ohne dabei den Umweg über eine zusätzliche Beschreibungssprache zu fordern. Der in Anhang H (S. 249) dargestellte Datenstrom nutzt einen 4 Byte großen Header zur eindeutigen Identifikation des Datenpaketes. Mit dem identifizierten Datenpaket können anschließend die im Datenmodell enthaltenen Informationen genutzt werden, um das Datenpaket zu interpretieren.

Abbildung 7.14 stellt die sechs Strategien zum Abbilden von Kommunikationsobjekten gegenüber und differenziert je Strategie in die Charakteristiken *Nutzdaten*, *Längen* und *Strukturdaten*. Die spezielle Charakteristik *Länge* beschreibt eine Nutzdaten-abhängige Information, welche sich im Fall von XML und JSON indirekt durch den Syntax ergibt.

Wird die Charakteristik *Länge* den Strukturinformationen zugeordnet, ergibt sich ein Nutzdatenanteil von 23 % für XML, 33 % für JSON, 22 % für

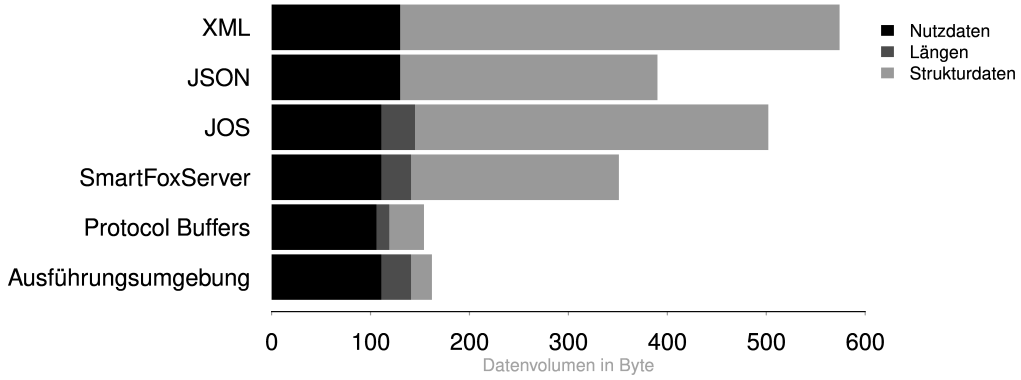



**Abbildung 7.14: Darstellung der Anteile von Nutz- und Strukturdaten innerhalb der Pakete die durch die Abbildung in XML, JSON und JOS, sowie SmartFoxServer und dem Format der Ausführungsumgebung entstehen.**

JOS, 69 % für ProtoBuf und 32 % für den SmartFoxServer. Mithilfe des Abbildungsformates der Ausführungsumgebung konnte ein Nutzdatenanteil von 69 % erreicht werden. Wird die Charakteristik *Länge* den Nutzdaten zugeordnet, erhöht sich der Wert auf 87 %.

Während die relative Betrachtung zwischen Nutz- und Strukturdatenverhältnis des Abbildungsformates der Ausführungsumgebung durchschnittlich eine Verdopplung im Bezug zu den Strategien XML, JSON, JOS und SmartFoxServer zeigt, sowie ein vergleichbares Ergebnis zu der Strategie ProtoBuf hat, visualisiert die Abbildung 7.14, dass das Datenpaket im Beispiel deutlich kompakter abgebildet wird, jedoch minimal umfangreicher als die Strategie ProtoBuf ist. Mit einem Gesamtvolumen von 162 Bytes ist das Abbildungsformat im gezeigtem Anwendungsfall weniger als halb so groß wie die Alternative JSON mit 390 Byte.

### 7.4.2 Performance der Abbildung

Für eine differenziertere Betrachtung des Abbildungsformates sind mehr Varianten an Kommunikationsobjekten notwendig. In der folgenden Betrachtung werden dafür weitere Kommunikationsobjekte definiert, welche Standarddatentypen kapseln und durch die Formate abgebildet werden sollen. Die folgende Liste beschreibt die verwendete Kommunikationsobjekte:

- **Boolean**
  Ein boolsches Kommunikationsobjekt mit einem Wahrheitswert als Attribut. Der Wert ist zufällig wahr oder falsch.

- **Integer**
  Ein numerisches Kommunikationsobjekt mit einer Ganzzahl als Attribut. Der Wert ist zufällig zwischen 0 und $2^{31}$.

- **Double**
  Ein numerisches Kommunikationsobjekt mit einer Gleitkommazahl als Attribut. Der Wert ist zufällig zwischen 0 und 1.

- **String**
  Ein textuelles Kommunikationsobjekt mit einer Zeichenkette als Attribut. Als Wert wird die Zeichenkettenrepräsentation einer UUID verwendet, welche keine Sonderzeichen enthält und eine konstante Länge aufweist.

- **Object**
  Ein Kommunikationsobjekt mit einem Zeiger auf ein weiteres Kommunikationsobjekt als Attribut. Das gekapselte Kommunikationsobjekt ist vom Typ *Integer*.

- **Boolean[x]**
  Ein Kommunikationsobjekt mit einer Reihung vom Typ eines Wahrheitswertes als Attribut.

- **Integer[x]**
  Ein Kommunikationsobjekt mit einer Reihung vom Typ einer Ganzzahl als Attribut.

- **Double[x]**
  Ein Kommunikationsobjekt mit einer Reihung vom Typ einer Gleitkommazahl als Attribut.

- **String[x]**
  Ein Kommunikationsobjekt mit einer Reihung vom Typ einer Zeichenkette als Attribut.

- **Object[x]**
  Ein Kommunikationsobjekt mit einer Reihung vom Typ einer weiteren Kommunikationsobjekt als Attribut.

- **Complex**
  Ein Kommunikationsobjekt mit mehreren Attributen mit unterschiedlichen Typen.

- **Test**
  Das Kommunikationsobjekt aus Abbildung 3.2.

Für diesen Versuch konnte das Abbildungsformat für die SmartFoxServer Alternative nicht betrachtet werden. Die in diesem Framework verankerte Algorithmik ließ sich nicht für eine dedizierte Analyse herauslösen.

Die verbleibenden Alternativen XML, JSON, JOS, ProtoBuf und das Abbildungsformat der Ausführungsumgebung wurden in einem Messwerkzeug eingebunden, welches Testreihen in einer bestimmbaren Menge an Durchläufen ausführt. Der für die Durchführung der Messvorgänge notwendige Datenspeicher wird vor der Messwerterhebung bereitgestellt um Verzerrungen zu minimieren. Jeder Durchlauf enthält eine initiale Ausführung des Abbildungsprozesses um zu vermeiden, dass Ladeprozesse von fehlenden Klassen und Allokationen von Speicherblöcken innerhalb der Java VM zu Verzögerungen führen. Für die Vergleichbarkeit der Ergebnisse wurden zudem alle Messungen auf der selben physischen Maschine durchgeführt.

Messwerte werden in Gruppen mit $n$ Serialisierungs- und Deserialisierungsvorgängen je Strategie erhoben. Die Erhebung umfasst dabei $m$ Messwertgruppen. Gemessen wird die Ausführungszeit $t_1$ für die Serialisierung und $t_2$ für die Deserialisierung je Strategie und Gruppe, sowie einmalig die Paketgröße $s$ je Kommunikationsobjekt und Strategie. Für die Bewertung wurden je Gruppe 10.000 Serialisierung und Deserialisierungen durchgeführt. Insgesammt wurden 100 Gruppen an Messwerten für die aufgeführten Kommunikationsobjekte erzeugt. Kommunikationsobjekte mit variablen Größen für gekapselte Reihungen wurden mit $x \in (5, 10, 25, 50, 75, 100, 125, 150, 175, 200)$ abgebildet. Für die Aggregation der Daten wurde der arithmetische Mittelwert bestimmt, wobei die resultierenden Gruppen um die 10 % der niedrigsten und die 10 %

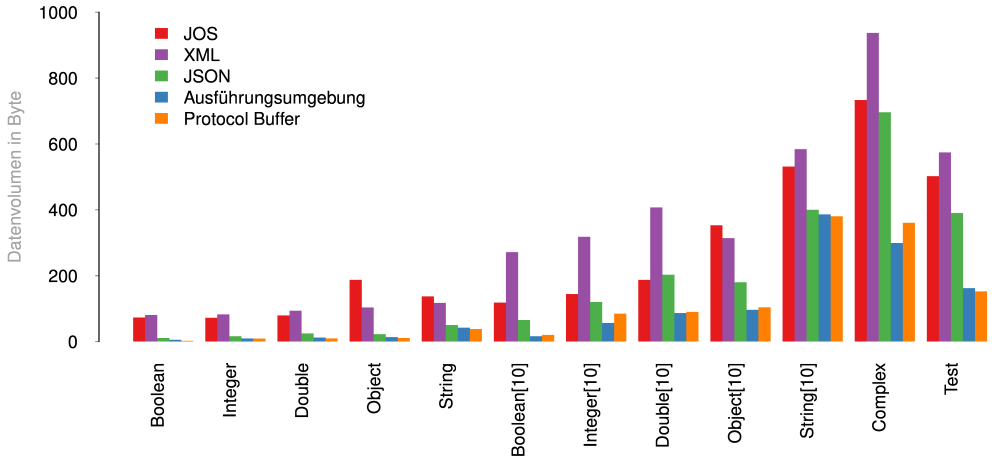



**Abbildung 7.15: Datenpaketgrößen aller Testobjekte in Byte.**

der höchsten Messwerte bereinigt wurden. Im Anhang T (S. 297), Anhang U (S. 299) und Anhang V (S. 301) können die Messergebnisse eingesehen werden.

Abbildung 7.15 zeigt die resultierenden Paketgrößen $s$ je Kommunikationsobjekt im Vergleich für die verschiedenen Strategien. Das Abbildungsformat der entwickelten Ausführungsumgebung erzeugt kompakte Datenpakete, was die Annäherung an die Alternative ProtoBuf zeigt und in dem effizienteren Verhältnis zwischen Nutz- und Strukturdaten begründet ist.

Abbildung 7.16 zeigt die durchschnittliche Serialisierungszeit eines Kommunikationsobjektes, ermittelt nach der Vorschrift 7.17. Die Messung zeigt eine effiziente Ausführungszeit bezüglich der nativen Implementierung in Java und dem effizienten Abbildungsformat ProtoBuf. In einigen Fällen übertrifft die Ausführungszeit die der Alternativen, was in der Praxis jedoch in diesem Maß nicht relevant ist.

$$y_{Abb7.16} = \frac{t_1}{n \cdot m} \tag{7.17}$$

Für einen Einblick in das Verhalten der Abbildungsformate im Fall von variablen Größen eines Types werden die Kommunikationsobjekte mit Reihungen genutzt. Deren gekapseltes Attribut kann in verschiedenen Durchläufen mit variablem Umfang betrachtet werden um einen Einblick bei steigender Anzahl von Elementen zu erhalten. Abbildung 7.17 visualisiert diese Betrachtung für das Kommunikationsobjekt *Integer[x]* für $x \in (5, 10, 25, 75, 100, 125, 150, 175, 200)$. Die Darstellung zeigt, dass sich die

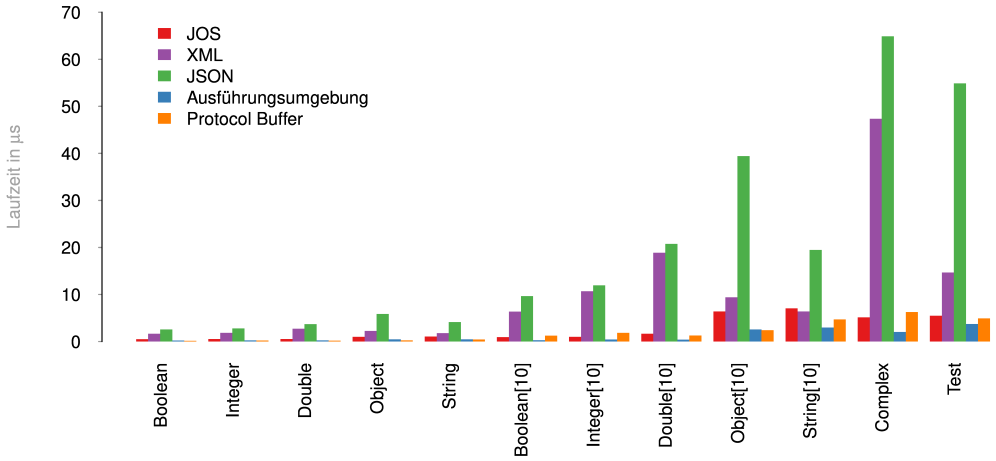



**Abbildung 7.16: Serialisierungszeiten der Kommunikationsobjekte.**

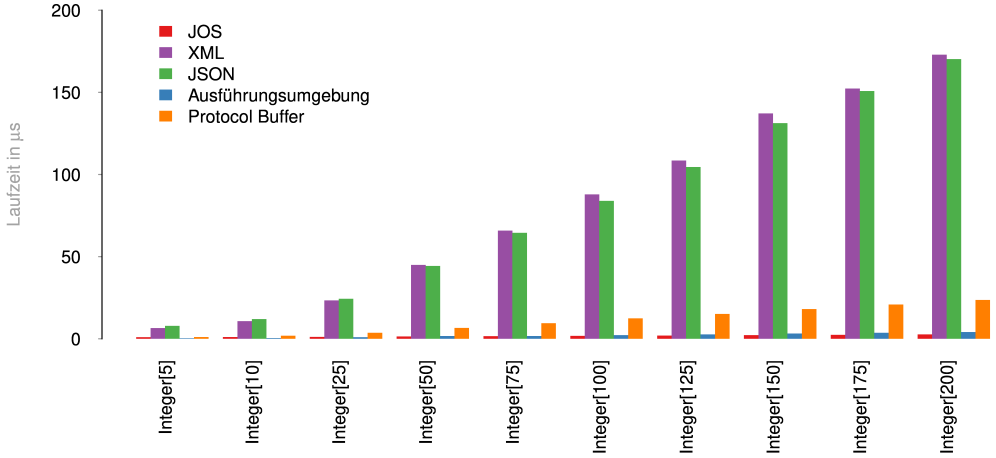



**Abbildung 7.17: Serialisierungszeit bei steigender Datengröße des Kommunikationsobjektes *Integer[x]*.**

Laufzeiten für eine Abbildung mit JSON und XML stark erhöhen, die für JOS und das Abbildungsformat der entwickelten Ausführungsumgebung annähren und schwach zunehmen. Anhang R (S. 293) zeigt zusätzlich weitere Betrachtungen für die Kommunikationsobjekte mit Reihungen vom Typ Zeichenkette, Wahrheitswert, Ganzzahl und Fließkommazahl. Dabei sind die Ergebnisse ähnlich und zeigen vergleichbare Geschwindigkeiten zwischen der entwickelten Ausführungsumgebung und JOS.

Abbildung 7.18 fasst die vorangegangenen Betrachtungen zusammen und vergleicht die Laufzeit zur Serialisierung bezüglich der Paketgröße. Einen linearen Trendverlauf auf die Messungen anwendend, zeigt das Abbildungsformat

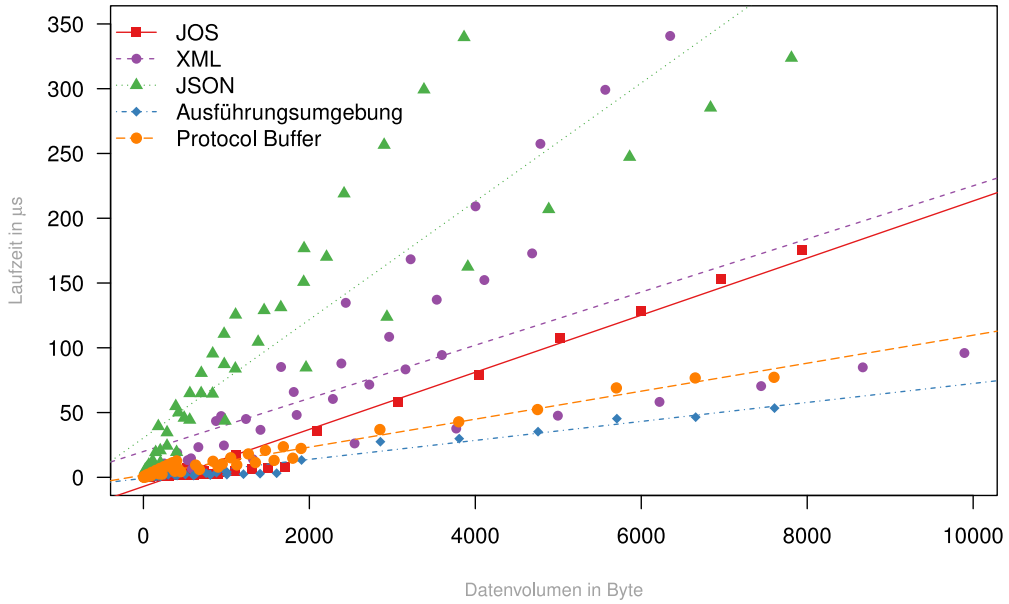



**Abbildung 7.18: Serialisierungszeit bezüglich Datengröße.**

der Ausführungsumgebung ein leicht effizientere Entwicklung bei der Serialisierungsgeschwindigkeit bzgl. ProtoBuf, während die Alternativen JOS, JSON und XML einen steilen Anstieg bei steigender Paketgröße aufweisen.

In Anlehnung an die Betrachtung der Ausführungszeit zur Serialisierung der verschiedenen Strategien zur Abbildung der Kommunikationsobjekte, wird im Folgenden die Ausführungszeit zur Deserialisierung der verschiedenen Strategien betrachtet. Diese ergibt sich nach Vorschrift 7.18, welche vergleichbar zur Vorschrift 7.17 ist.

$$y_{Abb7.19} = \frac{t_2}{n \cdot m} \tag{7.18}$$

Abbildung 7.19 stellt, wie Abbildung 7.16 im Fall der Serialisierung, die Dauer zur Ausführung der Deserialisierung der verschiedenen Kommunikationsobjekte gegenüber. Diese Dauer zur Ausführung ist für das Abbildungsformat XML aufwendig und im Fall des Kommunikationsobjektes *Boolean[200]* 200 mal langsamer als die Dauer des Abbildungsformates der Ausführungsumgebung. Im Durchschnitt liegt das Abbildungsformat XML bei einer ca. 114 mal langsameren Ausführungsdauer. Wird die Ausführungsdauer erneut mit dem Abbildungsformat JOS verglichen, ergibt sich eine ca. 2,6 mal schnellere

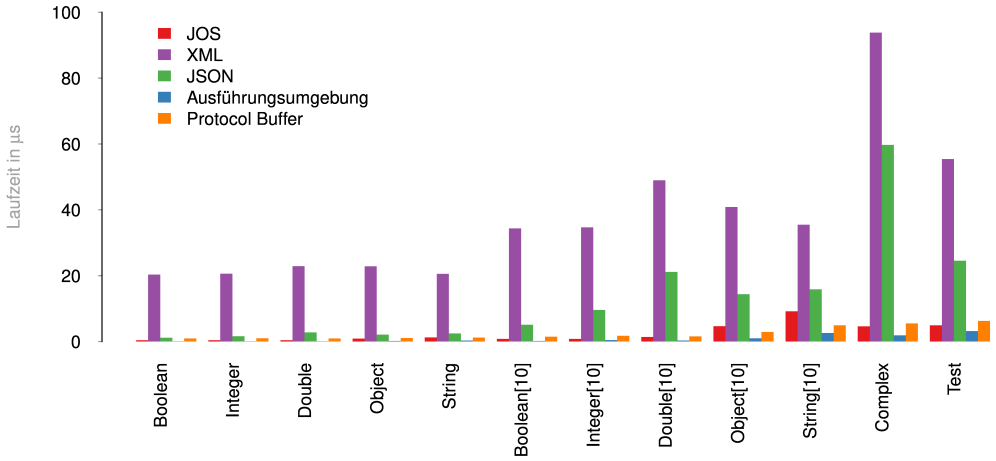



**Abbildung 7.19: Deserialisierungszeit von Kommunikationsobjekten.**

Deserialisierung für das Abbildungsformat der Ausführungsumgebung.

Die Betrachtung fortführend stellt Abbildung 7.20 die Ausführungsdauer für das Kommunikationsobjekt *Integer[x]* mit $x \in (5, 10, 25, 75, 100, 125, 150, 175, 200)$ dar. Die Alternativen zeigen sich im Vergleich teilweise langsamer. Im Detail ist JOS ca. 1,2 mal schneller, sowie ProtoBuf ca. 3,1 mal, JSON ca. 25 mal und XML ca. 57 mal langsamer als das Abbildungsformat der Ausführungsumgebung. Anhang S (S. 295) zeigt zusätzlich weitere Betrachtungen für die Kommunikationsobjekte mit Reihungen vom Typ Zeichenkette, Wahrheitswert, Ganzzahl und Fließkommazahl. Dabei sind die Ergebnisse ähnlich und zeigen vergleichbare Unterschiede bei den Ausführungszeiten.

Abbildung 7.21 vergleicht abschließend die Ausführungszeiten mit dem Volumen der resultierenden Datenpakete. Das Abbildungsformat der Ausführungsumgebung zeigt Vorteile in der Ausführungszeit gegebenüber den alternativen Strategien, was auf die Effizienz bei der Deserialisierung mithilfe von generisch erzeugten Konvertierungsvorschriften in Kombination mit einem kompakten Abbildungsformat schließen lässt.

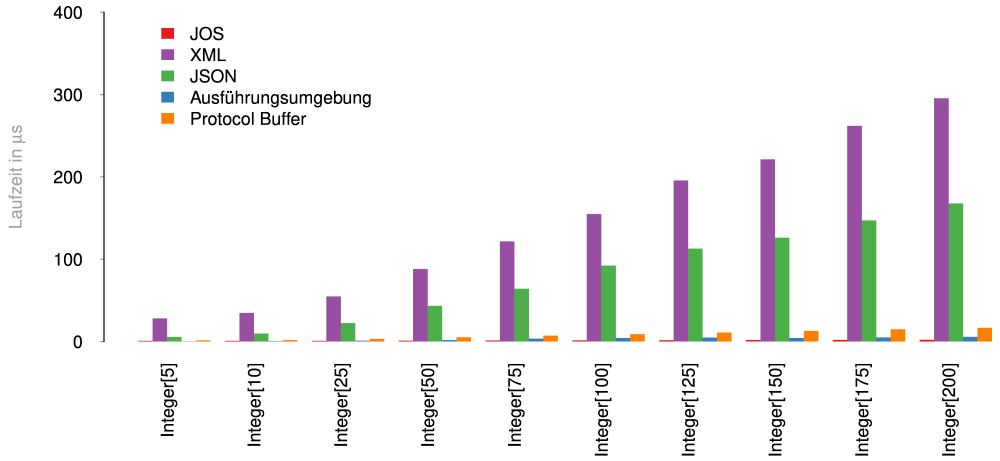



**Abbildung 7.20: Deserialisierungszeit bei steigender Datengröße des Kommunikationsobjektes *Integer[x]*.**

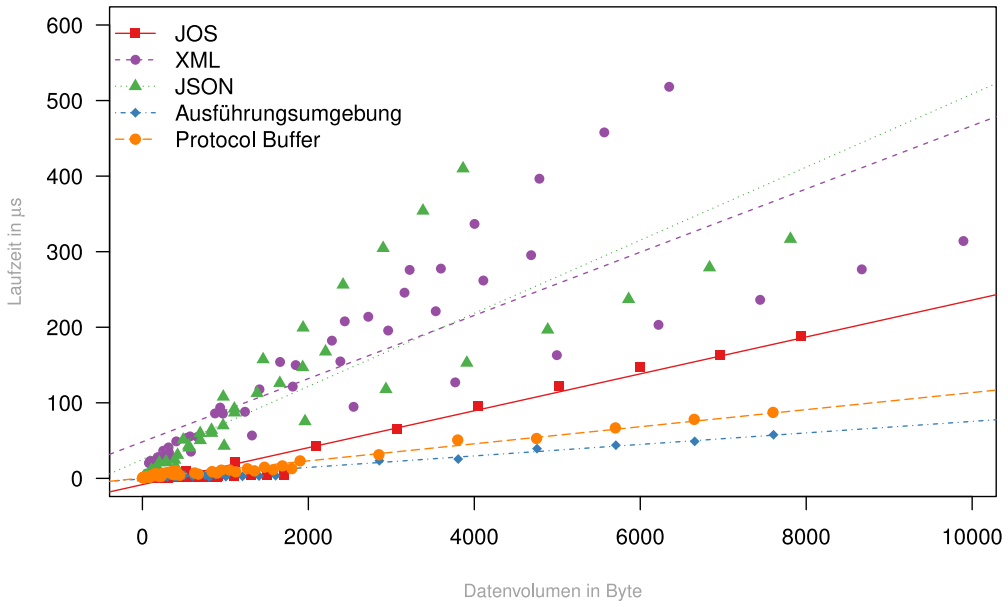



**Abbildung 7.21: Deserialisierungszeit bezüglich Datengröße.**

# 8. Diskussion der Ergebnisse

Die Realisierung einer Architekturen für ein MMOG, wie sie aktuell in Spielprojekten zu finden ist, umfasst eine Vielzahl an Aufgaben, bei denen redundante Schritte vermutet wurden. Im Fokus stand dabei die Kommunikationsinfrastruktur, welche zwischen Spieler (Client) und zentraler Instanz (Server) die Verarbeitung von Anfragen und die Verteilung von Antworten organisiert. Zur Vereinfachung der Entwicklung wurde eine Ausführungsumgebung realisiert, welche die identifizierten redundanten Elemente adressiert und Aufwände minimiert. Die dafür formulierten Behauptungen (siehe Kapitel 4, S. 89) unterstellen, dass sich Spiele auf ein abstraktes Spielmodell reduzieren lassen (vgl. These 1) und diese in aktuellen Architekturen einen Overhead zur Realisierung aufweisen (vgl. These 2) der zur Realisierung der Infrastruktur notwendig, aber nicht nötig, bzw. rendundant erscheint. Eine Ausführungsumgebung, die diese Redundanzproblematik adressiert und in gängigen Umgebungen, wie dem JBoss, ausführbar ist (vgl. These 3), sollte eine Entwicklung ermöglichen, in der die als redundant identifizierten Elemente vermieden werden (vgl. These 4a). Zusätzlich erlaubt eine sich selbst verwaltende Kommunikationsinfrastruktur die Anwendung eines proprietären Formates um die Ausführungsgeschwindigkeit bei der Abbildung und das Datenvolumen zu optimieren (vgl. These 4b).

Die vorangegangene Analyse der Spielprojekte, des Aufwands und des Abbildungsformates haben die Konzepte der Ausführungsumgebung unter verschiedenen Aspekten betrachtet. Im Folgenden werden die Ergebnisse hinsichtlich der erreichten Ziele diskutiert und bewertet.

In der Analyse der Spielprojekte konnten sechs Konzepte aus unterschiedlichen Genres betrachtet werden. Jedes Projekt konnte in der Betrachtung auf die im abstrakten Spielmodell definierten Elemente Spielwelt, Spielobjekt, Spielaktion und Spielereignis generalisiert werden. Dabei sind im Spielprojekt nicht alle Entitäten notwendigerweise auf jene Elemente zurückzuführen. Dies

trifft z. B. nicht zu, wenn die Entitäten zur Beschreibung komplexer Datentypen dienen. Darüber hinaus ist die untersuchte Projektmenge nicht ausreichend umfangreich, um die Aussage zu generalisieren. Hinsichtlich der These, dass sich Spielkonzepte auf einen EGM-Standard reduzieren lassen, trifft die Betrachtung zu. Eine Verallgemeinerung, dass sich jedes Spielkonzept reduzieren lässt, ist mit der aktuellen Menge an Projekten nicht möglich.

Zusätzlich zur Betrachtung der Spielmodelle, war für die Analyse der Spielprojekte interessant, welche Funktionalitäten bereitgestellt werden konnten. Im Fall der Studentenprojekte ist kein Anwendungsfall entstanden, in dem die automatisierte Ableitung aus dem Spielmodell zur Interaktion mit dem Spielmodell ausgehend vom Client nicht den Anforderungen gerecht wurde. Das Konzept konnte durchgängig angewendet werden und den Implementierungsaufwand auf das Spielmodell, sowie die Realisierung der Darstellung im Client reduziert werden. Dabei sei jedoch kritisch zu betrachten, dass die Studentenprojekte abschließend einen prototypischen Zustand erreicht haben und somit nicht mit weiterführenden Funktionalitäten, wie Bezahl- und Authentifizierungsdiensten, in Kontakt gekommen sind. Im Fall von Viechers, welches zur Betrachtung als komplexes Spielprojekt zur Verfügung stand, wurde aufgrund der Nichtverfügbarkeit der entfernten Aufrufe mittels manuell implementierter Anfrageverarbeitung und Kommunikationsobjekte gearbeitet. Dies ermöglichte einen Einblick in die verschiedenen Ausprägungen der Anfragen, die im Rahmen eines komplexen Anwendungsfalles entstehen können. Für diese Analyse wurden die Anfrageverarbeitungen hinsichtlich verschiedener Merkmale betrachtet, um diese als Aufrufe auf das Spielmodell zu identifizieren. Im Ergebnis sind von den 119 Anfrageverarbeitungen 91 % auf Aufrufe auf das Spielmodell zurückzuführen. Die verbleibenden 11 Anfrageverarbeitungen sind Instruktionen mit Kombinationen aus optionalen Modell- und Dienstaufrufen, z. B. der Aufruf von Chat-, Nachrichten-, Bezahl- und Authentifizierungsdiensten, die nicht in einem exklusiven Zusammenhang mit dem Spielmodell stehen. Bezüglich der These 4a lässt sich feststellen, dass die Entwicklung, insbesondere bei exklusiver Interaktion mit dem Spielmodell, möglich ist. Ein vollständiges Vermeiden jedoch bei ausreichend komplexen Systemen nicht zwingend gegeben ist. Eine mehrschichtige Verarbeitungslogik, welche die manuelle Anfrageverarbeitung in Sonderfällen ermöglicht, hilft diese Anfrageverarbeitungen zu

realisieren.

Die Analyse des Aufwands wurde verwendet, um einen Einblick in die Entwicklung des initial festgestellten Overheads in verschiedenen Spielprojekten zu betrachten. Hierfür standen drei Projekte zur Verfügung: (1) die Beispiel-Implementierung, (2) das Studentenprojekt MMO-Meischta sowie (3) Viechers. Die Beispiel-Implementierung, welche zur initialen Betrachtung in verschiedenen Strategien realisiert wurde, wurde im Rahmen der Analyse erneut aufgegriffen, um die Strategie „Ausführungsumgebung“ erweitert und erneut verglichen. Hierbei konnte gezeigt werden, dass im Fall des Beispiels das durchschnittliche Verhältnis zwischen Spielmodell und Overhead von 1:3,01 auf 1:0,113 reduziert wurde, wenn das Beispiel mithilfe der Ausführungsumgebung realisiert wird. Diesen Ansatz wiederholend wurde das Studentenprojekt MMO-Meischta gewählt, bereinigt und in drei alternativen Strategien, in Anlehnung an das Beispiel, realisiert. Dabei konnte erneut ein stetig steigender Overhead je bereitgestellter Funktionalität gezeigt werden, als auch die Reduktion des Overheads von durchschnittlich 1:1,422 auf 1:0,033 bezüglich dem Spielmodell. Als dritte Betrachtung wurde der komplexe Anwendungsfall Viechers herangezogen, welcher aufgrund des Umfangs weniger detailreich betrachtet werden konnte. Die grobe Klassifizierung und Bewertung der Implementierung ergab ein Verhältnis von 1:0,809 zwischen Overhead und Spielmodell. Die drei Betrachtungen deuten eine durchschnittliche Verdopplung des Aufwands bei Bereitstellung aller Funktionalitäten eines Spielmodells. Dabei fallen insbesondere bei steigender Komplexität die Anfrageverarbeitung / Ereignisverarbeitung des Controllers und Kommunikationsobjekte ins Gewicht. Dieser Overhead konnte in den Anwendungsfällen Stickman, Terra-Life-Evolution, Hack'N'Slay, Kokosnussbikini und MMO-Meischta auf ein Minimum reduziert, bzw. vermieden werden. Bezüglich der These 2 legt dies nah, dass der Aufwand messbar existiert und in den betrachteten Projekten vermeidbar ist.

Die Analyse des Abbildungsformates dient zur Betrachtung eines Ansatzes zur Abbildung von Daten im Fall von synchronen Entwicklungsständen zwischen Client und Server. Während der Einsatz von XML und JSON denkbar wäre, stand die Frage im Raum, ob sich effizientere Formate zur Abbildung realisieren lassen, die in diesem Szenario angewendet werden können. Im Ergebnis realisiert die Ausführungsumgebung ein Abbildungsformat, welches die

Daten ohne zusätzliche Informationen für Attribute abbildet und einen 4 Byte großen Header zur Identifikation der folgenden Daten verwendet. Im Rahmen der Analyse wurde das Abbildungsformat in zwei Varianten betrachtet. Die erste Variante greift die Beispielklassen aus Abbildung 3.2 auf und vergleicht Nutz- und Strukturdaten. Die Analyse zeigt ein kompaktes Datenvolumen, welches mit dem effizienten Format ProtoBuf vergleichbar ist und verglichen mit herkömmlichen Formaten sehr viel kompakter ausfällt. Ergänzend wurde das Verhältnis zwischen Nutz- und Strukturdaten betrachtet. Dieses Verhältnis ist im Ergebnis ebenso dem Format ProtoBuf ähnlich und zeigt ein Nutzdatenanteil von ca. 69 %. Dem folgend wurde in der zweiten Varianten zur Analyse weitere Kommunikationsobjekte als Beispiel definiert, welche gängige Datentypen abbilden, und in einer Versuchsreihe bezüglich Serialisierungs- und Deserialisierungszeit betrachtet wurden. Die generisch erzeugten Abbildungsinstruktionen zur Serialisierung und Deserialisierung konnten im Vergleich zu XML und JSON bessere Ausführungszeiten erreichen. Dabei Stand bei der Betrachtung nicht nur die reine Abbildung von Daten in Datenströme im Zentrum der Betrachtung, sondern die Erzeugung von Kommunikationsobjekten. Im Fall des modernen und effizienten Formates ProtoBuf konnte das Abbildungsformat der Ausführungsumgebung eine Verbesserung erreichen, da keine Wrapper Objekte notwendig sind. Bezüglich der These 4b zeigt die Analyse, dass ein metadatenfreies Abbildungsformat zur Serialisierung und Deserialisierung von Nachrichten mit minimalem Overhead erreichbar ist und innerhalb der realisierten Projekte stabil verwendet werden kann.

Für die These 3 ist keine weiterführende Betrachtung erforderlich. Begründet in dem gewählten Architekturdesign, den getroffenen Entscheidungen und der Abhängigkeit der Ausführungsumgebung zur Java EE, ist dessen Ausführbarkeit in einer gängigen Ausführungsumgebung implizit gegeben. Zusätzlich zeigen die Projekte die Ausführbarkeit von Spielkonzepten in dem realisierten Container innerhalb der Ausführungsumgebung. Das Spielprojekt Viechers zeigt hierbei eine ca. fünfjährige Laufzeit innerhalb des „Game Model Container“ zur Bereitstellung des Modells und wird hierfür in einem JBoss 5 betrieben.

# 9. Fazit und Ausblick

Ziel der Arbeit ist es Elemente einer Softwareentwicklung für MMOGs, zur Realisierung der Kommunikationsinfrastruktur, zu identifizieren sowie wiederkehrende bzw. redundante und somit vermeidbare Elemente in einer Ausführungsumgebung zu adressieren. Unter der Motivation des Spielkonzepts Viechers wurden im Vorangegangenen die Begrifflichkeiten Spiel, Spielen, Videospiele, MMOGs und browserbasierte MMOGs differenziert und Anforderungen an jene abgeleitet. Darauf aufbauend entstand ein abstraktes Spielmodell, das EGM, welches als Basis für ein minimales Beispiel diente, welches fünf Funktionalitäten über eine API definiert und anschließend von einem Client genutzt werden müssen, um die Interaktionen und die Visualisierung des Spiels zu realisieren. Die initialen Betrachtungen verwendend, sind im Folgenden im Rahmen einer Stand der Technik Betrachtung mögliche Strategien zur Realisierung dieses minimalen Beispiels erarbeitet und realisiert wurden. Diese Strategien umfassen einen proprietären TCP-Server in Java, einen REST-Service in JavaScript und Java sowie einen WebSocket-Service in JavaScript und eine SmartFoxServer-basierte Strategie zur Anwendung einer speziellen browserbasierten MMOG-Middleware in Java. Die erstellten Implementierungen der fünf Strategien wurden anschließend mit einem im Rahmen der Arbeit erstellten Werkzeug zur Klassifizierung von Quellcode und Ermittlung des Aufwands auf Basis der in dem COCOMO II Kostenmodells definierten LLoC analysiert, um einen Eindruck über die Aufwandsentwicklung zur Bereitstellung von Funktionalität zu erhalten. Für die Klassifizierung wurden acht Charakteristiken herausgestellt, die neben dem Spielmodell klassische Elemente einer Architektur zur Realisierung eines MMOGs aufweisen, z. B. Kommunikationsobjekte, Message Handler, Controller und Socketserver. Durch die Bewertung des Beispiels und dessen Analyse konnte ein Overhead zur Bereitstellung von Funktionalität identifiziert werden, welcher zzgl. zur Realisierung des Spielmodells notwendig

ist und durchschnittlich ein Verhältnis von 1:3,01 aufweist. In Ergänzung dazu wurden im Rahmen der Betrachtung des Stands der Technik Abbildungsformate für den Austausch von Nachrichten zwischen Client und Server analysiert. Standardisierte und klassische Formate wie JSON, XML und JOS zeigen hierbei ein niedriges Verhältnis zwischen Nutz- und Strukturdaten. Im Fall eines MMOGs werden diese Informationen bei jedem Nachrichtenaustausch transportiert und expandieren das Datenvolumen unnötig. Aktuelle Formate wie ProtoBuf adressieren dies, werden jedoch mit einem Fokus auf Nachrichten umgesetzt.

Aus der Betrachtung der Grundlagen ergaben sich fünf Thesen, die im Folgenden analysiert werden sollten. These 1 beschreibt die Möglichkeit zur Ableitung eines abstrakten Modells für Spiele, welche argumentativ im Rahmen der Definition des Spielmodells bereits erstellt wurde. These 2 adressiert den vermuteten Overhead bei der Realisierung von Spielen, welcher initial in der Beispiel-Implementierung gezeigt wurde. These 3 greift diese Situation auf und behauptet, dass ein „Game Container“ für Spielmodelle innerhalb gängiger Ausführungsumgebungen lauffähig ist. Abschließend werden technische Thesen zum Game Container formuliert, welche die Reduktion des Overheads bei der Spielentwicklung betreffen (vgl. These 4a) und eine effiziente Realisierung eines Abbildungsformates (vgl. These 4b).

Der Game Container, im Folgenden Ausführungsumgebung für Spielmodelle (kurz Ausführungsumgebung) genannt, wird in einer Architektur skizziert. Die Architektur beschreibt insbesondere die Randbedingungen, Aufgaben und Anforderungen an eine Ausführungsumgebung die Spielmodelle ausführen soll und die Kommunikationsinfrastruktur aus jenen ableitet. Dabei ist die Architektur nach der Vorlage arc42 [80] dokumentiert und fokussiert auf die Darstellung von Lösungsstrategien, Bausteinen und Entwurfsentscheidungen. Unbeachtet bleiben dabei Laufzeit- und Verteilungssichten, sowie Betrachtungen von Risiken. Die Architektur verwendend wurde die Ausführungsumgebung realisiert und spezifische Elemente, insbesondere Abbildungsformat und generische Komponenten zur Ableitung der Kommunikationsinfrastruktur sowie die Anwendung in Spielprojekten beschrieben. Die Ausführungsumgebung ist als eine Servicekomponente für gängige Ausführungsumgebungen, wie dem JBoss in Java EE, realisiert und organisiert die Ausführung eines Spielmodells, insbe-

sondere deren bereitzustellende Infrastruktur. Die verwendeten Standards und Architektur bedingt, dass die Ausführungsumgebung in einer gängigen Umgebung wie dem JBoss ausführbar ist, und damit ein Spielmodell in diesem Container in dem JBoss ausgeführt werden kann (vgl. These 3). Abschließend wird die resultierende Ausführungsumgebung genutzt, um erneut das Beispiel aus der initialen Betrachtung aufzugreifen um eine sechste Strategie für folgende Analysen zu realisieren. Die Anwendung der Ausführungsumgebung reduziert sich dabei auf die Markierung von Funktionalität innerhalb des Spielmodells.

Zur Analyse der Ausführungsumgebung und zur folgenden Diskussion der Ergebnisse für Rückschlüsse auf die ursprünglichen Thesen wurden im Rahmen der Evaluation verschiedene Betrachtungen durchgeführt. Basis dieser Betrachtungen sind sechs mit der Ausführungsumgebung realisierte Spielprojekte, welche sich im Umfang differenzieren. Als komplexes Beispiel dient dabei das initial motivierende Anwendungsbeispiel Viechers, welches mit einem frühen Stadium der Ausführungsumgebung realisiert wurde. Ergänzend ist ein Demonstrator Stickman entstanden, sowie vier Studentenprojekte im Rahmen von Lehrveranstaltungen, welche die Ausführungsumgebung in einem Projektverlauf anwenden. Der Aufbau des Versuchs stellte sicher, dass die unabhängig skizzierten Ideen in Unwissenheit über spezifische Merkmale der Ausführungsumgebungen entstanden sind. Hiermit sollten unabhängige Spielkonzepte entstehen, welche erst im Folgenden mit diesem Werkzeug realisiert werden, um einen Eindruck über deren Anwendbarkeit zu erhalten.

Für die Analyse der Spielprojekte wurden zwei Bereiche betrachtet: (1) die resultierenden Spielprojekte und Funktionalitäten, sowie (2) der Aufwand zur Realisierung selbiger. In der Analyse der Spielprojekte konnte zum einen dargestellt werden, dass sich die Projekte auf Basis eines abstrakten Spielmodells umsetzen ließen als auch das die Bereitstellung der Funktionalitäten ohne Anwendung eines Kommunikationsprotokolls möglich ist. Dies betrifft Funktionalitäten des Spielmodells. Gesonderte Anfrageverarbeitungen sind für Anfragen bzgl. zusätzlicher Dienste, welche nicht im Modell selbst realisiert sind, notwendig. Die Analyse verwendet hierfür mittels Reverse Engineering erstellte Klassendiagramme zur Betrachtung von Abhängigkeiten zwischen verwendeten Entitätstypen des Spielmodells als auch eine Erfassung der notwendigen Funktionalitäten mittels Zählung. Zusätzlich konnten die im Projekt Viechers

ausschließlich manuell erstellten Anfrageverarbeitungen inhaltlich analysiert werden um einen Vergleich zwischen Aufrufen, die auf das Spielmodell zurückführbar sind und jenen, die zusätzlicher Natur sind, zu erhalten. Die Diskussion zu den Ergebnissen stellt hierbei heraus, dass Spielkonzepte auf ein abstraktes Modell zurückführbar sind, jedoch eine verallgemeinernde Aussage basierend auf der betrachtenden Menge nicht möglich ist (vgl. These 1). Zusätzlich wurde in der Diskussion festgehalten, dass eine Entwicklung insbesondere für die Bereitstellung eines Spielmodells ohne Kommunikationsprotokoll im Sinn einer herkömmlichen Strategie möglich ist (vgl. These 4a).

Für die Analyse des Aufwands wurden drei Aufwandsbetrachtungen durchgeführt: (1) die Beispiel-Implementierung aus der initialen Betrachtung, (2) eine zusätzliche Aufwandsanalyse in Anlehnung an das Beispiel unter Verwendung eines Studentenprojektes und (3) das Spiel Viechers. Für die Bewertungen kommt das entwickelte Werkzeug zur Klassifizierung des Quellcodes und Quantifizierung des Aufwands unter Verwendung der LLoC aus dem COCOMO II Kostenmodell zum Einsatz. Dabei konnte für das Beispiel ausgehend von dem initialen Verhältnis zwischen Spielmodell und Overhead von 1:3,01 mithilfe der Ausführungsumgebung ein Verhältnis von 1:0,113 erreicht werden. Dieser minimale Overhead entsteht insbesondere durch die notwendige Markierung bereitzustellender Funktionalität. Das Vorgehen adaptierend wurde das Studentenprojekt MMO-Meischta von der Abhängigkeit zur Ausführungsumgebung bereinigt und anschließend in drei alternativen Strategien in Java realisiert: ein TCP-Server, ein REST-Service und SmartFoxServer-basierter Server. Die drei zusätzlichen Implementierungen, zzgl. der Implementierung mit der Ausführungsumgebung, wurden anschließend mit demselben Verfahren wie das Beispiel analysiert und ausgewertet. Im Ergebnis konnte für die alternativen Strategien ein durchschnittliches Verhältnis zwischen Spielmodell und Overhead von 1:1,422 gemessen werden. Die Ausführungsumgebung konnte dies auf 1:0,033 reduzieren, was eine signifikante Reduktion des Aufwands darstellt. Wie im Fall des Beispiels reduziert sich der Aufwand auf die Markierung der bereitzustellenden Funktionalität. Abschließend wurde das Projekt Viechers genutzt, um einen Eindruck im Fall einer komplexen Implementierung zu erhalten. Hierbei wurden die manuell implementierten Anfrageverarbeitungen, Kommunikationsobjekte und Infrastrukturelemente mit dem Spielmodell

verglichen und ergaben in diesem Fall ein Verhältnis von 1:0,809 zwischen Spielmodell und Overhead. Alle drei Analysen deuten auf einen Overhead im Fall der manuellen Implementierung hin, welcher durch die Ausführungsumgebung, mittels Abstraktion und Ableitung notwendiger Kommunikationsinfrastuktur aus dem Spielmodell, signifikant reduziert werden kann. Bezüglich der These 2 konnte damit gezeigt werden, dass dieser vermutete Overhead in den betrachteten Spielprojekten existiert und durchschnittlich mit einer Verdopplung des Aufwands zum Spielmodell zu rechnen ist.

Die Analyse wird abgeschlossen mit der finalen Betrachtung des Abbildungsformates. Hierbei wurde in Ergänzung zur initialen Betrachtung von Nutz- und Strukturdatenverhältnis das Abbildungsformat hinzugezogen, um dessen Verhältnis zu analysieren. Die Analyse zeigt ein insgesamt kompaktes Datenvolumen, welches mit dem aktuell effiziente Format ProtoBuf vergleichbar ist und verglichen mit herkömmlichen Formaten sehr viel Kompakter ausfällt. Das Verhältnis zwischen Nutz- und Strukturdaten ist mit dem Format ProtoBuf vergleichbar bei ca. 69 % Nutzdatenanteil. In Ergänzung wurden weitere Kommunikationsobjekte definiert, welche für eine Betrachtung der Ausführungszeit zur Serialisierung und Deserialisierung mittels XML, JSON, JOS und ProtoBuf genutzt wurden. Die Ausführungszeit zeigt, dass das Format der Ausführungsumgebung im Vergleich zu XML und JSON erheblich schneller arbeitet, z. B. im Fall der Deserialisierung mehr als 50 mal so schnell wie im Vergleich zu XML erzeugt mittels JAXB. Die Laufzeit ist geringfügig schneller als jene, welche mit dem aktuellen Abbildungsformat ProtoBuf im Testaufbau erreichbar sind. Die Betrachtung zeigt, bzgl. der These 4b, das eine typensichere Objektserialisierung metadatenfrei, bzgl. der Beschreibung von Datentypen und Attribute, performant möglich ist.

Die Ausführungsumgebung lies sich im Rahmen der Spielprojekte als ein hilfreiches und effizientes Entwicklungswerkzeug nutzen. Diese Umgebung abstrahiert Kommunikation auf ein Niveau, bei der nicht der Nachrichtenaustausch, die Validierung und die Transformation von Nachrichten in Kommunikationsobjekte, sowie der anschließende Aufruf von Modellmethoden im Vordergrund steht. Im Fall der Umgebung reduziert sich die Arbeit auf eine modelllastige Implementierung, in der die Assoziationen, Spezialisierungen und Entitätstypen des Spielkonzepts im Vordergrund stehen. Mittels Erweiterung

des abstrakten Spielmodells und Markierung von Funktionalität, welche von einem Client aufgerufen werden kann, sind zusätzliche Aufwände stark minimiert. Zusätzlich unterstützt das abstrakte Spielmodell die Implementierung, da häufig antreffbare Elemente in der Realisierung eines Spielmodells bereits zur Verfügung stehen.

Die verwendeten Techniken greifen zur Realisierung der Ableitungen und Erzeugung von Kommunikationsinfrastruktur in Kernmechanismen der Java VM ein und generieren Quellcode, der zur Laufzeit eingesetzt werden kann, um Anfragen und Daten zu verarbeiten. Damit werden Optimierungsmechanismen der Java VM erhalten und genutzt. Darüber hinaus sind weitere Vereinfachungen zur Unterstützung der Entwicklung fragwürdig und stoßen an die Grenzen der Sprachmöglichkeiten in Java. Zum Beispiel ist bereits durch die Konstruktion von Proxies aus Klassen (anstatt wie von Java vorgesehen aus Schnittstellen) die Gefahr auf tote Attribute zuzugreifen; oder ein Unterschied ob Klassen über die Spielwelt mittels des Aufrufs einer *create*-Methode oder mittels Sprachkonstrukt *new* instanziiert werden. Je mehr Sonderfälle entstehen, desto größer ist der initiale Lernaufwand und desto abhängiger ist ein Konzept von der Ausführungsumgebung. Ein nächster Schritt könnte eine eigene Sprache zur Implementierung des Spielmodells sein, welches die vermeidbare Instruktionen unterbindet und ein Kompilat erzeugt welches in der Ausführungsumgebung ausgeführt werden kann.

Darüber hinaus sind zukünftig die Betrachtung offener Anforderungen an die Ausführungsumgebung und zusätzliche Analysen möglich. Zu offenen Anforderungen gehören insbesondere die Finalisierung einer Skalierbarkeit der Ausführungsumgebung, welche aktuell in der Architektur berücksichtigt wurde, sowie die Betrachtung von Sicherheitsaspekten, insbesondere Autorisierung und Authentifizierung für den Zugriff auf Funktionalität sowie der Ableitung jener Informationen aus Zusammenhängen im Modell. Für die Analysen ist eine weitere Betrachtung des Overheads möglich, z. B. mit der Überführung weiterer Studentenprojekte in alternativen Strategien zur detaillierten Betrachtung sowie die Anwendung existierende und öffentlich zugängliche Spielkonzepte (z. B. das Open Source Spiele Browserquest von Mozilla) auf die Ausführungsumgebung.

# A. Einfache REST-Server Implementierung mit node.js

```
(function(module) {
  var Position = function(x, y) {
    if(x instanceof Position) {
      y = x.getY();
      x = x.getX();
    }
    Object.defineProperty(this, 'x', {
      enumerable: true,
      get: function() {
        return x;
      }
    });
    Object.defineProperty(this, 'y', {
      enumerable: true,
      get: function() {
        return y;
      }
    });
  };
  var Avatar = function(id, name) {
    var position = new Position(0, 0);
    Object.defineProperty(this, 'id', {
      enumerable: true,
      get: function() {
        return id;
      }
    });
    Object.defineProperty(this, 'position', {
      enumerable: true,
      get: function() {
```

```
31        return position;
32      }
33    });
34    Object.defineProperty(this, 'name', {
35      enumerable: true,
36      get: function() {
37        return name;
38      },
39      set: function(value) {
40        name = value;
41      }
42    });
43    this.move = function(to) {
44      position = to;
45    };
46  };
47  var BasicWorld = function() {
48    var players = {};
49    var ids = 1;
50    Object.defineProperty(this, 'players', {
51      get: function() {
52        return players;
53      }
54    });
55    this.getPlayer = function(id) {
56      return players[id];
57    };
58    this.newAvatar = function(name) {
59      var player = new Avatar(ids ++, name)
60      players[player.id] = player;
61      return player;
62    };
63  };
64
65  module.exports = {
66    BasicWorld: BasicWorld,
67    Avatar: Avatar,
68    Position: Position
69  };
```

```
70 })(module);
```

**Quellcode A.1: JavaScript basierte Implementierung des minimalen Spielmodells.**

```
1  (function(module) {
2    // Create world instance of BasicWorld
3    var model = require('./model');
4    var world = new (model.BasicWorld)();
5
6    var BAD_REQUEST = 400, NOT_FOUND = 404;
7    var avatarApi = {
8      newAvatar: function(data) {
9        if(!data.name)
10         throw {msg: 'missing name', error: BAD_REQUEST}
11       return world.newAvatar(data.name);
12     },
13     getAvatars: function(data) {
14       return world.players;
15     },
16     getAvatar: function(data) {
17       var player = world.players[data.id];
18       if(!player)
19         throw {msg: 'not found', error: NOT_FOUND}
20       return player;
21     },
22     updateAvatar: function(data) {
23       if(!data.id)
24         throw {msg: 'missing id', error: BAD_REQUEST}
25       if(!data.name)
26         throw {msg: 'missing name', error: BAD_REQUEST}
27       var player = world.players[data.id];
28       if(!player)
29         throw {msg: 'not found', error: NOT_FOUND}
30       player.name = data.name;
31       return player;
32     },
33     move: function(data) {
34       if(!data.id)
35         throw {msg: 'missing id', error: BAD_REQUEST}
36       if(!data.to)
```

```
37          throw {msg: 'missing to', error: BAD_REQUEST}
38        var player = world.players[data.id];
39        if(!player)
40          throw {msg: 'not found', error: NOT_FOUND}
41        var to = new model.Position(
42          data.to.x || 0,
43          data.to.y || 0
44        );
45        player.move(to);
46        return player;
47      }
48    };
49
50    module.exports = avatarApi;
51  })(module);
```

**Quellcode A.2: JavaScript basierte Implementierung der API des minimalen Spielmodells.**

```
1  // Import dependencies
2  var express = require('express'), bodyParser = require('body
       -parser'), cors = require('cors');
3
4  // Initiate HTTP-REST module
5  var app = express();
6  app.use(bodyParser.json({}));
7  app.use(cors());
8
9  // Controller
10 var avatarApi = require('./api');
11
12 // MessageHandler
13 var addId = function(id, object) {
14   object.id = id;
15   return object;
16 };
17 var handle = function(cmd, data, req, res) {
18   var result;
19   if(avatarApi[cmd]) {
20     try {
21       result = avatarApi[cmd](data);
```

```
22        if(!result) {
23          res.status(204);
24        }
25      } catch(e) {
26        res.status(e.error || 400);
27        result = e;
28      }
29    } else {
30      res.status(400);
31      result = {msg: 'unkown command', error: 400}
32    }
33    res.send(result);
34 };
35 app.get('/avatars', function(req, res) {
36    handle('getAvatars', {}, req, res);
37 });
38 app.get('/avatars/:id', function(req, res) {
39    handle('getAvatar', {id: req.params.id}, req, res);
40 });
41 app.post('/avatars', function(req, res) {
42    handle('newAvatar', req.body, req, res);
43 });
44 app.put('/avatars/:id', function(req, res) {
45    handle('updateAvatar', addId(req.params.id, req.body), req
       , res);
46 });
47 app.put('/avatars/:id/move', function(req, res) {
48    handle('move', addId(req.params.id, req.body), req, res);
49 });
50
51 // Start server
52 app.listen(3000);
```

**Quellcode A.3: JavaScript basierte Implementierung des eines Controllers eines REST Messages Handlers zur Nutzung der API des minimalen Spielmodells.**

# B. Einfache WebSockt-Server Implementierung mit node.js

```
1  var url = require('url');
2  var server = new (require('ws').Server)({ port: 8080 });
3
4  // Controller
5  var avatarApi = require('./api');
6
7  // MessageHandler
8  server.on('connection', function connection(ws) {
9    var location = url.parse(ws.upgradeReq.url, true);
10   if (location.path == '/avatars') {
11     ws.on('message', function(message) {
12       var msg = JSON.parse(message);
13       var result;
14       if(avatarApi[msg.cmd]) {
15         try {
16           result = avatarApi[msg.cmd](msg.data);
17         } catch(e) {
18           result = e;
19         }
20       } else {
21         result = {msg: 'unkown command', error: 400}
22       }
23       ws.send(JSON.stringify(result));
24     });
25   } else {
26     ws.close();
27   }
```

```
28 });
```

**Quellcode B.1: JavaScript basierte Implementierung eines Controllers auf Basis eines WebSocket Messages Handlers zur Nutzung der API des minimalen Spielmodells aus Anhang A.**

# C. XML Daten Analyse

XML-Datenstrom der durch die Serialisierung der Instanzen aus Abbildung 3.2 entsteht. Im Folgenden das Datenpaket leserlich eingerückt und die Analyse der Binärdaten ohne Leerzeichen.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <testObject smallInt="15" medInt="120700" bigInt="
    10900800700" doubleVal="3.14159265359">
3   <muppetNames>Kermit</muppetNames>
4   <muppetNames>Fozzy</muppetNames>
5   <muppetNames>Piggy</muppetNames>
6   <muppetNames>Animal</muppetNames>
7   <muppetNames>Gonzo</muppetNames>
8   <fruits name="Apple" color="red" inStock="true" price="1.5
    "/>
9   <fruits name="Lemon" color="yellow" inStock="false" price=
    "0.8"/>
10  <fruits name="Apricoat" color="orange" inStock="true"
    price="1.9"/>
11  <fruits name="Kiwi" color="green" inStock="true" price="
    2.3"/>
12 </testObject>
```

```
000: 3c3f 786d 6c20 7665 7273 696f 6e3d 2231  <?xml version="1
010: 2e30 2220 656e 636f 6469 6e67 3d22 5554  .0" encoding="UT
020: 462d 3822 2073 7461 6e64 616c 6f6e 653d  F-8" standalone=
030: 2279 6573 223f 3e3c 7465 7374 4f62 6a65  "yes"?><testObje
040: 6374 2073 6d61 6c6c 496e 743d 2231 3522  ct smallInt="15"
050: 206d 6564 496e 743d 2231 3230 3730 3022   medInt="120700"
060: 2062 6967 496e 743d 2231 3039 3030 3830   bigInt="1090080
070: 3037 3030 2220 646f 7562 6c65 5661 6c3d  0700" doubleVal=
080: 2233 2e31 3431 3539 3236 3533 3539 223e  "3.14159265359">
090: 3c6d 7570 7065 744e 616d 6573 3e4b 6572  <muppetNames>Ker
0a0: 6d69 743c 2f6d 7570 7065 744e 616d 6573  mit</muppetNames
0b0: 3e3c 6d75 7070 6574 4e61 6d65 733e 466f  ><muppetNames>Fo
0c0: 7a7a 793c 2f6d 7570 7065 744e 616d 6573  zzy</muppetNames
0d0: 3e3c 6d75 7070 6574 4e61 6d65 733e 5069  ><muppetNames>Pi
0e0: 6767 793c 2f6d 7570 7065 744e 616d 6573  ggy</muppetNames
0f0: 3e3c 6d75 7070 6574 4e61 6d65 733e 416e  ><muppetNames>An
100: 696d 616c 3c2f 6d75 7070 6574 4e61 6d65  imal</muppetName
110: 733e 3c6d 7570 7065 744e 616d 6573 3e47  s><muppetNames>G
120: 6f6e 7a6f 3c2f 6d75 7070 6574 4e61 6d65  onzo</muppetName
130: 733e 3c66 7275 6974 7320 6e61 6d65 3d22  s><fruits name="
140: 4170 706c 6522 2063 6f6c 6f72 3d22 7265  Apple" color="re
150: 6422 2069 6e53 746f 636b 3d22 7472 7565  d" inStock="true
160: 2220 7072 6963 653d 2231 2e35 222f 3e3c  " price="1.5"/><
170: 6672 7569 7473 206e 616d 653d 224c 656d  fruits name="Lem
180: 6f6e 2220 636f 6c6f 723d 2279 656c 6c6f  on" color="yello
190: 7722 2069 6e53 746f 636b 3d22 6661 6c73  w" inStock="fals
1a0: 6522 2070 7269 6365 3d22 302e 3822 2f3e  e" price="0.8"/>
1b0: 3c66 7275 6974 7320 6e61 6d65 3d22 4170  <fruits name="Ap
1c0: 7269 636f 6174 2220 636f 6c6f 723d 226f  ricoat" color="o
1d0: 7261 6e67 6522 2069 6e53 746f 636b 3d22  range" inStock="
1e0: 7472 7565 2220 7072 6963 653d 2231 2e39  true" price="1.9
1f0: 222f 3e3c 6672 7569 7473 206e 616d 653d  "/><fruits name=
200: 224b 6977 6922 2063 6f6c 6f72 3d22 6772  "Kiwi" color="gr
210: 6565 6e22 2069 6e53 746f 636b 3d22 7472  een" inStock="tr
220: 7565 2220 7072 6963 653d 2232 2e33 222f  ue" price="2.3"/
230: 3e3c 2f74 6573 744f 626a 6563 743e       ></testObject>
```

# D. JSON Daten Analyse

JSON-Datenstrom der durch die Serialisierung der Instanzen aus Abbildung 3.2 entsteht. Im Folgenden das Datenpaket leserlich eingerückt und die Analyse der Binärdaten ohne Leerzeichen.

```
{ "smallInt": 15,
  "doubleVal": 3.14159265359,
  "muppetNames": ["Kermit", "Fozzy", "Piggy", "Animal", "
   Gonzo"],
  "fruits": [{
    "color": "red",
    "price": 1.5,
    "name": "Apple",
    "inStock": true
  }, {
    "color": "yellow",
    "price": 0.8,
    "name": "Lemon",
    "inStock": false
  }, {
    "color": "orange",
    "price": 1.9,
    "name": "Apricoat",
    "inStock": true
  }, {
    "color": "green",
    "price": 2.3,
    "name": "Kiwi",
    "inStock": true
  }],
  "medInt": 120700,
  "bigInt": 10900800700}
```

```
000: 7b22 736d 616c 6c49 6e74 223a 3135 2c22  {"smallInt":15,"
010: 646f 7562 6c65 5661 6c22 3a33 2e31 3431  doubleVal":3.141
020: 3539 3236 3533 3539 2c22 6d75 7070 6574  59265359,"muppet
030: 4e61 6d65 7322 3a5b 224b 6572 6d69 7422  Names":["Kermit"
040: 2c22 466f 7a7a 7922 2c22 5069 6767 7922  ,"Fozzy","Piggy"
050: 2c22 416e 696d 616c 222c 2247 6f6e 7a6f  ,"Animal","Gonzo
060: 225d 2c22 6672 7569 7473 223a 5b7b 2263  "],"fruits":[{"c
070: 6f6c 6f72 223a 2272 6564 222c 2270 7269  olor":"red","pri
080: 6365 223a 312e 352c 226e 616d 6522 3a22  ce":1.5,"name":"
090: 4170 706c 6522 2c22 696e 5374 6f63 6b22  Apple","inStock"
0a0: 3a74 7275 657d 2c7b 2263 6f6c 6f72 223a  :true},{"color":
0b0: 2279 656c 6c6f 7722 2c22 7072 6963 6522  "yellow","price"
0c0: 3a30 2e38 2c22 6e61 6d65 223a 224c 656d  :0.8,"name":"Lem
0d0: 6f6e 222c 2269 6e53 746f 636b 223a 6661  on","inStock":fa
0e0: 6c73 657d 2c7b 2263 6f6c 6f72 223a 226f  lse},{"color":"o
0f0: 7261 6e67 6522 2c22 7072 6963 6522 3a31  range","price":1
100: 2e39 2c22 6e61 6d65 223a 2241 7072 6963  .9,"name":"Apric
110: 6f61 7422 2c22 696e 5374 6f63 6b22 3a74  oat","inStock":t
120: 7275 657d 2c7b 2263 6f6c 6f72 223a 2267  rue},{"color":"g
130: 7265 656e 222c 2270 7269 6365 223a 322e  reen","price":2.
140: 332c 226e 616d 6522 3a22 4b69 7769 222c  3,"name":"Kiwi",
150: 2269 6e53 746f 636b 223a 7472 7565 7d5d  "inStock":true}]
160: 2c22 6d65 6449 6e74 223a 3132 3037 3030  ,"medInt":120700
170: 2c22 6269 6749 6e74 223a 3130 3930 3038  ,"bigInt":109008
180: 3030 3730 307d                           00700}
```

# E. Java ObjectStream Daten Analyse

Java Objekt-Datenstrom der durch die Serialisierung der Instanzen aus Abbildung 3.2 entsteht. Markiert sind die Werte der Attribute.

```
000: aced 0005 7372 0018 6465 2e66 7375 2e73  ....sr..de.fsu.s
010: 7472 6561 6d2e 5465 7374 4f62 6a65 6374  tream.TestObject
020: c56c 4596 9516 f20c 0200 064a 0006 6269  .lE........J..bi
030: 6749 6e74 4400 0964 6f75 626c 6556 616c  gIntD..doubleVal
040: 4900 066d 6564 496e 7453 0008 736d 616c  I..medIntS..smal
050: 6c49 6e74 5b00 0666 7275 6974 7374 0016  lInt[..fruitst..
060: 5b4c 6465 2f66 7375 2f73 7472 6561 6d2f  [Lde/fsu/stream/
070: 4672 7569 743b 5b00 0b6d 7570 7065 744e  Fruit;[..muppetN
080: 616d 6573 7400 135b 4c6a 6176 612f 6c61  amest..[Ljava/la
090: 6e67 2f53 7472 696e 673b 7870 0000 0002  ng/String;xp....
0a0: 89bd 04bc 4009 21fb 5444 2eea 0001 d77c  ....@.!.TD.....|
0b0: 000f 7572 0016 5b4c 6465 2e66 7375 2e73  ..ur..[Lde.fsu.s
0c0: 7472 6561 6d2e 4672 7569 743b fdef f2c7  tream.Fruit;....
0d0: 36c1 f13d 0200 0078 7000 0000 0473 7200  6..=...xp....sr.
0e0: 1364 652e 6673 752e 7374 7265 616d 2e46  .de.fsu.stream.F
0f0: 7275 6974 320f e926 03d2 9d7c 0200 045a  ruit2..&...|...Z
100: 0007 696e 5374 6f63 6b46 0005 7072 6963  ..inStockF..pric
110: 654c 0005 636f 6c6f 7274 0012 4c6a 6176  eL..colort..Ljav
120: 612f 6c61 6e67 2f53 7472 696e 673b 4c00  a/lang/String;L.
130: 046e 616d 6571 007e 0007 7870 013f c000  .nameq.~..xp.?..
140: 0074 0003 7265 6474 0005 4170 706c 6573  .t..redt..Apples
150: 7100 7e00 0600 3f4c cccd 7400 0679 656c  q.~...?L..t..yel
160: 6c6f 7774 0005 4c65 6d6f 6e73 7100 7e00  lowt..Lemonsq.~.
170: 0601 3ff3 3333 7400 066f 7261 6e67 6574  ..?.33t..oranget
180: 0008 4170 7269 636f 6174 7371 007e 0006  ..Apricoatsq.~..
190: 0140 1333 3374 0005 6772 6565 6e74 0004  .@.33t..greent..
1a0: 4b69 7769 7572 0013 5b4c 6a61 7661 2e6c  Kiwiur..[Ljava.l
1b0: 616e 672e 5374 7269 6e67 3bad d256 e7e9  ang.String;..V..
1c0: 1d7b 4702 0000 7870 0000 0005 7400 064b  .{G...xp....t..K
1d0: 6572 6d69 7474 0005 466f 7a7a 7974 0005  ermitt..Fozzyt..
1e0: 5069 6767 7974 0006 416e 696d 616c 7400  Piggyt..Animalt.
1f0: 0547 6f6e 7a6f                           .Gonzo
```

# F. Protocol Buffer Daten Analyse

Protocol Buffer Objekt-Datenstrom der durch die Serialisierung der Instanzen aus Abbildung 3.2 entsteht. Markiert sind die Werte der Attribute. Anschließend die Beschreibung der Nachrichten.

```
000: 9801 08fc ae07 100f 18bc 89f4 cd28 21ea  ...........(!.
010: 2e44 54fb 2109 402a 130a 0372 6564 1205  .DT.!.@*...red..
020: 4170 706c 651d 0000 c03f 2001 2a16 0a06  Apple....? .*...
030: 7965 6c6c 6f77 1205 4c65 6d6f 6e1d cdcc  yellow..Lemon...
040: 4c3f 2000 2a19 0a06 6f72 616e 6765 1208  L? .*...orange..
050: 4170 7269 636f 6174 1d33 33f3 3f20 012a  Apricoat.33.? .*
060: 140a 0567 7265 656e 1204 4b69 7769 1d33  ...green..Kiwi.3
070: 3313 4020 0132 064b 6572 6d69 7432 0546  3.@ .2.Kermit2.F
080: 6f7a 7a79 3205 5069 6767 7932 0641 6e69  ozzy2.Piggy2.Ani
090: 6d61 6c32 0547 6f6e 7a6f                 mal2.Gonzo
```

```
message TestObject {
  required int32 medInt = 1;
  required int32 smallInt = 2;
  required int64 bigInt = 3;
  required double doubleVal = 4;

  message Fruit {
    required string color = 1;
    required string name = 2;
    required float price = 3;
    required bool inStock = 4;
  }

  repeated Fruit fruits = 5;
  repeated string muppetNames = 6;
}
```

**Quellcode F.1: Beschreibung der Nachricht.**

# G. SmartFoxServer Daten Analyse

SmartFoxServer-Datenstrom der durch die Serialisierung der Instanzen aus Abbildung 3.2 entsteht. Im Folgenden das Datenpaket für die Analyse der Binärdaten.

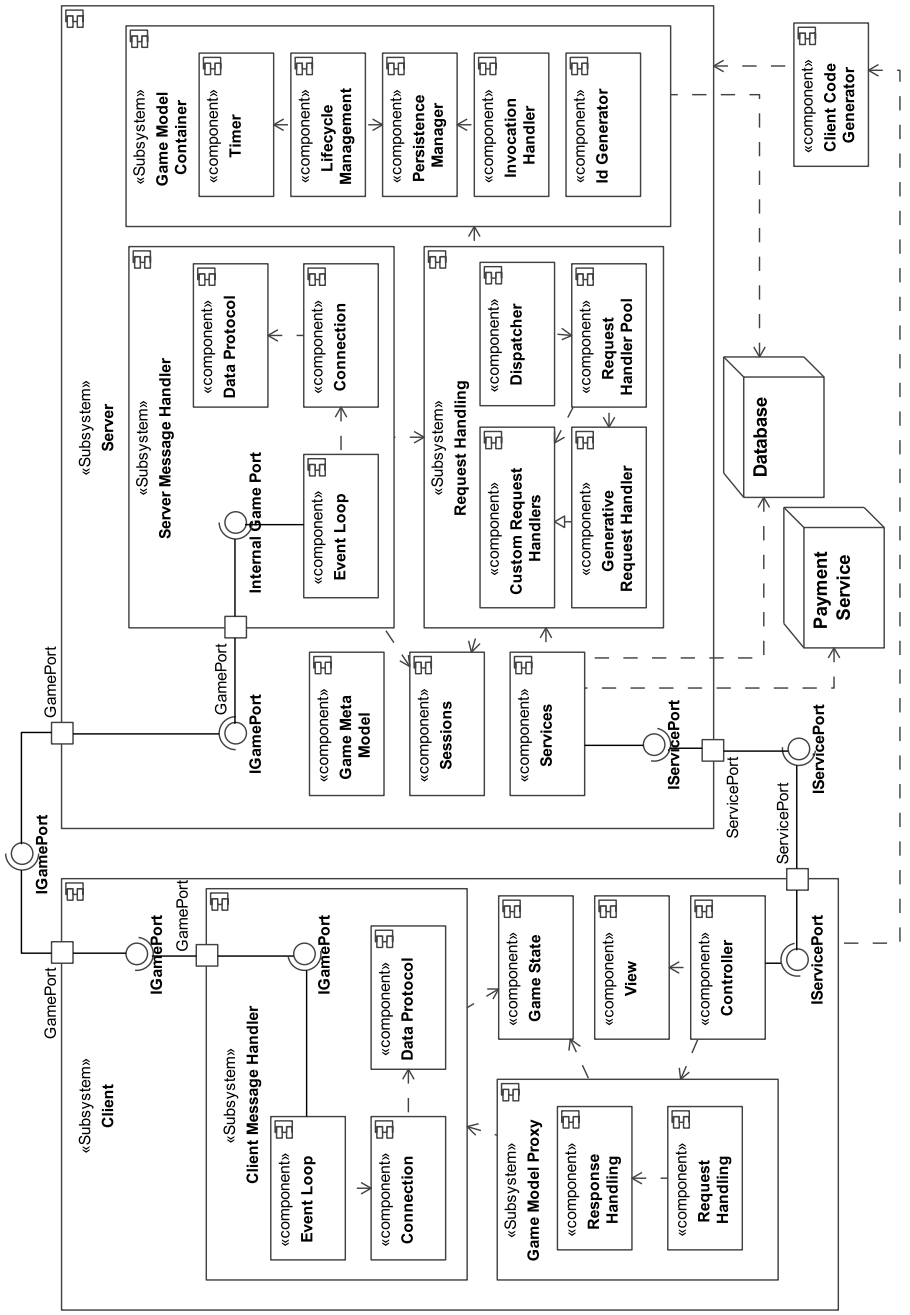
```
000: 1200 0600 0964 6f75 626c 6556 616c 0740  .....doubleVal.@
010: 0921 fb54 442d 1800 0666 7275 6974 7311  .!.TD-...fruits.
020: 0004 1200 0400 046e 616d 6508 0005 4170  .......name...Ap
030: 706c 6500 0563 6f6c 6f72 0800 0372 6564  ple..color...red
040: 0007 696e 5374 6f63 6b01 0100 0570 7269  ..inStock.....pri
050: 6365 063f c000 0012 0004 0004 6e61 6d65  ce.?..........name
060: 0800 054c 656d 6f6e 0005 636f 6c6f 7208  ...Lemon..color.
070: 0006 7965 6c6c 6f77 0007 696e 5374 6f63  ..yellow..inStoc
080: 6b01 0000 0570 7269 6365 063f 4ccc cd12  k....price.?L...
090: 0004 0004 6e61 6d65 0800 0841 7072 6963  ....name...Apric
0a0: 6f61 7400 0563 6f6c 6f72 0800 066f 7261  oat..color...ora
0b0: 6e67 6500 0769 6e53 746f 636b 0101 0005  nge..inStock....
0c0: 7072 6963 6506 3ff3 3333 1200 0400 046e  price.?.33.....n
0d0: 616d 6508 0004 4b69 7769 0005 636f 6c6f  ame...Kiwi..colo
0e0: 7208 0005 6772 6565 6e00 0769 6e53 746f  r...green..inSto
0f0: 636b 0101 0005 7072 6963 6506 4013 3333  ck....price.@.33
100: 0008 736d 616c 6c49 6e74 020f 0006 6d65  ..smallInt....me
110: 6449 6e74 0400 01d7 7c00 0b6d 7570 7065  dInt....|..muppe
120: 744e 616d 6573 1000 0500 064b 6572 6d69  tNames.....Kermi
130: 7400 0546 6f7a 7a79 0005 5069 6767 7900  t..Fozzy..Piggy.
140: 0641 6e69 6d61 6c00 0547 6f6e 7a6f 0006  .Animal..Gonzo..
150: 6269 6749 6e74 0500 0000 0289 bd04 bc    bigInt.........
```

# H. Abbildungsformat Daten Analyse

Datenstrom des Abbildungsformates der Ausführungsumgebung der durch die Serialisierung der Instanzen aus Abbildung 3.2 entsteht. Markiert sind die Nutzdaten.

```
000: 8181 8081 4204 4de8 25e0 0000 4009 21fb  ....B.M.%...@.!.
010: 5444 2eea 8084 8182 8082 0003 7265 6401  TD..........red.
020: 0005 4170 706c 653f c000 0081 8280 8200  ..Apple?........
030: 0679 656c 6c6f 7700 0005 4c65 6d6f 6e3f  .yellow...Lemon?
040: 4ccc cd81 8280 8200 066f 7261 6e67 6501  L........orange.
050: 0008 4170 7269 636f 6174 3ff3 3333 8182  ..Apricoat?.33..
060: 8082 0005 6772 6565 6e01 0004 4b69 7769  ....green...Kiwi
070: 4013 3333 3200 01d7 7c80 8500 064b 6572  @.332...|....Ker
080: 6d69 7400 0546 6f7a 7a79 0005 5069 6767  mit..Fozzy..Pigg
090: 7900 0641 6e69 6d61 6c00 0547 6f6e 7a6f  y..Animal..Gonzo
0a0: 000f                                     ..
```

# I. Komponentenübersicht



# J. Quellcodebeispiele zur Ausführungsumgebung

```
@Entity
@NamedQuery(
  name = "grow_update",
  query = "UPDATE SomeTree t SET t.size = :size WHERE t.id =
    :id"
)
@Persist(type = PersistType.BY_USING_PERSIST_ACTION)
public class SomeTree extends AbstractGameObject<TreeWorld>
   {
  // ...
  private int size;
  // ...
  @PersistAction("grow_update")
  public void grow() {
    size ++;
  }
  // ...
}
```

**Quellcode J.1: Beispiel für Persistenzkonfiguration in einer einfachen Spielentität.**

```
  @SocketObject(id = 255, namespace = 1)
  public class SomeObject {
    private int x = 2;
    private int y = 511;
  }
```

**Quellcode J.2: Kommunikationsobjekt mit zwei ganzzahligen Attributen, auf Getter und Setter wurde verzichtet**

```
1  @SocketObject(id = 1, namespace = 1)
2  public class SomeArrayObject {
3    private int[] list = new int[2, 511];
4  }
```

**Quellcode J.3: Kommunikationsobjekt mit einer Reihung mit ganzzahligem Typ. Auf Getter und Setter wurde verzichtet.**

```
1  @SocketObject(id = 2)
2  public class SomeParent {
3    private SomeObject list = new SomeObject(2, 511);
4  }
```

**Quellcode J.4: Kommunikationsobjekt mit einem komplexen Datentyp. Auf Getter und Setter wurde verzichtet.**

```
1  @SocketObject(id = 3)
2  public class Something {
3    private Object something = new SomeObject(2, 511);
4  }
```

**Quellcode J.5: Kommunikationsobjekt mit einem variablem Datentyp. Auf Getter und Setter wurde verzichtet.**

```
1 @SocketObject
2 public class Player {
3   private int x;
4   private String name;
5   private int id;
6   public int getX() {
7     return this.x;
8   }
9   public String getName() {
10    return this.name;
11  }
12  @SocketObjectIdentifier
13  public int getId() {
```

```
14      return this.id;
15    }
16    public void setX(int value) {
17      this.x = value;
18    }
19    public void setName(String value) {
20      this.name = value;
21    }
22    public void setId(int value) {
23      this.id = value;
24    }
25  }
```

**Quellcode J.6: Beispiel Entität eines Spielmodells, welche instanziert versendet werden kann.**

```
1  @SocketObject
2  public class XUpdate {
3    private int x;
4    private int id;
5    public int getX() {
6      return this.x;
7    }
8    @SocketObjectIdentifier(target=Player.class)
9    public int getId() {
10     return this.id;
11   }
12   public void setX(int value) {
13     this.x = value;
14   }
15   public void setId(int value) {
16     this.id = value;
17   }
18 }
```

**Quellcode J.7: Beispiel Kommunikationsobjekt, welches das Attribut *x* aus der Entität *Player* vom Quellcode J.6 aktualisiert.**

```
1  @SocketObject
2  public class Player {
```

```
3   // ...
4   private int id;
5   @SocketObjectIdentifier
6   public int getId() {
7     return this.id;
8   }
9   public void setId(int value) {
10    this.id = value;
11  }
12  @SocketObjectRemoteMethod
13  public Player dance() {
14    // ...
15  }
16 }
```

**Quellcode J.8: Erweiterte Entität *Player* zur Bereitstellung der Methode *dance* für das Beispiel in Quellcode 6.4.**

# K. Ideenskizze Terra-Life-Evolution

Inhaltlich übernommene Ideenskizze des Studentenprojekts Terra-Life-Evolution. Nummerierung der Abbildungen wurde angepasst. Der Inhalt dieser Ideenskizze wurde erstellt von Oleksandr G., Christoph K., Paul M. und Felix B.

## *Übersicht*

*Terra Life Evolution ist ein klassisches MMORPG, welches in einem Webbrowser spielbar ist. Es erfindet das Rad nicht neu, jedoch macht es das Rad etwas runder. Ziel des Spiels ist es, seinen Avatar durch verschiedene Level der Spielwelt zu führen um Aufgaben zu erfüllen, wofür es Erfahrungspunkte gibt.*

## *Look and Feel*

*Das Spielfeld besteht aus Feldern, so wie es in Smallquest der Fall ist. Es wird also ein kartesisches Koordinatensystem genutzt, auf dem sich der Spieler um einen bestimmten Wert in x- und y-Richtung bewegen kann. Die Grafik wird in einer Pixelgrafik gehalten. Der Avatar des Spielers steht dabei immer im Zentrum des angezeigten Bereichs. Bewegt er sich, so wird die Welt um ihn herum verschoben. Diese Welt ist sehr viel größer als der angezeigte Bereich und wird durch natürliche Hindernisse begrenzt.*

# *Features*

***Integral**: Skill-System*
***Chrome**: 2 Rassen (Mensch und Mutant), unterschiedliche Level/Gebiete, cross-play trotzdem möglich.*

*Das Haupt-Feature wodurch sich TLE von anderen Spielen absetzt, ist die absolut variable Gestaltung der Fähigkeiten seines Avatars. Spiele dieses Genres stellen den Spieler zu Beginn vor die Wahl einer Klasse seines Charakters. TLE macht dies gerade nicht. Der Spieler muss nur die Zugehörigkeit zu einer Rasse wählen. In welche Richtung sich sein Charakter entwickelt bestimmt er selbst. So ist es durchaus möglich durch Vergabe der Erfahrungspunkte einen klassischen Krieger zu erstellen. Doch muss man diesen Weg nicht gehen. Verteilt man seine Punkte anders, so können auch Mischformen entstehen mit ihren eigenen Vor- und Nachteilen.*

*Zur individuellen Gestaltung seines Charakters bietet TLE einen Pool aus Erfahrungspunkten, welche individuell auf drei verschiedene Skill-Bäume (Waffen, Rüstung, Charakter) verteilt werden können.*

*Für einen anhaltenden Spielspaß sorgt die Möglichkeit, jederzeit mit anderen Spielern, sich als eine Gruppe in der Spielwelt zu bewegen, um die gestellten Aufgaben zu bewältigen. Dabei macht es keinen Unterschied, von welcher Rasse die Mitspieler sind. Prinzipiell ist es jedem Spieler erlaubt die Gebiete der anderen Rasse zu betreten.*

## *Spielwelt*

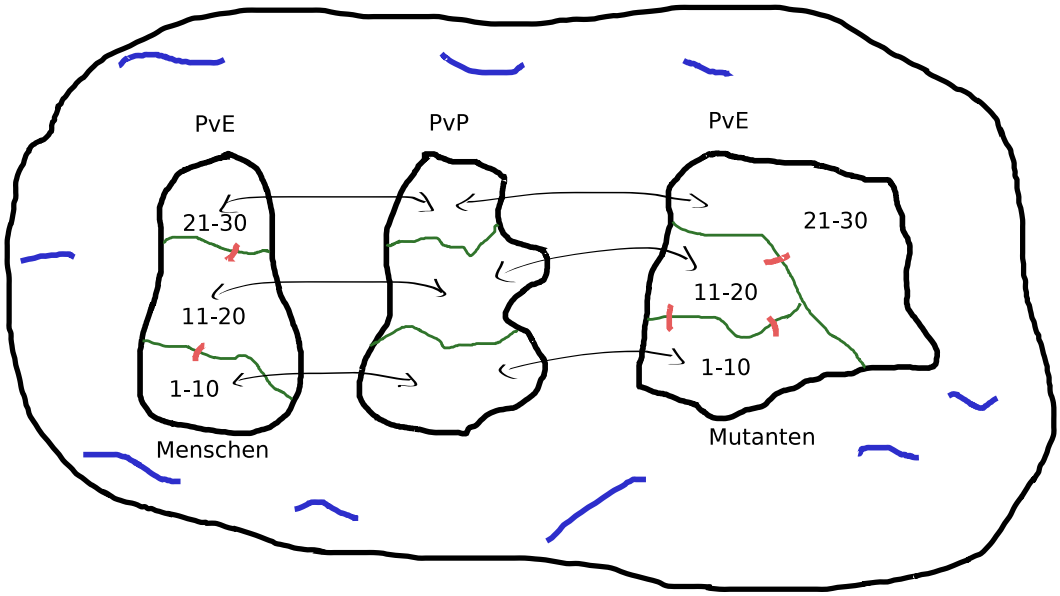



**Abbildung K.1: Übersicht Spielewelt**

*Wie die Abbildung K.1 zeigt, gibt es 3 unterschiedliche Hauptinseln, 2 Player versus Environment (PvE) und eine Player versus Player (PvP) Insel. Diese Inseln sind in Spielerlevel unterteilt. Es gibt Zonen für die Spielerlevels 1 bis 10, 11 bis 20 und 21 bis 30. Auf den PvE Inseln bekämpft man Monster, die dem Spieler Erfahrungspunkte geben. Im PvP kann man seine Kampfkünste gegen andere Spieler testen.*

## *Regeln*

*Das Spiel ist wie ein klassisches MMORPG aufgebaut. Es gibt Quests, welche die Story näher bringen sollen. Als Belohnung für abgeschlossene Quests, bekommt man Erfahrung, Gold und neu Quests (sofern es noch weitere gibt). Dadurch wird dem Spieler die Möglichkeit geboten, schneller als nur durch Kämpfe aufzusteigen und neue Ausrüstung zu kaufen.*

*Ziel des Spiel ist es, alle Quests erfolgreich zu absolvieren. Hat ein Spieler dies geschafft, so erhält er das neu Ziel, der beste zu sein. Dies bedeutet, die beste Ausrüstung zu erhalten und zusammen mit der Skillung und des Geschicks*

*des Spielers, der beste zu werden. Der Spieler soll also folglich in dem PvP-Level der stärkste werden.*

*Die Herausforderung des Spielers liegt darin, die Skillung, für die er sich entscheidet, zu meistern und die perfekte Ausrüstung für diese Skillung zu finden. Hat ein Spieler dies erreicht, wird es noch andere Spieler geben, die dies schaffen. Ein Langzeitanreiz wird es sein, seine ganz persönliche Skillung mit anderen Spielern im PvP-Bereich zu messen.*

# *Oberfläche*

## *GUI*

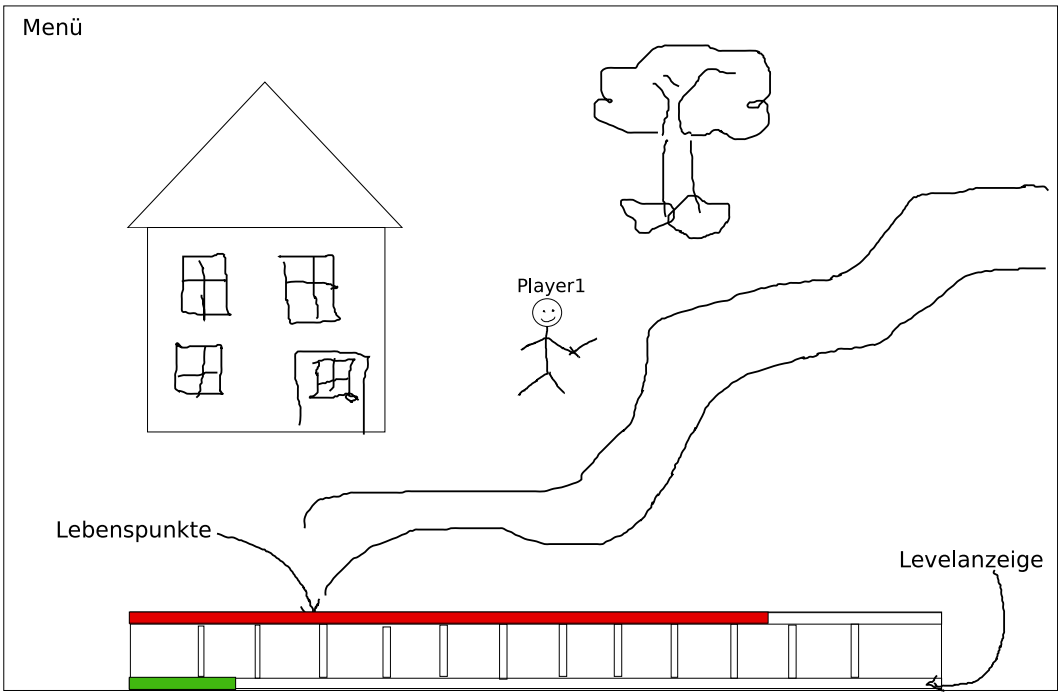



**Abbildung K.2: Skizze GUI**

## *Skills*

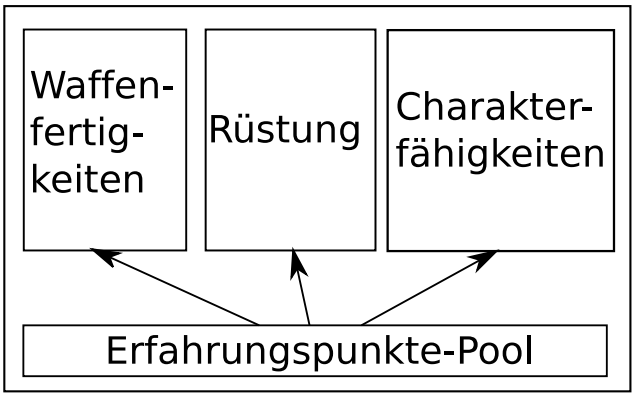



**Abbildung K.3: Skizze Skillbaum**

## Interaktion

*Durch einen einfachen Mausklick oder eventuell durch die WASD-Tasten kann sich der Spieler über das zweidimensionale Raster bewegen. Der Spieler bewegt sich dann zu dem entsprechendem Feld. Ein Kampf kann über 2 Wege initiert werden:*

1. *Gewisse Gegner haben ein Sichtfeld, das bestimmte Felder vor diesen abdeckt und überprüft, ob ein Spieler auf diesen ist. Befindet sich dort ein Spieler, rennt das Monster zu dem Spieler auf kürzestem Wege und greift diesen an.*

2. *Der Spieler klickt auf den Gegner(dies entspricht dem Standartangriff) oder steht vor dem Gegner und macht einen Angriff.*

*Der Kampf wird dann durch Skills und Bewegungen des Spielers beeinflusst.*

# L. Ideenskizze Hack'N'Slay

Inhaltlich übernommene Ideenskizze des Studentenprojekts Kokosnussbikini. Nummerierung der Abbildungen wurde angepasst. Der Inhalt dieser Ideenskizze wurde erstellt von Silvio H., Julien K. und Markus K.

## *Übersicht*

*Unser Projekt ist ein 2D Hack and Slay MMO-Browsergame. Alle Spieler spielen auf einer fiktiven Karte. Das Ziel ist es den Baum des Lebens zu beschützen. Es werden die Spieler gegen Wellen von Zombies und einen Zombieboss spielen.*

## *Features*

### *Integral Features*

*Als Integral Features besitzt das Spiel mehrere Charakterklassen, die sich gegenseitig durch jeweils eine Spezial- und Basisfähigkeit ergänzen.*

### *Chrome Features*

*Ein Chrome Feature werden die Kisten sein. Diese sind mit nützlichen Bonies, aber auch mit negativen Folgen gefüllt. Beispielsweise:*

- *Respawn aller Toten,*
- *10% mehr Schaden für die Spieler,*
- *ein schwächerer Zombieboss,*

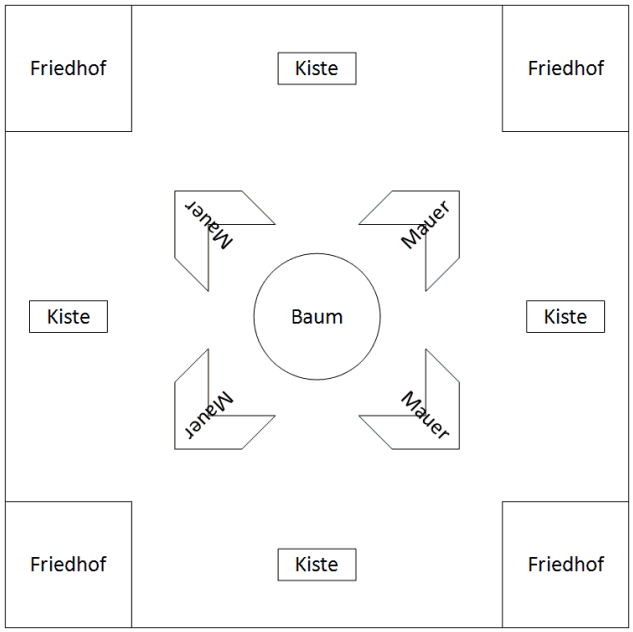



**Abbildung L.1: Karte des Spieles**

- *10% mehr Leben für die Spieler,*
- *der Zombieboss kommt früher,*
- *der Baum wird geheilt,*
- *leben des Baumes erhöht sich,*
- *alle Fähigkeiten werden zurückgesetzt.*

*Ein weiteres Chrome Feature ist die Grundheilung der Spieler.*

# *Interface*

*Das Spiel wird mittels der Pfeiltasten, der rechten Shift-Taste und der linken Maustaste gespielt. Mit den Pfeiltasten kann sicher der Spieler nach dem Manhattan-Prinzip bewegen. Mit der Shift-Taste kann der Spieler zwischen seiner Basis und Spezialfähigkeit, seines Charakters, wechseln. Mittels der linken Maustaste werden Gegner markiert. Der Spieler sieht während des Spieles nur*

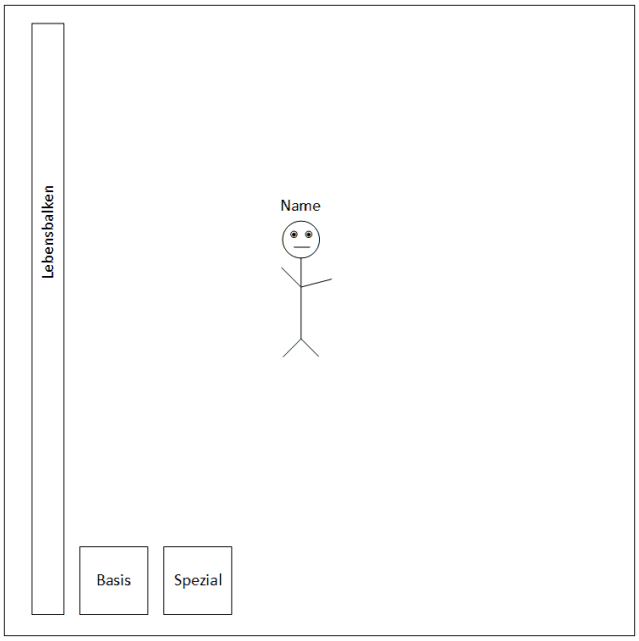



**Abbildung L.2: Spieleransicht**

*einen Teil der Karte. Weiterhin sieht er seinen Lebensbalken und seine Fähigkeiten, von der stets eine aktiviert ist. In seiner Teilansicht (den Teil der Karte in dem er sich befindet) ist er stets zentriert.*

## *Gameplay*

*Es gibt im Spiel drei Charaktere. Diese sind ein Heiler, ein Ritter und ein Jäger. Jeder der Charaktere hat eine Basisfähigkeit und eine Spezialfähigkeit. Die Basisfähigkeit wird jede Sekunde, die Spezialfähigkeit kann nur alle fünf Sekunden aktiviert werden. Die Aktivierung erfolgt automatisch.*

*Der Heiler hat die Basisfähigkeit heilen und die Spezialfähigkeit einfrieren. Die Fähigkeiten werden in seinem Teilfenster ausgeführt. Er hat nur wenig Lebenspunkte und kann leicht verletzt werden.*

*Der Ritter hat die Basisfähigkeit schlagen und als Spezialfähigkeit ein Schild. Er macht wenig Schaden, hat dafür aber viele Lebenspunkte. Der Schlag wird jede halbe Sekunde auf den Gegner ausgeführt, den er durch einen Mausklick markiert hat. Das Schild wirkt für zehn Sekunden.*

*Der Jäger hat als Basisfähigkeiten das schießen eines Pfeiles und als Spezialfähigkeit einen Flächenschaden. Der Pfeil immer auf den Markierten Gegner*

*geschickt. Der Pfeil fliegt jede Sekunde. Er besitzt wenig Lebenspunkte, erzeugt dafür viel Schaden. Der Flächenschaden kann jede fünf Sekunden eingesetzt werden.*

*Die Folgende Tabelle fasst die Fähigkeiten der Charakterklassen und deren Rüstung zusammen.*

*Die Monster werden mit fortlaufendem Spiel stärker und treten in größeren Mengen auf. Die Grundheilung erhöht jede zehn Sekunden die Gesundheit des Spielers um 5%. Weiter werden die Bonuskisten zufällig gefüllt und können pro Runde nur einmal zu öffnen. Sie öffnen sich automatisch wenn man davor steht.*

## *Regeln*

- *Vor der ersten Welle sind fünf Minuten Wartezeit.*
- *Spieler die Sterben oder dem Spiel beitreten werden bei Beginn der nächsten Runde gespawnt.*
- *Die Anzahl der Monster hängt von der Anzahl der Spieler ab.*

## *Level Design*

*Das Spiel ist immer nur auf der gesamten Karte zu spielen. Die Spieler haben dabei die Aufgabe den Baum zu beschützten. Die Spieler spawnen in einem bestimmten Radius um den Baum. Es müssen nun mehrere Wellen überstanden werden. Nach einer bestimmten Anzahl von Wellen, in der letzten, erscheint ein Endboss. Dieser kann erst besiegt werden wenn alle Friedhöfe zerstört sind. Die Friedhöfe sind erst in der letzten Runde des Spiels verwundbar. Wenn der Endboss besiegt ist, ist das Spiel gewonnen. Wenn der Baum zerstört ist es verloren.*

*Nach einem verlorerem oder gewonnenem Spiel beginnt das Spiel von vorne.*

# M. Ideenskizze Kokosnussbikini

Inhaltlich übernommene Ideenskizze des Studentenprojekts Kokosnussbikini. Der Inhalt dieser Ideenskizze wurde erstellt von Sebastian K., Panajotis M., Theresa P. und Marcel D.

## *Übersicht*

- *2D Comic Mittelalter MMORPG im Browser*
- *Player vs Enemy (evtl. später PvP Arena)*
- *fiktive Welt mit Städten, Wäldern, Dungeons und Quests*

### *Ziele*

- *Entwicklung eines individuellen Charakters mit seltenen Gegenständen, Rüstungen und Skills*
- *entdecken der Spielwelt*
- *Crafting / Housing*

## *Features*

### *Integral Features*

- *Charakterverbesserung durch Waffen, Rüstungen und Skills (keine Charakter-Level)*

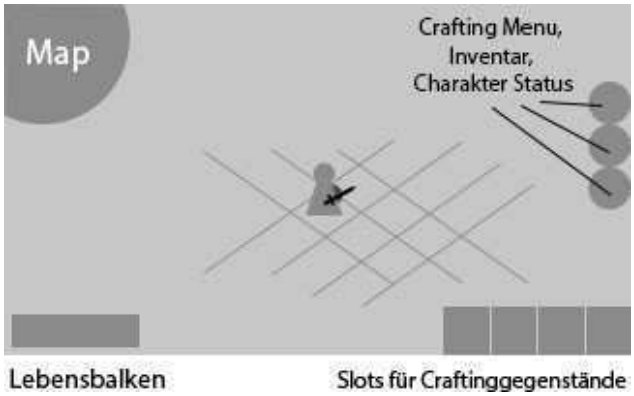
- *Spieler kann sich frei in der Spielwelt bewegen*
- *Gebiete für stärkere und schwächere Spieler*
- *Skill – Level – System „learning by doing“*
- *Crafting von Gegenständen und Gebäuden abhängig vom Skill (z.B. Zelt -> Haus -> Burg)*
- *Zusammenarbeit von Spielern aus unterschiedlichen Skillrichtungen (Holzfäller, Steinmetz,...)*
- *Handel zwischen den Spielern (auf Marktplatz innerhalb einer Stadt)*
- *Rundenbasierte Kämpfe – Einsatz von Waffen, Schilden, Tränken, Fähigkeiten, Kampffertigkeiten und Zaubern*

### Chrome Features

- *Housing – Spieler kann eigenes Zuhause bauen und dies individuell mit gefundenen und gecrafteten Gegenständen gestalten*
- *Kleidung für Charaktere, die ausschließlich dem Aussehen dienen*

## *Interface*

- *isometrische Ansicht*
- *Spieler ist immer zentriert und bewegt sich frei, seine Position wird aber immer einem Feld zugeorndet*




## *Gameplay*

- *der Spieler bewegt sich frei in der Spielwelt und kann dort verborgene Dungeons, Städte oder Schätze finden*
- *es gibt verschiedene Arten von Monstern, die der Spieler bekämpfen kann, um seinen Skill zu steigern oder seltene Gegenstände zu finden, dabei kann der Spieler auf verschiedene Waffenklassen zurückgreifen und sich dadurch spezialisieren*
- *Tiere können gejagt werden, um dadurch entsprechend der Skillstufe verschiedene Craftinggegenstände zu erhalten*
- *Pflanzen ernten / Bäume fällen / Steine und Erze abbauen (Skillverbesserung und Erhalt von Craftinggegenständen)*
- *stirbt ein Spieler, spawnt er entweder in der Anfangsstadt oder seinem selbstgebauten Zuhause (Haus / Zelt) – Malus?*

## *Regeln*

- *der Spieler beginnt an einer zufälligen Stelle in einem vordefinierten Anfangsgebiet (Zentrum der Spielwelt) ohne Waffen und Rüstungen*
- *dem Spieler werden erste Hinweise bezüglich des Craftingsystems gegeben, wodurch er sich eine erste Waffe bauen und kleine Monster bekämpfen kann*

- *das erste Ziel des Spiels wird es sein, die Anfangsstadt zu finden*
- *danach kann der Spieler machen, was er will (Quests / Crafting / Monster)*

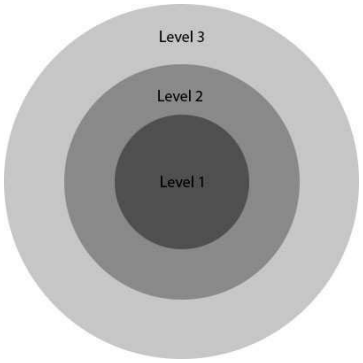
### *Abbauen*

- *befindet sich der Spieler neben einem Rohstoff (Baum, Stein,...) und besitzt das entsprechende Abbauwerkzeug, kann dieser durch einen links Klick auf den Rohstoff und der Auswahl des Befehls „abbauen“ abgebaut werden*
- *nach einer kurzen Wartezeit, welche mit Hilfe eines Balkens angezeigt wird, wird der Rohstoff abgebaut und befindet sich danach im Inventar des Spielers*

### *Crafting*

- *im Crafting Menu werden dem Spieler die entsprechenden Gegenstände und Gebäude angezeigt die er unter Berücksichtigung seines Skills bauen kann*
- *besitzt er alle benötigten Roshtoffe und Werkzeuge, kann er den Gegenstand oder das Gebäude craften, nach einer gewissen Wartezeit befindet es sich im Inventar*

# *Level Design*




- *kreisförmiger Levelaufbau*
- *im Zentrum liegt das Anfangsgbiet*
- *je weiter nach außen man kommt, umso stärker werden die Gegner*
- *seltene Gegenstände und Baumaterialien in höheren Levels*

# N. Ideenskizze MMO-Meischta

Inhaltlich übernommene Ideenskizze des Studentenprojekts MMO-Meischta. Der Inhalt dieser Ideenskizze wurde erstellt von Christoph T., Sebastian G., Christopher M. und Tien D. L.

## *Übersicht*

*Spieler schlüpft in die Rolle eines Necromancers, der in seinem Unterschlupf Monster erschafft, um damit Terror und Macht zu erlangen. Dabei ist er jedoch nicht alleine, da andere Spieler in der Nähe eigene Unterschüpfe beziehen und marodierende NPCs umgehen. Man kann seinen Unterschlüpf ausbauen oder in der Overworld-Ansicht mit seinem Monster NPC/PC aggressen. So erhält man ruhm, neue Teile und andere Resourcen.*

## *Features*

- *Monster werden ähnlich Frankenstein aus verschiedenen Teilen zusammengesetzt*
  - *verändern die Werte des Monsters*
  - *geben neue Fähigkeiten*
  - *erhaltbar durch Quests, Kämpfe, Herstellbar*
- *Der Necromancer interagiert nicht mit anderen Elementen, er befehligt sein Monster oder seine Minions (Igors)*
  - *bauen Turm aus (Bauigors)*

  - *berichten über anstehende Angriffe (General Igor)*
  - *zeigen Statistiken (Butler Igor)*
  - *setzen das Monster zusammen/Erstellen neue Teile (Fleischer Igor)*
  - *erstellen Tränke für temp. Buffs (Alche Igor)*
  - *schmieden Runen (Runen Igor)*
  - *Spionieren andere PCs aus (Spigor)*
  - *überbringen Nachichten anderer Spieler (Postigor ud. brennender, fliegender Totenschädel)*
  - *Sammeln neue Teile (Boris)*

- *Der Turm besteht aus Sektionen, die getrennt voneinander ausgebaut werden müssen mit Blut (Ingame Währung)*
  - *Runenschmiede (Verbessert Teile dauerhaft, aber limitiert auf 0-3 je Teil) - (Ausbau bringt bessere Runenrezepte)*
  - *Portal (Angriff auf die Oberwelt... zeitliches, automatisches Farmen, PVP Arena) - (Ausbau bringt längere Farmzeiten, niedrigere Cooldowns für Arena)*
  - *Alchemist (Themp. Buffs) - (Ausbau bringt bessere Trankrezepte, länger wirkend)*
  - *Folterkammer (Monsteredit, Teile synthetisieren) - (Ausbau bringt bessere Werte für die Monster (multiplier) )*
  - *Privatgemach (Messagesystem, Ranking, Truhe) - (Ausbau bringt größere Truhe)*

- *Der Kampf gegen andere NPC/PC basiert auf einem Kartenspiel*
  - *Jede Karte repräsentiert eine Aktion des Monsters*
  - *Karten haben werte von 0-9*

  - *eine Karte kann eine andere Stechen, wenn diese eine niedrigere Zahl aufweist. 0 kann alle Stechen*
  - *Karten haben Schaden, Schnelligkeit, Kosten*
  - *Jedes Monster hat ein Deck*
  - *Karten können zu Combos kombiniert werden*
  - *bei Leerem Deck muss man sein Deck ein paar Runden nachladen*
  - *Wenn die LP eines Monsters 0 sind, endet der Kampf*

- *Der Turm kann angegriffen werden. im Overworld-Screen*
  - *in diesem Fall verteidigen die Igors den Turm*
  - *Kann Schäden am Turm hervorrufen*
  - *kann Blut stehlen*

## *Gameplay*

*Man beginnt mit einem Monster, schickt es in den ersten Kampf und erhält ein neues Teil. Dann verbessert man sein Monster und bekommt nach und nach stärkere Widersacher. Nach geraumer Zeit ist man in der Lage, andere Spieler herauszufordern und/oder ihren Turm anzugreifen, um die immer teurer werdenden Teile/Tränke/Runen zu finanzieren und anderen zu schaden.*

*Interaktion durch Gilden, PVP, Raids, Überfälle*

## *Regeln*

- *Ein Necromancer beginnt mit ein paar Teilen, die ein Monster ergeben.*
- *Ein Monster besteht aus 6-8 Teilen: Kopf, 2 Arme, 2 Beine, Torso, (Flügel, Schwanz)*
- *Teile haben Kosten; ein Monster darf abhängig vom Necromancerlevel einen Bestimmten wert nicht übersteigen*

- *Der Turm wird mit Blut ausgebaut*
- *Der Turm besteht aus Sektionen, die einzeln augebaut werden (s.o.)*
- *Igors werden mit Blut bezahlt (Sehr witzig, Igors sind Handlanger und werden nicht bezahlt...)*
- *Immer nur ein Monster kann für längere Zeit farmen geschickt werden*
- *Nach dem Kampf erhält der Gewinner blut und/oder Teile, der Verlierer muss ein*
- *Monster eine Kurze Zeit ausruhen lassen (1-5min)*
- *Ein Necromancer hat gewonnen, wenn er die Weltherrschaft an sich gerissen hat. Dies ist kein fester Zustand, ein anderer Necromancer kann nach einem Cooldown diese für sich beanspruchen. Daraus folgt kein festes Ende.*

## *Level Design*

- *Beginn eines Tutorials nur im Turm, Grundideen (Monsterbau, Kampf, Modifikation) werden erklärt*
- *4 Teilung (Turm, Overworld, Kampfscreen, Dungeonmap)*
  - *Dungeons werden instanziiert und haben verschiedene Level (wie immer eigentlich)*
  - *Alle Spieler habe ihren Unterschlupf in der Selben Overworld*
  - *Jeder Spieler hat einen eigenen Unterschlupf/Turm... von Beginn an*
  - *Für den Kampf wechselt der View in einen Kampfscreen*

# O. Messergebnisse der Beispiel-Implementierungen

| *Entität* | *Strat.* | *Funktionalität* | *Char.* | *LoC* | *LLoC* |
|---|---|---|---|---|---|
| Main | tcp | init | socket | 15 | 13 |
| ObjectHandler | all | init | controller | 4 | 3 |
| GetAvatarRequest | all | GetAvatar | communicationObject | 14 | 11 |
| GetAvatarRequest | all | GetAvatar | configuration | 3 | 0 |
| GetAvatarsRequest | all | GetAvatars | communicationObject | 6 | 5 |
| GetAvatarsRequest | all | GetAvatars | configuration | 2 | 0 |
| MoveAvatarRequest | all | MoveAvatar | communicationObject | 23 | 18 |
| MoveAvatarRequest | all | MoveAvatar | configuration | 4 | 0 |
| NewAvatarRequest | all | NewAvatar | communicationObject | 14 | 11 |
| NewAvatarRequest | all | NewAvatar | configuration | 3 | 0 |
| UpdateAvatarRequest | all | UpdateAvatar | communicationObject | 22 | 17 |
| UpdateAvatarRequest | all | UpdateAvatar | configuration | 4 | 0 |
| AvatarResponse | all | NewAvatar | communicationObject | 38 | 29 |
| AvatarResponse | all | NewAvatar | configuration | 5 | 0 |
| AvatarsResponse | all | GetAvatars | communicationObject | 25 | 19 |
| AvatarsResponse | all | GetAvatars | configuration | 3 | 0 |
| Avatar | all | init | model | 28 | 20 |
| AvatarApi | all | init | api | 4 | 3 |
| AvatarApi | all | NewAvatar | api | 1 | 1 |
| AvatarApi | all | GetAvatar | api | 1 | 1 |
| AvatarApi | all | GetAvatars | api | 1 | 1 |
| AvatarApi | all | UpdateAvatar | api | 1 | 1 |
| AvatarApi | all | MoveAvatar | api | 1 | 1 |
| AvatarApiSupport | all | init | api | 8 | 6 |
| AvatarApiSupport | all | NewAvatar | api | 4 | 2 |
| AvatarApiSupport | all | GetAvatar | api | 4 | 2 |
| AvatarApiSupport | all | GetAvatars | api | 4 | 2 |

| | | | | | |
|---|---|---|---|---|---|
| AvatarApiSupport | all | UpdateAvatar | api | 6 | 4 |
| AvatarApiSupport | all | MoveAvatar | api | 6 | 4 |
| BasicWorld | all | init | model | 24 | 18 |
| Position | all | init | model | 29 | 19 |
| BasicGameApplication | rest | init | controller | 5 | 4 |
| BasicGameApplication | rest | init | configuration | 1 | 0 |
| BasicGameController | rest | init | controller | 9 | 7 |
| BasicGameController | rest | init | configuration | 1 | 0 |
| BasicGameWebService | rest | init | controller | 17 | 16 |
| BasicGameWebService | rest | NewAvatar | controller | 4 | 3 |
| BasicGameWebService | rest | NewAvatar | configuration | 3 | 1 |
| BasicGameWebService | rest | GetAvatar | controller | 4 | 3 |
| BasicGameWebService | rest | GetAvatar | configuration | 3 | 1 |
| BasicGameWebService | rest | GetAvatars | controller | 4 | 3 |
| BasicGameWebService | rest | GetAvatars | configuration | 3 | 1 |
| BasicGameWebService | rest | UpdateAvatar | controller | 4 | 3 |
| BasicGameWebService | rest | UpdateAvatar | configuration | 3 | 1 |
| BasicGameWebService | rest | MoveAvatar | controller | 4 | 3 |
| BasicGameWebService | rest | MoveAvatar | configuration | 3 | 1 |
| BasicGameWebService | rest | init | configuration | 2 | 0 |
| BasicGameExtension | smartfox | init | controller | 23 | 16 |
| BasicGameExtension | smartfox | NewAvatar | controller | 2 | 2 |
| BasicGameExtension | smartfox | GetAvatar | controller | 2 | 2 |
| BasicGameExtension | smartfox | GetAvatars | controller | 2 | 2 |
| BasicGameExtension | smartfox | UpdateAvatar | controller | 2 | 2 |
| BasicGameExtension | smartfox | MoveAvatar | controller | 2 | 2 |
| GetAvatarHandler | smartfox | GetAvatar | controller | 17 | 14 |
| GetAvatarsHandler | smartfox | GetAvatars | controller | 16 | 13 |
| MoveAvatarHandler | smartfox | MoveAvatar | controller | 17 | 14 |
| NewAvatarHandler | smartfox | NewAvatar | controller | 17 | 14 |
| UpdateAvatarHandler | smartfox | UpdateAvatar | controller | 17 | 14 |
| GetAvatarRequest | smartfox | GetAvatar | communicationObject | 11 | 8 |
| MoveAvatarRequest | smartfox | MoveAvatar | communicationObject | 18 | 14 |
| NewAvatarRequest | smartfox | NewAvatar | communicationObject | 11 | 8 |
| UpdateAvatarRequest | smartfox | UpdateAvatar | communicationObject | 14 | 10 |
| AvatarResponse | smartfox | NewAvatar | communicationObject | 10 | 8 |
| AvatarsResponse | smartfox | GetAvatars | communicationObject | 14 | 11 |
| PositionResponse | smartfox | NewAvatar | communicationObject | 9 | 7 |
| Connection | tcp | init | socket | 67 | 49 |

| | | | | | |
|---|---|---|---|---|---|
| JsonMessageHandler | tcp | init | messageHandler | 65 | 57 |
| JsonMessageHandler | tcp | NewAvatar | messageHandler | 0 | 0 |
| JsonMessageHandler | tcp | GetAvatar | messageHandler | 0 | 0 |
| JsonMessageHandler | tcp | GetAvatars | messageHandler | 0 | 0 |
| JsonMessageHandler | tcp | UpdateAvatar | messageHandler | 0 | 0 |
| JsonMessageHandler | tcp | MoveAvatar | messageHandler | 0 | 0 |
| MessageHandler | tcp | init | messageHandler | 4 | 3 |
| Server | tcp | init | socket | 28 | 19 |
| SimpleAvatarApi | tcp | init | api | 13 | 9 |
| SimpleAvatarApi | tcp | NewAvatar | api | 5 | 4 |
| SimpleAvatarApi | tcp | GetAvatar | api | 4 | 3 |
| SimpleAvatarApi | tcp | GetAvatars | api | 5 | 4 |
| SimpleAvatarApi | tcp | UpdateAvatar | api | 4 | 3 |
| SimpleAvatarApi | tcp | MoveAvatar | api | 4 | 3 |
| api | jsboth | init | api | 8 | 7 |
| api | jsboth | NewAvatar | api | 5 | 4 |
| api | jsboth | GetAvatar | api | 6 | 5 |
| api | jsboth | GetAvatars | api | 3 | 2 |
| api | jsboth | UpdateAvatar | api | 11 | 10 |
| api | jsboth | MoveAvatar | api | 15 | 11 |
| model | jsboth | init | model | 69 | 45 |
| rest | jsrest | init | controller | 28 | 21 |
| rest | jsrest | NewAvatar | controller | 3 | 3 |
| rest | jsrest | GetAvatar | controller | 3 | 4 |
| rest | jsrest | GetAvatars | controller | 3 | 4 |
| rest | jsrest | UpdateAvatar | controller | 3 | 3 |
| rest | jsrest | MoveAvatar | controller | 3 | 3 |
| websocket | jswebsocket | init | controller | 24 | 18 |
| Avatar | gameengine | init | model | 16 | 12 |
| Avatar | gameengine | init | model | 6 | 4 |
| Avatar | gameengine | UpdateAvatar | configuration | 2 | 1 |
| Avatar | gameengine | MoveAvatar | configuration | 1 | 0 |
| Avatar | gameengine | init | configuration | 2 | 1 |
| BasicWorld | gameengine | init | model | 16 | 9 |
| BasicWorld | gameengine | init | configuration | 4 | 2 |
| BasicWorld | gameengine | init | model | 12 | 9 |
| BasicWorld | gameengine | GetAvatars | configuration | 2 | 1 |
| BasicWorld | gameengine | NewAvatar | configuration | 1 | 0 |
| BasicWorld | gameengine | GetAvatar | configuration | 1 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| Position | gameengine | init | model | 27 | 19 |
| Position | gameengine | init | configuration | 2 | 1 |

# P. Auswertung der Beispiel-Implementierungen

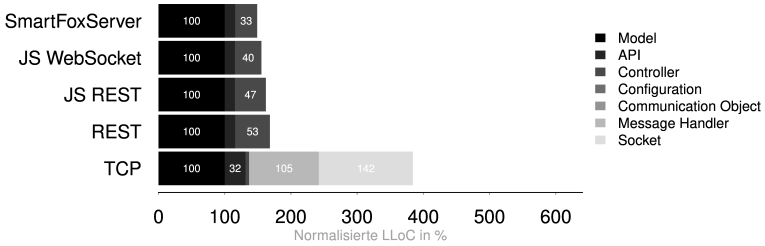



**Abbildung P.1: Darstellung des Aufwands je Characteristik im Schritt *init*.**

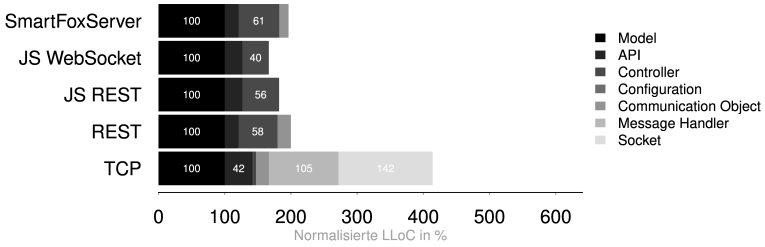



**Abbildung P.2: Darstellung des kumulierten Aufwands je Characteristik von Schritt init bis *getAvatar*.**

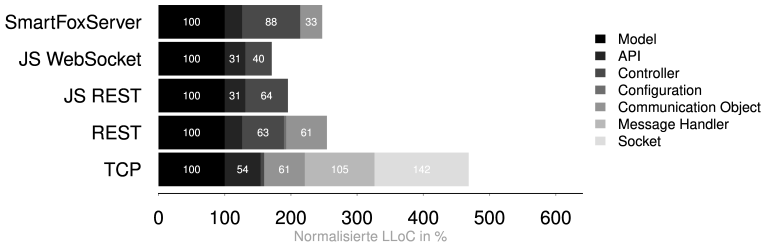



**Abbildung P.3: Darstellung des kumulierten Aufwands je Characteristik von Schritt init bis *getAvatars*.**

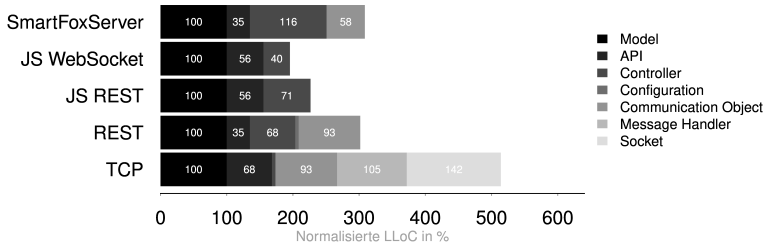



**Abbildung P.4: Darstellung des kumulierten Aufwands je Characteristik von Schritt init bis *moveAvatar*.**

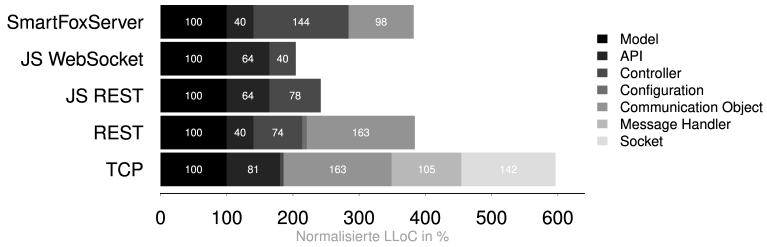



**Abbildung P.5: Darstellung des kumulierten Aufwands je Characteristik von Schritt init bis *newAvatar*.**

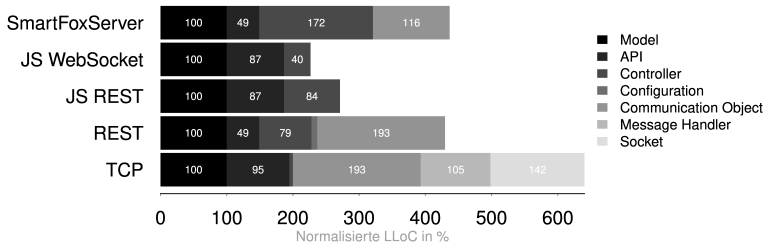



**Abbildung P.6: Darstellung des kumulierten Aufwands je Characteristik von Schritt init bis *updateAvatar*.**

# Q. Messergebnisse der Spielprojekts MMO-Meischta

| *Entität* | *Strat.* | *Funktionalität* | *Char.* | *LoC* | *LLoC* |
|---|---|---|---|---|---|
| MmoMeischtaApi | all | init | api | 12 | 11 |
| MmoMeischtaApi | all | GetCard | api | 1 | 1 |
| MmoMeischtaApi | all | GetLPPercent | api | 1 | 1 |
| MmoMeischtaApi | all | Reload | api | 1 | 1 |
| MmoMeischtaApi | all | RotatingAction | api | 1 | 1 |
| MmoMeischtaApi | all | SetBlood | api | 1 | 1 |
| MmoMeischtaApi | all | GetBlood | api | 1 | 1 |
| MmoMeischtaApi | all | nc.GetMonster | api | 1 | 1 |
| MmoMeischtaApi | all | nc.SetMonster | api | 1 | 1 |
| MmoMeischtaApi | all | GetElement | api | 1 | 1 |
| MmoMeischtaApi | all | SetElement | api | 1 | 1 |
| MmoMeischtaApi | all | GetPlayer1 | api | 1 | 1 |
| MmoMeischtaApi | all | GetPlayer2 | api | 1 | 1 |
| MmoMeischtaApi | all | PlayCard | api | 1 | 1 |
| MmoMeischtaApi | all | AddNecromancer | api | 1 | 1 |
| MmoMeischtaApi | all | GetNecromancer | api | 1 | 1 |
| MmoMeischtaApi | all | AddToBattleList | api | 1 | 1 |
| MmoMeischtaApi | all | Login | api | 1 | 1 |
| MmoMeischtaApiSupport | all | init | api | 48 | 38 |
| MmoMeischtaApiSupport | all | GetCard | api | 7 | 5 |
| MmoMeischtaApiSupport | all | GetLPPercent | api | 7 | 5 |
| MmoMeischtaApiSupport | all | Reload | api | 9 | 6 |
| MmoMeischtaApiSupport | all | RotatingAction | api | 7 | 5 |
| MmoMeischtaApiSupport | all | SetBlood | api | 9 | 6 |
| MmoMeischtaApiSupport | all | GetBlood | api | 7 | 5 |
| MmoMeischtaApiSupport | all | nc.GetMonster | api | 7 | 5 |
| MmoMeischtaApiSupport | all | nc.SetMonster | api | 10 | 7 |

| | | | | | |
|---|---|---|---|---|---|
| MmoMeischtaApiSupport | all | GetElement | api | 9 | 6 |
| MmoMeischtaApiSupport | all | SetElement | api | 10 | 7 |
| MmoMeischtaApiSupport | all | GetPlayer1 | api | 7 | 5 |
| MmoMeischtaApiSupport | all | GetPlayer2 | api | 7 | 5 |
| MmoMeischtaApiSupport | all | PlayCard | api | 7 | 5 |
| MmoMeischtaApiSupport | all | AddNecromancer | api | 4 | 2 |
| MmoMeischtaApiSupport | all | GetNecromancer | api | 4 | 2 |
| MmoMeischtaApiSupport | all | AddToBattleList | api | 4 | 2 |
| MmoMeischtaApiSupport | all | Login | api | 4 | 2 |
| AbstractGameObjectIdentifier | all | init | controller | 26 | 17 |
| BasicApi | all | init | controller | 6 | 5 |
| Controller | all | init | controller | 79 | 58 |
| ErrorResponse | all | init | comm.Object | 30 | 21 |
| Handler | all | init | controller | 21 | 16 |
| JsonMessageHandler | tcp | init | messageHandler | 61 | 52 |
| MessageHandler | tcp | init | messageHandler | 4 | 3 |
| ObjectCollection | all | init | controller | 23 | 15 |
| ObjectHandler | all | init | controller | 5 | 4 |
| SimpleObjectHandler | all | init | controller | 13 | 9 |
| Broadcaster | all | init | controller | 55 | 39 |
| AbstractGameObject | all | init | controller | 19 | 14 |
| AbstractWorld | all | init | controller | 38 | 29 |
| Configurator | all | init | controller | 20 | 16 |
| InternalGameException | all | init | controller | 7 | 5 |
| ObjectRegistry | all | init | controller | 47 | 36 |
| SessionContext | all | init | controller | 6 | 4 |
| Timer | all | init | controller | 6 | 4 |
| Element | all | init | model | 4 | 3 |
| Necromancer | all | init | model | 89 | 70 |
| Session | all | init | model | 106 | 81 |
| World | all | init | model | 116 | 91 |
| CardFactory | all | init | model | 29 | 22 |
| Factory | all | init | model | 33 | 21 |
| ItemFactory | all | init | model | 14 | 12 |
| PartFactory | all | init | model | 6 | 4 |
| StateFactory | all | init | model | 20 | 15 |
| Ingredient | all | init | model | 4 | 3 |
| Item | all | init | model | 34 | 25 |
| PartItem | all | init | model | 16 | 12 |

| | | | | | |
|---|---|---|---|---|---|
| Potion | all | init | model | 4 | 3 |
| Recipe | all | init | model | 4 | 3 |
| Effect | all | init | model | 43 | 31 |
| Monster | all | init | model | 105 | 78 |
| Part | all | init | model | 76 | 59 |
| PartPosition | all | init | model | 4 | 2 |
| BattleMonster | all | init | model | 67 | 51 |
| State | all | init | model | 40 | 30 |
| AttackCard | all | init | model | 30 | 22 |
| Card | all | init | model | 91 | 67 |
| CardHandler | all | init | controller | 21 | 18 |
| MagicCard | all | init | model | 3 | 2 |
| NormalCard | all | init | model | 32 | 23 |
| SpecialCard | all | init | model | 6 | 5 |
| Arm | all | init | model | 12 | 8 |
| Corpus | all | init | model | 12 | 8 |
| Head | all | init | model | 12 | 8 |
| Leg | all | init | model | 12 | 8 |
| Tail | all | init | model | 12 | 8 |
| Wing | all | init | model | 12 | 8 |
| Laboratory | all | init | model | 11 | 6 |
| Portal | all | init | model | 11 | 6 |
| Room | all | init | model | 27 | 21 |
| Runeforge | all | init | model | 11 | 6 |
| Solar | all | init | model | 17 | 11 |
| TortureChamber | all | init | model | 11 | 6 |
| Tower | all | init | model | 21 | 15 |
| BattleMonsterWebService | rest | init | controller | 26 | 20 |
| BattleMonsterWebService | rest | GetCard | controller | 3 | 2 |
| BattleMonsterWebService | rest | GetCard | configuration | 3 | 1 |
| BattleMonsterWebService | rest | bm.GetMonster | controller | 3 | 2 |
| BattleMonsterWebService | rest | bm.GetMonster | configuration | 3 | 1 |
| BattleMonsterWebService | rest | Reload | controller | 3 | 2 |
| BattleMonsterWebService | rest | Reload | configuration | 3 | 1 |
| BattleMonsterWebService | rest | RotatingAction | controller | 3 | 2 |
| BattleMonsterWebService | rest | RotatingAction | configuration | 3 | 1 |
| BattleMonsterWebService | rest | bm.SetMonster | controller | 3 | 2 |
| BattleMonsterWebService | rest | bm.SetMonster | configuration | 3 | 1 |
| MmoMeischtaApplication | rest | init | controller | 6 | 4 |

| | | | | | |
|---|---|---|---|---|---|
| MmoMeischtaController | rest | init | controller | 37 | 27 |
| NecromancerWebService | rest | init | controller | 16 | 13 |
| NecromancerWebService | rest | SetBlood | controller | 3 | 2 |
| NecromancerWebService | rest | SetBlood | configuration | 3 | 1 |
| NecromancerWebService | rest | GetBlood | controller | 3 | 2 |
| NecromancerWebService | rest | GetBlood | configuration | 3 | 1 |
| NecromancerWebService | rest | nc.GetMonster | controller | 3 | 2 |
| NecromancerWebService | rest | nc.GetMonster | configuration | 3 | 1 |
| NecromancerWebService | rest | nc.SetMonster | controller | 3 | 2 |
| NecromancerWebService | rest | nc.SetMonster | configuration | 3 | 1 |
| NormalCardWebService | rest | init | controller | 13 | 10 |
| NormalCardWebService | rest | GetElement | controller | 3 | 2 |
| NormalCardWebService | rest | GetElement | configuration | 3 | 1 |
| NormalCardWebService | rest | SetElement | controller | 3 | 2 |
| NormalCardWebService | rest | SetElement | configuration | 3 | 1 |
| SessionWebService | rest | init | controller | 14 | 11 |
| SessionWebService | rest | GetPlayer1 | controller | 3 | 2 |
| SessionWebService | rest | GetPlayer1 | configuration | 3 | 1 |
| SessionWebService | rest | GetPlayer2 | controller | 3 | 2 |
| SessionWebService | rest | GetPlayer2 | configuration | 3 | 1 |
| SessionWebService | rest | PlayCard | controller | 3 | 2 |
| SessionWebService | rest | PlayCard | configuration | 3 | 1 |
| WorldWebService | rest | init | controller | 17 | 14 |
| WorldWebService | rest | AddNecromancer | controller | 3 | 2 |
| WorldWebService | rest | AddNecromancer | configuration | 3 | 1 |
| WorldWebService | rest | GetNecromancer | controller | 3 | 2 |
| WorldWebService | rest | GetNecromancer | configuration | 3 | 1 |
| WorldWebService | rest | AddToBattleList | controller | 3 | 2 |
| WorldWebService | rest | AddToBattleList | configuration | 3 | 1 |
| WorldWebService | rest | Login | controller | 3 | 2 |
| WorldWebService | rest | Login | configuration | 3 | 1 |
| MmoMeischta | rest | init | controller | 67 | 62 |
| MmoMeischta | rest | GetCard | controller | 1 | 1 |
| MmoMeischta | rest | bm.GetMonster | controller | 1 | 1 |
| MmoMeischta | rest | Reload | controller | 1 | 1 |
| MmoMeischta | rest | RotatingAction | controller | 1 | 1 |
| MmoMeischta | rest | bm.SetMonster | controller | 1 | 1 |
| MmoMeischta | rest | SetBlood | controller | 1 | 1 |
| MmoMeischta | rest | GetBlood | controller | 1 | 1 |

| | | | | | |
|---|---|---|---|---|---|
| MmoMeischta | rest | GetElement | controller | 1 | 1 |
| MmoMeischta | rest | SetElement | controller | 1 | 1 |
| MmoMeischta | rest | GetPlayer1 | controller | 1 | 1 |
| MmoMeischta | rest | GetPlayer2 | controller | 1 | 1 |
| MmoMeischta | rest | PlayCard | controller | 1 | 1 |
| MmoMeischta | rest | AddNecromancer | controller | 1 | 1 |
| MmoMeischta | rest | GetNecromancer | controller | 1 | 1 |
| MmoMeischta | rest | AddToBattleList | controller | 1 | 1 |
| MmoMeischta | rest | Login | controller | 1 | 1 |
| AddNecromancerHandler | all | AddNecromancer | controller | 11 | 8 |
| AddToBattleListHandler | all | AddToBattleList | controller | 12 | 9 |
| GetNecromancerHandler | all | GetNecromancer | controller | 17 | 12 |
| LoginHandler | all | Login | controller | 12 | 10 |
| MmoMeischtaHandler | all | init | controller | 6 | 5 |
| GetCardHandler | all | GetCard | controller | 18 | 13 |
| GetLpPercentHandler | all | init | controller | 11 | 8 |
| GetMonsterHandler | all | bm.GetMonster | controller | 18 | 13 |
| ReloadHandler | all | Reload | controller | 13 | 9 |
| RotatingActionHandler | all | RotatingAction | controller | 11 | 8 |
| SetMonsterHandler | all | bm.SetMonster | controller | 13 | 9 |
| GetBloodHandler | all | GetBlood | controller | 11 | 8 |
| GetMonsterHandler | all | nc.GetMonster | controller | 18 | 13 |
| SetBloodHandler | all | SetBlood | controller | 13 | 9 |
| SetMonsterHandler | all | nc.SetMonster | controller | 13 | 9 |
| GetElementHandler | all | GetElement | controller | 18 | 13 |
| SetElementHandler | all | SetElement | controller | 13 | 9 |
| GetPlayer1Handler | all | GetPlayer1 | controller | 18 | 13 |
| GetPlayer2Handler | all | GetPlayer2 | controller | 18 | 13 |
| PlayCardHandler | all | PlayCard | controller | 13 | 9 |
| AddNecromancerRequest | all | AddNecromancer | comm.Object | 19 | 14 |
| AddNecromancerRequest | all | AddNecromancer | configuration | 3 | 0 |
| AddToBattleListRequest | all | AddToBattleList | comm.Object | 19 | 14 |
| AddToBattleListRequest | all | AddToBattleList | configuration | 3 | 0 |
| GetNecromancerRequest | all | GetNecromancer | comm.Object | 19 | 14 |
| GetNecromancerRequest | all | GetNecromancer | configuration | 3 | 0 |
| LoginRequest | all | Login | comm.Object | 27 | 20 |
| LoginRequest | all | Login | configuration | 4 | 0 |
| GetCardRequest | all | GetCard | comm.Object | 17 | 13 |
| GetCardRequest | all | GetCard | configuration | 3 | 0 |

| | | | | | |
|---|---|---|---|---|---|
| GetLpPercentRequest | all | init | comm.Object | 7 | 6 |
| GetLpPercentRequest | all | init | configuration | 2 | 0 |
| GetMonsterRequest | all | bm.GetMonster | comm.Object | 7 | 6 |
| GetMonsterRequest | all | bm.GetMonster | configuration | 2 | 0 |
| ReloadRequest | all | Reload | comm.Object | 7 | 6 |
| ReloadRequest | all | Reload | configuration | 2 | 0 |
| RotatingActionRequest | all | RotatingAction | comm.Object | 17 | 13 |
| RotatingActionRequest | all | RotatingAction | configuration | 3 | 0 |
| SetMonsterRequest | all | bm.SetMonster | comm.Object | 16 | 12 |
| SetMonsterRequest | all | bm.SetMonster | configuration | 2 | 0 |
| GetBloodRequest | all | GetBlood | comm.Object | 7 | 6 |
| GetBloodRequest | all | GetBlood | configuration | 2 | 0 |
| GetMonsterRequest | all | nc.GetMonster | comm.Object | 8 | 7 |
| GetMonsterRequest | all | nc.GetMonster | configuration | 3 | 0 |
| SetBloodRequest | all | SetBlood | comm.Object | 16 | 12 |
| SetBloodRequest | all | SetBlood | configuration | 2 | 0 |
| SetMonsterRequest | all | nc.SetMonster | comm.Object | 17 | 13 |
| SetMonsterRequest | all | nc.SetMonster | configuration | 3 | 0 |
| GetElementRequest | all | GetElement | comm.Object | 7 | 6 |
| GetElementRequest | all | GetElement | configuration | 2 | 0 |
| SetElementRequest | all | SetElement | comm.Object | 18 | 14 |
| SetElementRequest | all | SetElement | configuration | 3 | 0 |
| GetPlayer1Request | all | GetPlayer1 | comm.Object | 7 | 6 |
| GetPlayer1Request | all | GetPlayer1 | configuration | 2 | 0 |
| GetPlayer2Request | all | GetPlayer2 | comm.Object | 7 | 6 |
| GetPlayer2Request | all | GetPlayer2 | configuration | 2 | 0 |
| PlayCardRequest | all | PlayCard | comm.Object | 21 | 16 |
| PlayCardRequest | all | PlayCard | configuration | 3 | 0 |
| BattleMonsterResponse | all | RotatingAction | comm.Object | 30 | 23 |
| BattleMonsterResponse | all | init | configuration | 4 | 0 |
| BloodResponse | all | GetBlood | comm.Object | 21 | 16 |
| BloodResponse | all | init | configuration | 3 | 0 |
| BooleanResponse | all | Login | comm.Object | 19 | 14 |
| BooleanResponse | all | init | configuration | 3 | 0 |
| CardResponse | all | GetCard | comm.Object | 91 | 68 |
| CardResponse | all | init | configuration | 2 | 0 |
| DoubleResponse | all | GetLPPercent | comm.Object | 19 | 14 |
| DoubleResponse | all | init | configuration | 3 | 0 |
| ElementResponse | all | GetElement | comm.Object | 22 | 17 |

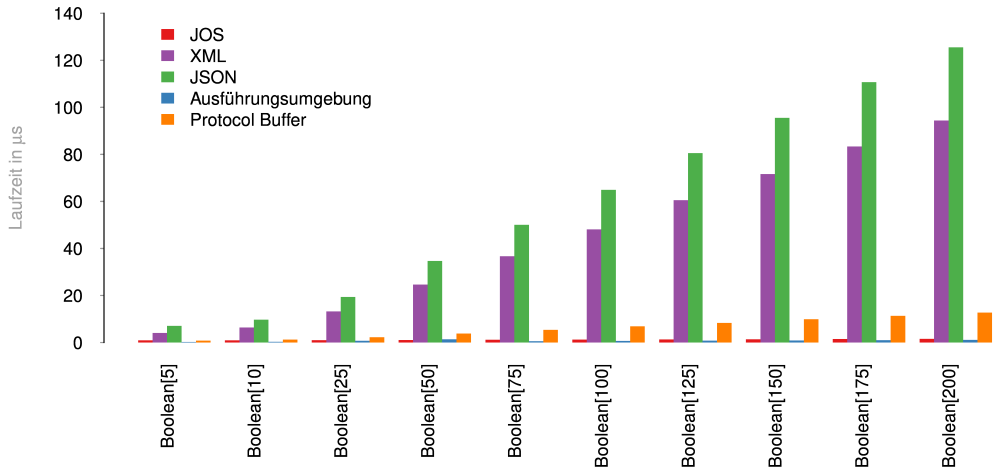
| | | | | | |
|---|---|---|---|---|---|
| ElementResponse | all | init | configuration | 3 | 0 |
| ItemResponse | all | AddToBattleList | comm.Object | 39 | 30 |
| ItemResponse | all | init | configuration | 5 | 0 |
| LoginResponse | all | Login | comm.Object | 11 | 8 |
| LoginResponse | all | Login | configuration | 2 | 0 |
| MonsterResponse | all | bm.GetMonster | comm.Object | 39 | 30 |
| MonsterResponse | all | init | configuration | 5 | 0 |
| NecromancerResponse | all | GetPlayer1 | comm.Object | 55 | 42 |
| NecromancerResponse | all | init | configuration | 7 | 0 |
| SessionResponse | all | AddToBattleList | comm.Object | 47 | 36 |
| SessionResponse | all | init | configuration | 6 | 0 |
| TowerResponse | all | GetPlayer1 | comm.Object | 13 | 10 |
| TowerResponse | all | init | configuration | 2 | 0 |
| DataHandler | smartfox | init | controller | 5 | 4 |
| IFSFObjectHandler | smartfox | init | controller | 52 | 38 |
| IFSFObjectHandler | smartfox | GetCard | controller | 20 | 18 |
| IFSFObjectHandler | smartfox | bm.GetMonster | controller | 15 | 12 |
| IFSFObjectHandler | smartfox | Reload | controller | 6 | 5 |
| IFSFObjectHandler | smartfox | RotatingAction | controller | 18 | 15 |
| IFSFObjectHandler | smartfox | bm.SetMonster | controller | 7 | 5 |
| IFSFObjectHandler | smartfox | SetBlood | controller | 7 | 6 |
| IFSFObjectHandler | smartfox | GetBlood | controller | 13 | 11 |
| IFSFObjectHandler | smartfox | nc.GetMonster | controller | 6 | 5 |
| IFSFObjectHandler | smartfox | nc.SetMonster | controller | 7 | 6 |
| IFSFObjectHandler | smartfox | GetElement | controller | 13 | 11 |
| IFSFObjectHandler | smartfox | SetElement | controller | 7 | 6 |
| IFSFObjectHandler | smartfox | GetPlayer1 | controller | 23 | 20 |
| IFSFObjectHandler | smartfox | GetPlayer2 | controller | 6 | 5 |
| IFSFObjectHandler | smartfox | PlayCard | controller | 7 | 6 |
| IFSFObjectHandler | smartfox | AddNecromancer | controller | 7 | 6 |
| IFSFObjectHandler | smartfox | GetNecromancer | controller | 7 | 6 |
| IFSFObjectHandler | smartfox | AddToBattleList | controller | 26 | 23 |
| IFSFObjectHandler | smartfox | Login | controller | 21 | 18 |
| IFSFObjectHandler | smartfox | init | controller | 7 | 6 |
| MmoMeischtaExtension | smartfox | init | controller | 37 | 27 |
| SmartFoxHandlerWrapper | smartfox | init | controller | 16 | 12 |
| Connection | tcp | init | socket | 68 | 50 |
| Server | tcp | init | socket | 29 | 20 |
| StartTcp | tcp | init | socket | 79 | 44 |

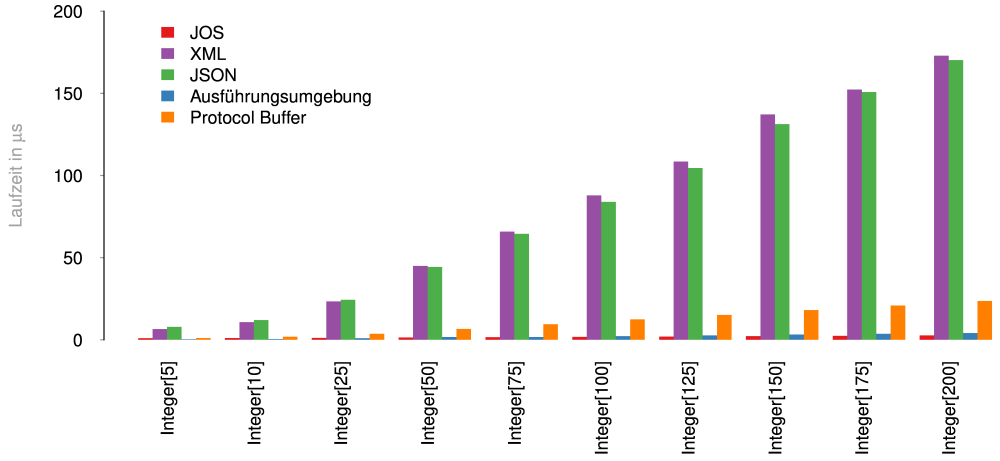
| | | | | | |
|---|---|---|---|---|---|
| GenerateJSCode | gameengine | init | model | 7 | 5 |
| Server | gameengine | init | model | 13 | 11 |
| Element | gameengine | init | model | 4 | 3 |
| Necromancer | gameengine | init | model | 74 | 59 |
| Necromancer | gameengine | init | model | 15 | 11 |
| Necromancer | gameengine | nc.GetMonster | configuration | 2 | 1 |
| Necromancer | gameengine | nc.SetMonster | configuration | 1 | 0 |
| Necromancer | gameengine | GetBlood | configuration | 1 | 0 |
| Necromancer | gameengine | SetBlood | configuration | 1 | 0 |
| Necromancer | gameengine | init | configuration | 2 | 1 |
| Session | gameengine | init | model | 71 | 55 |
| Session | gameengine | init | model | 35 | 26 |
| Session | gameengine | GetPlayer1 | configuration | 2 | 1 |
| Session | gameengine | GetPlayer2 | configuration | 1 | 0 |
| Session | gameengine | PlayCard | configuration | 1 | 0 |
| Session | gameengine | init | configuration | 2 | 1 |
| World | gameengine | init | model | 76 | 57 |
| World | gameengine | init | configuration | 6 | 2 |
| World | gameengine | init | model | 49 | 40 |
| World | gameengine | GetNecromancer | configuration | 2 | 1 |
| World | gameengine | AddNecromancer | configuration | 3 | 1 |
| World | gameengine | Login | configuration | 1 | 0 |
| World | gameengine | AddToBattleList | configuration | 1 | 0 |
| CardFactory | gameengine | init | model | 29 | 22 |
| Factory | gameengine | init | model | 28 | 19 |
| ItemFactory | gameengine | init | model | 14 | 12 |
| PartFactory | gameengine | init | model | 6 | 4 |
| StateFactory | gameengine | init | model | 15 | 12 |
| Ingredient | gameengine | init | model | 4 | 3 |
| Item | gameengine | init | model | 34 | 25 |
| Item | gameengine | init | configuration | 2 | 1 |
| PartItem | gameengine | init | model | 16 | 12 |
| PartItem | gameengine | init | configuration | 2 | 1 |
| Potion | gameengine | init | model | 4 | 3 |
| Recipe | gameengine | init | model | 4 | 3 |
| Effect | gameengine | init | model | 43 | 31 |
| Monster | gameengine | init | model | 105 | 78 |
| Monster | gameengine | init | configuration | 2 | 1 |
| Part | gameengine | init | model | 76 | 59 |

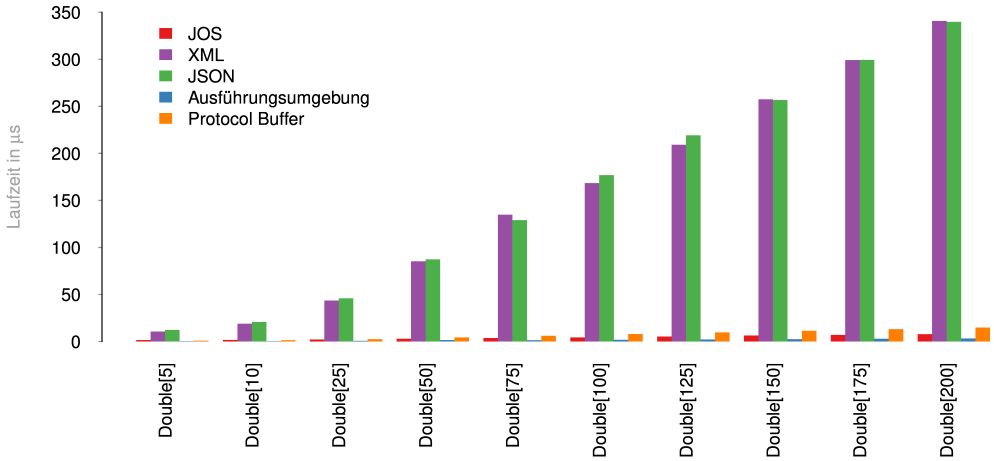
| | | | | | |
|---|---|---|---|---|---|
| Part | gameengine | init | configuration | 2 | 1 |
| PartPosition | gameengine | init | model | 4 | 2 |
| BattleMonster | gameengine | init | model | 38 | 31 |
| BattleMonster | gameengine | init | model | 29 | 20 |
| BattleMonster | gameengine | bm.GetMonster | configuration | 2 | 1 |
| BattleMonster | gameengine | bm.SetMonster | configuration | 1 | 0 |
| BattleMonster | gameengine | Reload | configuration | 1 | 0 |
| BattleMonster | gameengine | RotatingAction | configuration | 1 | 0 |
| BattleMonster | gameengine | GetCard | configuration | 1 | 0 |
| BattleMonster | gameengine | GetLPPercent | configuration | 1 | 0 |
| BattleMonster | gameengine | init | configuration | 2 | 1 |
| State | gameengine | init | model | 40 | 30 |
| State | gameengine | init | configuration | 2 | 1 |
| AttackCard | gameengine | init | model | 30 | 22 |
| AttackCard | gameengine | init | configuration | 2 | 1 |
| Card | gameengine | init | model | 93 | 68 |
| Card | gameengine | init | configuration | 2 | 1 |
| CardHandler | gameengine | init | model | 21 | 18 |
| MagicCard | gameengine | init | model | 3 | 2 |
| MagicCard | gameengine | init | configuration | 2 | 1 |
| NormalCard | gameengine | init | model | 26 | 19 |
| NormalCard | gameengine | init | model | 6 | 4 |
| NormalCard | gameengine | GetElement | configuration | 2 | 1 |
| NormalCard | gameengine | SetElement | configuration | 1 | 0 |
| NormalCard | gameengine | init | configuration | 2 | 1 |
| SpecialCard | gameengine | init | model | 6 | 5 |
| SpecialCard | gameengine | init | configuration | 2 | 1 |
| Arm | gameengine | init | model | 12 | 8 |
| Arm | gameengine | init | configuration | 2 | 1 |
| Corpus | gameengine | init | model | 12 | 8 |
| Corpus | gameengine | init | configuration | 2 | 1 |
| Head | gameengine | init | model | 12 | 8 |
| Head | gameengine | init | configuration | 2 | 1 |
| Leg | gameengine | init | model | 12 | 8 |
| Leg | gameengine | init | configuration | 2 | 1 |
| Tail | gameengine | init | model | 12 | 8 |
| Tail | gameengine | init | configuration | 2 | 1 |
| Wing | gameengine | init | model | 12 | 8 |
| Wing | gameengine | init | configuration | 2 | 1 |

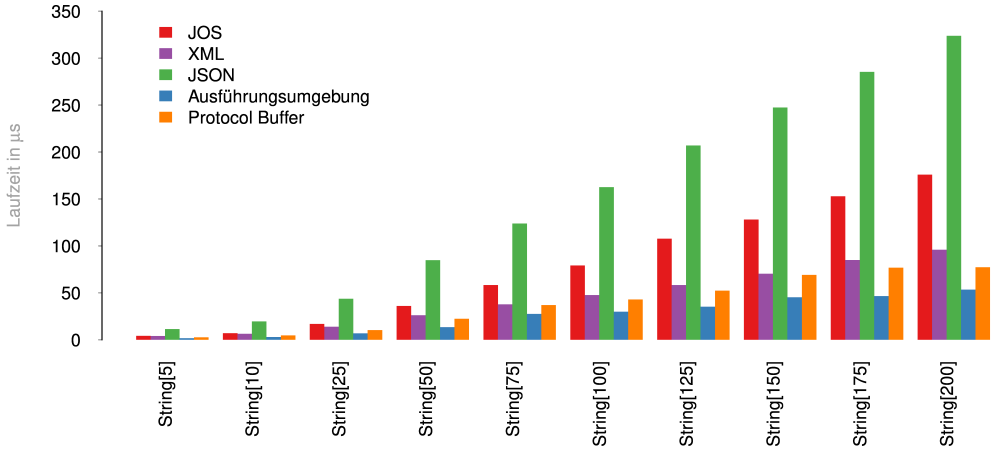
| | | | | | |
|---|---|---|---|---|---|
| Laboratory | gameengine | init | model | 11 | 6 |
| Portal | gameengine | init | model | 11 | 6 |
| Room | gameengine | init | model | 27 | 21 |
| Room | gameengine | init | configuration | 2 | 1 |
| Runeforge | gameengine | init | model | 11 | 6 |
| Solar | gameengine | init | model | 17 | 11 |
| TortureChamber | gameengine | init | model | 11 | 6 |
| Tower | gameengine | init | model | 21 | 15 |
| Tower | gameengine | init | configuration | 2 | 1 |

# R. Serialisierung von Reihungen

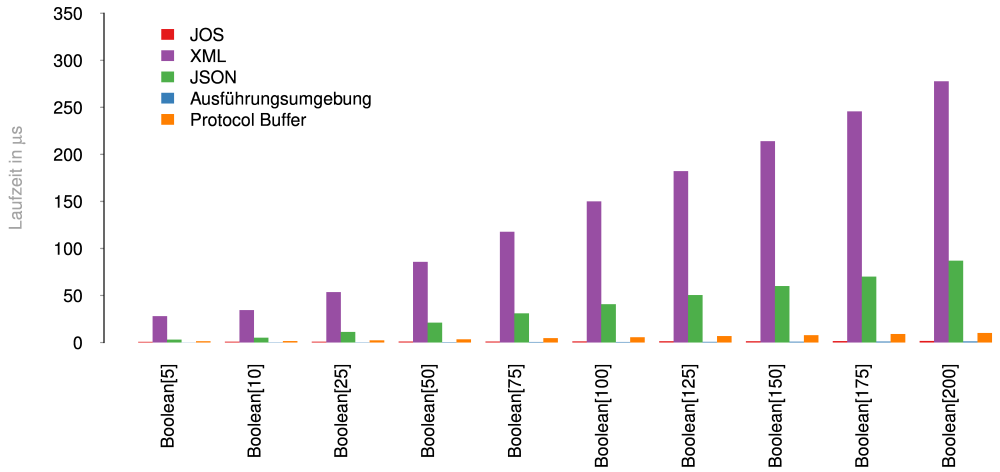


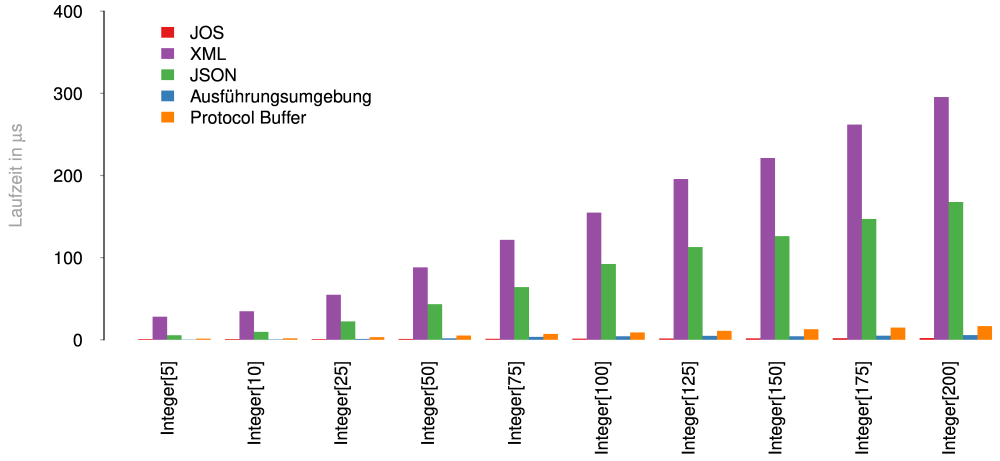



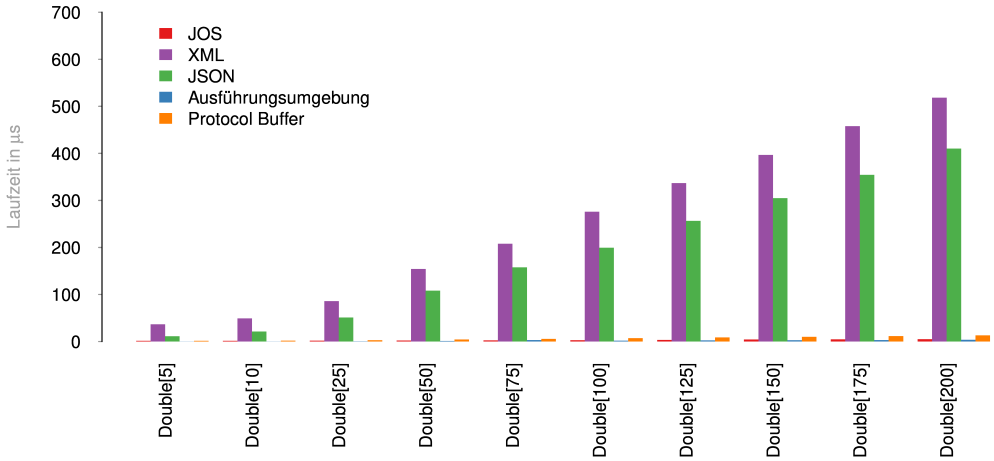

Laufzeit in µs
350
300
250
200
150
100
50
0
JOS
XML
JSON
Ausführungsumgebung
Protocol Buffer
Double[5]
Double[10]
Double[25]
Double[50]
Double[75]
Double[100]
Double[125]
Double[150]
Double[175]
Double[200]

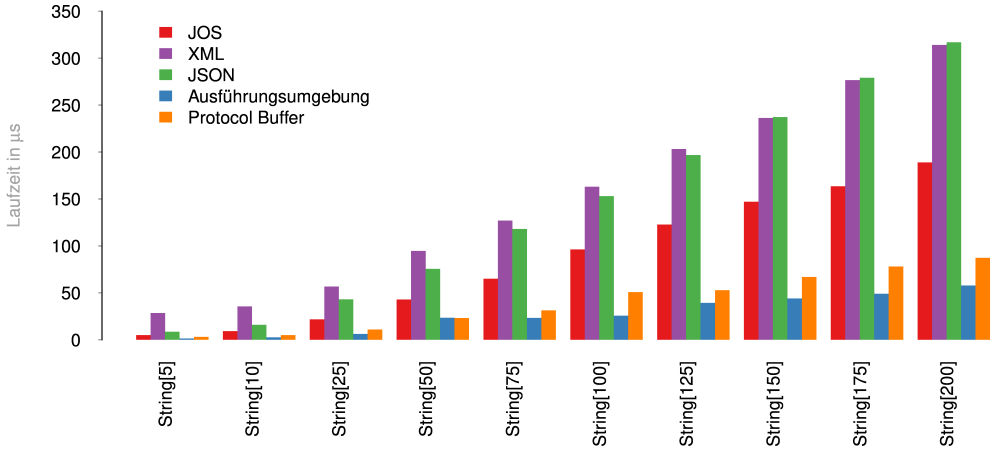

Laufzeit in µs
350
300
250
200
150
100
50
0
JOS
XML
JSON
Ausführungsumgebung
Protocol Buffer
String[5]
String[10]
String[25]
String[50]
String[75]
String[100]
String[125]
String[150]
String[175]
String[200]

# S. Deserialisierung von Reihungen




Laufzeit in µs
700
600
500
400
300
200
100
0
JOS
XML
JSON
Ausführungsumgebung
Protocol Buffer
Double[5]
Double[10]
Double[25]
Double[50]
Double[75]
Double[100]
Double[125]
Double[150]
Double[175]
Double[200]


Laufzeit in µs
350
300
250
200
150
100
50
0
JOS
XML
JSON
Ausführungsumgebung
Protocol Buffer
String[5]
String[10]
String[25]
String[50]
String[75]
String[100]
String[125]
String[150]
String[175]
String[200]

# T. Messergebnisse Serialisierung

Ergebnis der Messung der Ausführungszeit bei der Serialisierung von unterschiedlichen Objekten in Nanosekunden.

| | *JOS* | *XML* | *JSON* | *Ausführungsumgebung* | *Protocol Buffer* |
|---|---|---|---|---|---|
| *Boolean[10]* | 917 | 6335 | 9657 | 267 | 1239 |
| *Boolean[100]* | 1221 | 48081 | 64872 | 650 | 6896 |
| *Boolean[125]* | 1297 | 60497 | 80485 | 754 | 8297 |
| *Boolean[150]* | 1377 | 71590 | 95468 | 867 | 9898 |
| *Boolean[175]* | 1459 | 83313 | 110677 | 972 | 11289 |
| *Boolean[200]* | 1528 | 94387 | 125489 | 1076 | 12695 |
| *Boolean[25]* | 966 | 13174 | 19365 | 711 | 2253 |
| *Boolean[5]* | 897 | 4050 | 7077 | 243 | 811 |
| *Boolean[50]* | 1057 | 24588 | 34646 | 1370 | 3808 |
| *Boolean[75]* | 1140 | 36644 | 49989 | 549 | 5332 |
| *Boolean* | 490 | 1634 | 2557 | 207 | 142 |
| *Complex* | 5132 | 47324 | 64867 | 2012 | 6257 |
| *Double[10]* | 1638 | 18872 | 20743 | 388 | 1282 |
| *Double[100]* | 4331 | 168341 | 176766 | 1746 | 7888 |
| *Double[125]* | 5225 | 209173 | 219061 | 2102 | 9565 |
| *Double[150]* | 6319 | 257426 | 256574 | 2460 | 11410 |
| *Double[175]* | 7011 | 299155 | 299305 | 2829 | 13065 |
| *Double[200]* | 7803 | 340691 | 339713 | 3166 | 14756 |
| *Double[25]* | 2081 | 43506 | 45889 | 675 | 2480 |
| *Double[5]* | 1495 | 10649 | 12254 | 312 | 819 |
| *Double[50]* | 2844 | 85189 | 87295 | 1429 | 4331 |
| *Double[75]* | 3647 | 134726 | 128969 | 1363 | 6062 |
| *Double* | 519 | 2720 | 3687 | 218 | 168 |
| *Integer[10]* | 991 | 10672 | 11919 | 425 | 1830 |
| *Integer[100]* | 1752 | 87846 | 83883 | 2180 | 12366 |
| *Integer[125]* | 1961 | 108467 | 104533 | 2660 | 15090 |
| *Integer[150]* | 2184 | 137153 | 131226 | 3148 | 18064 |

| | | | | | |
|---|---|---|---|---|---|
| *Integer[175]* | 2408 | 152256 | 150781 | 3622 | 20827 |
| *Integer[200]* | 2639 | 172848 | 170165 | 4110 | 23600 |
| *Integer[25]* | 1123 | 23303 | 24314 | 908 | 3635 |
| *Integer[5]* | 944 | 6486 | 7874 | 327 | 1048 |
| *Integer[50]* | 1347 | 44962 | 44345 | 1612 | 6559 |
| *Integer[75]* | 1561 | 65805 | 64503 | 1679 | 9433 |
| *Integer* | 507 | 1818 | 2760 | 226 | 194 |
| *Object[10]* | 6396 | 9379 | 39392 | 2552 | 2408 |
| *Object* | 992 | 2254 | 5862 | 438 | 239 |
| *String[10]* | 7033 | 6381 | 19452 | 2962 | 4681 |
| *String[100]* | 79099 | 47552 | 162506 | 29866 | 42845 |
| *String[125]* | 107600 | 58180 | 206942 | 35255 | 52316 |
| *String[150]* | 128075 | 70380 | 247297 | 45291 | 69021 |
| *String[175]* | 152841 | 84949 | 285321 | 46440 | 76711 |
| *String[200]* | 175791 | 95903 | 323769 | 53448 | 77195 |
| *String[25]* | 16916 | 13872 | 43618 | 6887 | 10312 |
| *String[5]* | 4110 | 4036 | 11402 | 1587 | 2650 |
| *String[50]* | 35932 | 26106 | 84692 | 13374 | 22273 |
| *String[75]* | 58268 | 37638 | 123839 | 27496 | 36872 |
| *String* | 1058 | 1779 | 4115 | 456 | 413 |
| *Test* | 5462 | 14669 | 54870 | 3717 | 4915 |

# U. Messergebnisse Deserialisierung

Ergebnis der Messung der Ausführungszeit bei der Deserialisierung von unterschiedlichen Objekten in Nanosekunden.

| | *JOS* | *XML* | *JSON* | *Ausführungsumgebung* | *Protocol Buffer* |
|---|---|---|---|---|---|
| *Boolean[10]* | 782 | 34336 | 5079 | 173 | 1471 |
| *Boolean[100]* | 1162 | 149944 | 40609 | 616 | 5615 |
| *Boolean[125]* | 1263 | 182130 | 50435 | 737 | 6777 |
| *Boolean[150]* | 1365 | 213946 | 60015 | 873 | 7781 |
| *Boolean[175]* | 1472 | 245686 | 70007 | 987 | 9009 |
| *Boolean[200]* | 1574 | 277559 | 87010 | 1112 | 10056 |
| *Boolean[25]* | 858 | 53597 | 11266 | 286 | 2244 |
| *Boolean[5]* | 764 | 27953 | 3110 | 149 | 1252 |
| *Boolean[50]* | 951 | 85666 | 21143 | 369 | 3432 |
| *Boolean[75]* | 1056 | 117730 | 30923 | 487 | 4596 |
| *Boolean* | 389 | 20308 | 1139 | 75 | 910 |
| *Complex* | 4591 | 93792 | 59708 | 1878 | 5494 |
| *Double[10]* | 1352 | 48923 | 21140 | 294 | 1556 |
| *Double[100]* | 2751 | 275821 | 199403 | 1833 | 7004 |
| *Double[125]* | 3175 | 336736 | 256255 | 2253 | 8526 |
| *Double[150]* | 4103 | 396432 | 304683 | 2673 | 9894 |
| *Double[175]* | 4477 | 457821 | 354152 | 3088 | 11481 |
| *Double[200]* | 4861 | 518237 | 410043 | 3507 | 12856 |
| *Double[25]* | 1574 | 85813 | 51073 | 548 | 2582 |
| *Double[5]* | 1266 | 36427 | 11239 | 214 | 1264 |
| *Double[50]* | 1958 | 154055 | 108019 | 1147 | 4129 |
| *Double[75]* | 2348 | 207770 | 157460 | 2893 | 5609 |
| *Double* | 398 | 22875 | 2749 | 86 | 894 |
| *Integer[10]* | 788 | 34636 | 9549 | 409 | 1738 |
| *Integer[100]* | 1350 | 154768 | 92188 | 4163 | 8940 |
| *Integer[125]* | 1503 | 195604 | 112871 | 4786 | 10911 |
| *Integer[150]* | 1659 | 221229 | 126097 | 4291 | 12740 |

| | | | | | |
|---|---|---|---|---|---|
| *Integer[175]* | 1846 | 261870 | 147061 | 4976 | 14750 |
| *Integer[200]* | 1990 | 295407 | 167707 | 5668 | 16518 |
| *Integer[25]* | 876 | 54730 | 22321 | 947 | 3141 |
| *Integer[5]* | 764 | 27957 | 5378 | 255 | 1378 |
| *Integer[50]* | 1032 | 88007 | 43230 | 1729 | 5179 |
| *Integer[75]* | 1193 | 121507 | 64051 | 3410 | 7144 |
| *Integer* | 399 | 20569 | 1571 | 87 | 941 |
| *Object[10]* | 4636 | 40819 | 14362 | 967 | 2881 |
| *Object* | 864 | 22810 | 2099 | 144 | 1058 |
| *String[10]* | 9154 | 35447 | 15833 | 2581 | 4882 |
| *String[100]* | 96192 | 162943 | 152890 | 25568 | 50767 |
| *String[125]* | 122749 | 203063 | 196716 | 39335 | 52789 |
| *String[150]* | 146937 | 236175 | 237200 | 43921 | 66798 |
| *String[175]* | 163494 | 276590 | 279044 | 48965 | 78050 |
| *String[200]* | 188855 | 314019 | 316773 | 57731 | 87301 |
| *String[25]* | 21750 | 56689 | 42963 | 6234 | 10888 |
| *String[5]* | 4936 | 28486 | 8552 | 1364 | 2977 |
| *String[50]* | 42889 | 94627 | 75440 | 23513 | 23139 |
| *String[75]* | 64935 | 126964 | 117998 | 23355 | 31259 |
| *String* | 1232 | 20545 | 2460 | 307 | 1163 |
| *Test* | 4905 | 55396 | 24530 | 3166 | 6269 |

# V. Messergebnisse Datenvolumen

Ergebnis der Messung des Datenvolumens bei der Serialisierung von unterschiedlichen Objekten in Byte.

| | *JOS* | *XML* | *JSON* | *Ausführungsumgebung* | *Protocol Buffer* |
|---|---|---|---|---|---|
| *Boolean[10]* | 118 | 271 | 65 | 16 | 20 |
| *Boolean[100]* | 208 | 1846 | 560 | 106 | 200 |
| *Boolean[125]* | 233 | 2284 | 698 | 131 | 250 |
| *Boolean[150]* | 258 | 2721 | 835 | 156 | 300 |
| *Boolean[175]* | 283 | 3159 | 973 | 181 | 350 |
| *Boolean[200]* | 308 | 3597 | 1111 | 206 | 400 |
| *Boolean[25]* | 133 | 534 | 148 | 31 | 50 |
| *Boolean[5]* | 113 | 184 | 38 | 11 | 10 |
| *Boolean[50]* | 158 | 971 | 285 | 56 | 100 |
| *Boolean[75]* | 183 | 1410 | 424 | 81 | 150 |
| *Boolean* | 73 | 81 | 11 | 5 | 2 |
| *Complex* | 733 | 937 | 696 | 299 | 360 |
| *Double[10]* | 187 | 407 | 203 | 86 | 90 |
| *Double[100]* | 907 | 3220 | 1936 | 806 | 900 |
| *Double[125]* | 1107 | 4002 | 2418 | 1006 | 1125 |
| *Double[150]* | 1307 | 4785 | 2901 | 1206 | 1350 |
| *Double[175]* | 1507 | 5565 | 3381 | 1406 | 1575 |
| *Double[200]* | 1707 | 6348 | 3864 | 1606 | 1800 |
| *Double[25]* | 307 | 876 | 492 | 206 | 225 |
| *Double[5]* | 147 | 250 | 106 | 46 | 45 |
| *Double[50]* | 507 | 1659 | 975 | 406 | 450 |
| *Double[75]* | 707 | 2439 | 1455 | 606 | 675 |
| *Double* | 79 | 93 | 24 | 12 | 9 |
| *Integer[10]* | 144 | 318 | 120 | 56 | 85 |
| *Integer[100]* | 504 | 2386 | 1108 | 506 | 839 |
| *Integer[125]* | 604 | 2961 | 1383 | 631 | 1052 |
| *Integer[150]* | 704 | 3534 | 1656 | 756 | 1265 |

| | | | | | |
|---|---|---|---|---|---|
| *Integer[175]* | 804 | 4110 | 1932 | 881 | 1470 |
| *Integer[200]* | 904 | 4685 | 2207 | 1006 | 1686 |
| *Integer[25]* | 204 | 663 | 285 | 131 | 212 |
| *Integer[5]* | 124 | 203 | 65 | 31 | 43 |
| *Integer[50]* | 304 | 1236 | 558 | 256 | 419 |
| *Integer[75]* | 404 | 1811 | 833 | 381 | 629 |
| *Integer* | 72 | 82 | 16 | 9 | 9 |
| *Object[10]* | 353 | 314 | 180 | 96 | 104 |
| *Object* | 187 | 103 | 22 | 13 | 11 |
| *String[10]* | 531 | 584 | 400 | 386 | 380 |
| *String[100]* | 4041 | 4994 | 3910 | 3806 | 3800 |
| *String[125]* | 5016 | 6219 | 4885 | 4756 | 4750 |
| *String[150]* | 5991 | 7444 | 5860 | 5706 | 5700 |
| *String[175]* | 6966 | 8669 | 6835 | 6656 | 6650 |
| *String[200]* | 7941 | 9894 | 7810 | 7606 | 7600 |
| *String[25]* | 1116 | 1319 | 985 | 956 | 950 |
| *String[5]* | 336 | 339 | 205 | 196 | 190 |
| *String[50]* | 2091 | 2544 | 1960 | 1906 | 1900 |
| *String[75]* | 3066 | 3769 | 2935 | 2856 | 2850 |
| *String* | 137 | 117 | 50 | 42 | 38 |
| *Test* | 502 | 574 | 390 | 162 | 152 |

# Literaturverzeichnis

[1] *POGOProtos*. URL `https://github.com/AeonLucid/POGOProtos`.

[2] *Java Remote Method Invocation - Distributed Computing for Java*. URL `http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html`.

[3] Ernest Adams. *Fundamentals of Game Design*. New Riders, Berkeley, 2nd Edition, 2010.

[4] *ActionScript 3.0 Language Specification*. Adobe Systems Incorporated, 3 Edition, 2006.

[5] Adobe Systems Software Ireland Ltd. *Flash Player*. URL `https://get.adobe.com/flashplayer`.

[6] Thor Alexander. *Massively Multiplayer Game Development*. Charles River Media, Inc., Hingham, Massachusetts, Februar 2003.

[7] Pedro De Almeida und Abel Gomes. *Advanced Strategic Browser-based Massive Multiplayer Online Game: A Game Suggestion*.

[8] Sebastian Apel. *Browserbasierte Echtzeit Simulation "Evolution"*. Studienarbeit, Friedrich-Schiller-Universität Jena, Jena, Germany, 2008.

[9] Grenville Armitage, Mark Claypool, und Philip Branch. *Networking and Online Games*. Wiley and Sons, 2006.

[10] Tim Berners-Lee, Roy Fielding, und Larry. Masinter. *Uniform Resource Identifiers (URI): Generic Syntax*. RFC 2396, Information Sciences Institute, 1998. URL `https://tools.ietf.org/html/rfc2396`.

[11] Eric Bethke. *Game Development and Production*. Wordware Publishing, 2003.

[12] Ashwin R. Bharambe, Jeff Pang, und Srinivasan Seshan. *Colyseus: A Distributed Architecture for Interactive Multiplayer Games*. In *NSDI '06: 3rd Symposium on Network Design and Implementation*, San Jose, California, USA, Mai 2006.

[13] Bigpoint S.à.r.l. and Co, SCS. *DarkOrbit*. URL `http://www.darkorbit.com`.

[14] Jonathan Blow. *Game Development: Harder Than You Think*. *Queue*, 1 (10):29–37, Juli 2004.

[15] Sladjan Bogojevic und Mohsen Kazemzadeh. *The Architecture of Massive Multiplayer Online Games*. Master's thesis, Lund Institute of Technology, Lund University, Lund, September 2003.

[16] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, und François Yergeau. *Extensible Markup Language (XML) 1.0*. W3C, 5 Edition, November 2008. URL `http://www.w3.org/TR/REC-xml/#sec-documents`.

[17] Bundesministerium für Wirtschaft und Energie. *EXIST*. URL `http://www.exist.de/`.

[18] Ed Burns und Roger Kitain. *JavaServer Faces Specification*. Sun Microsystems, Inc., Juni 2009.

[19] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, und Michael Stal. *A System of Patterns, Pattern - Oriented Software Architecture*. Wiley, 1 Edition, 1996.

[20] Nicolas Chen. *Convention over Configuration*. URL `http://softwareengineering.vazexqi.com/files/convention_over_configuration.pdf`.

[21] Qu Chen, Artur Galiullin, und Jeffrey Woo. *A Reference Architecture for Multiplayer Online Games*. 2013. URL `http://www.docstoc.com/docs/147222972/MMORPG_architecture`.

[22] Shigeru Chiba. *Javassist*, Juni 2015. URL http://jboss-javassist.github.io/javassist/.

[23] Mark Claypool und Robert W. Lindeman. *Game Development Timeline*, 2008. URL http://web.cs.wpi.edu/~imgd1001/a08/slides/imgd1001_04_GameDevTimeline.pdf.

[24] George Coulouris, Jean Dollimore, und Tim Kindberg. *Verteilte Systeme : Konzepte und Design*. Pearson Studium, München, 3 Edition, 2003.

[25] Dave Crane und Phil McCarthy. *What Are Comet and Reverse Ajax?*, pages 1–9. Apress, Berkeley, CA, 2009. ISBN 978-1-4302-0864-8. doi: 10.1007/978-1-4302-0864-8_1.

[26] Douglas Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627, The Internet Society, 2006. URL http://tools.ietf.org/html/rfc4627.

[27] Linda DeMichiel. *Enterprise JavaBeans Specification, Version 2.1*. Sun Microystems, Inc., November 2003.

[28] Linda DeMichiel und Michael Keith. *JSR 220: Enterprise JavaBeans,Version 3.0*. Sun Microystems, Inc., November 2006.

[29] *ECMAScript Language Specification*. Ecma International, 5.1 Edition, Juni 2011.

[30] Electrotank. *EUP Brochure Web*, . URL http://www.electrotank.com/images/stories/technology/EUP_Brochure_web_110923b.pdf.

[31] Electrotank. *ES5 Manual*, . URL http://www.electrotank.com/docs/es5/manual/.

[32] Elektrotank. *Elektrotank*. URL http://www.elektrotank.com.

[33] Epic Games. *Unreal Engine*. URL https://www.unrealengine.com/.

[34] Exit Games. *Photon Server*. URL https://www.photonengine.com.

[35] Exit Games. *Photon Server Manual*, 2015. URL https://doc.photonengine.com/en/onpremise/current/getting-started/photon-server-intro.

[36] Ian Fette und Alexey Melnikov. *The WebSocket Protocol*. RFC 6455, Internet Engineering Task Force, Dezember 2011. URL https://tools.ietf.org/html/rfc6455.

[37] Stefan Fiedler, Michael Wallner, und Michael Weber. *A Communication Architecture for Massive Multiplayer Games*. In *Proceedings of the 1st Workshop on Network and System Support for Games*, NetGames '02, pages 14–22, New York, NY, USA, 2002. ACM. ISBN 1-58113-493-2. doi: 10.1145/566500.566503.

[38] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, und Tim Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616, The Internet Society, 1999. URL https://tools.ietf.org/html/rfc2616.

[39] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[40] Erich Gamma, Richard Helm, Ralph Johnson, und John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 Edition, 1994.

[41] Google, Inc. *Developer Guide*. URL https://developers.google.com/protocol-buffers/docs/overview.

[42] gotoAndPlay(). *SmartFoxServer*. URL http://www.smartfoxserver.com.

[43] gotoAndPlay(). *SmartFoxServer 2X Zones and Rooms Architecture*, 2011. URL http://docs2x.smartfoxserver.com/Overview/zones-room-architecture.

[44] gotoAndPlay(). *SmartFoxServer 2X platform stack*, 2011. URL http://docs2x.smartfoxserver.com/Overview/sfs2x-platform-stack.

[45] gotoAndPlay(). *The SmartFoxServer 2X client-server protocol*, 2011. URL `http://docs2x.smartfoxserver.com/Overview/sfs2x-protocol`.

[46] Jason Gregory. *Game Engine Architecture*. A K Peters/CRC Press, Boca Raton, 2009.

[47] Elliotte Rusty Harold. *XML Bible*. IDG Books Worldwide, Inc., 1999.

[48] Huizinga. *Homo ludens*.

[49] ISO/IEC JTC 1. *ISO 7498-1: Information technology - Open Systems Interconnection - Basic Reference Model*, 1 Edition, 1994.

[50] Gavin King. *JSR-299: Contexts and Dependency Injection for the Java EE platform*. Sun Microystems, Inc., Dezember 2009.

[51] Little Workshop. *Browserquest*. URL `http://browserquest.mozilla.org`.

[52] Microsoft. *Age of Empires*. URL `https://www.ageofempires.com`.

[53] Mojang. *Minecraft*. URL `https://minecraft.net`.

[54] *BSON Specification Version 1.1*. MongoDB, Inc. URL `http://bsonspec.org/spec.html`.

[55] Rajiv Mordani. *Java Servlet Specification*. Sun Microystems, Inc., Dezember 2009.

[56] Andrew Mulholland und Teijo Hakala. *Developer's Guide to Multiplayer Games*. Wordware Publishing, 2002.

[57] Node.js Foundation. *Node.js*. URL `https://nodejs.org`.

[58] Oracle. *Java Object Serialization Specification*, 6 Edition, 2010.

[59] *Introduction to Java Platform, Enterprise Edition 7*. Oracle, Juni 2013.

[60] *Java Management Extensions (JMX)*. Oracle, 4 Edition, September 2013.

[61] Santiago Pericas-Geertsen und Marek Potociar. *JAX-RS: Java API for RESTful Web Services*. Sun Microystems, Inc., Mai 2013.

[62] Jon Postel. *User Datagram Protocol*. RFC 768, Information Sciences Institute, 1980. URL `https://tools.ietf.org/html/rfc768`.

[63] Jon Postel. *Transmission Control Protocol*. RFC 793, Information Sciences Institute, 1981. URL `https://tools.ietf.org/html/rfc793`.

[64] Jon Postel. *Internet Protocol*. RFC 791, Information Sciences Institute, 1981. URL `https://tools.ietf.org/html/rfc791`.

[65] Re-Logic. *Terraria*. URL `http://terraria.org`.

[66] Mark Roth und Eduardo Pelegrí-Llopart. *JavaServer Pages Specification*. Sun Microystems, Inc., November 2003.

[67] Simlity. *Viechers*. URL `http://viechers.de`.

[68] Jouni Smed und Harri Hakonen. *Algorithms and Networking for Computer Games*. John Wiley & Sons, 2006.

[69] Gernot Starke und Peter Hruschka. *arc42*. URL `http://www.arc42.de`.

[70] StrongLoop. *Express*. URL `http://expressjs.com`.

[71] *Java Platform, Enterprise Edition (Java EE) Specification, v6*. Sun Microystems, Inc., Dezember 2009.

[72] Travian Games GmbH. *Travian*. URL `http://travian.de`.

[73] Mark Twain. *The Adventures of Tom Sawyer*.

[74] Unity Technologies. *Unity Engine*. URL `https://unity3d.com`.

[75] USC Center for Systems and Software Engineering. *Unified Code Count*. URL `http://sunset.usc.edu/ucc_wp`.

[76] *COCOMO II - Model Definition Manual*. USC Center for Systems and Software Engineering, 2000.

[77] Juha-Matti Vanhatupa. *Browser Games for Online Communities*. *International Journal of Wireless & Mobile Networks*, 2(3):39–47, 2010.

[78] *SOAP Version 1.2 Part 0: Primer (Second Edition).* W3C, April 2007. URL `http://www.w3.org/TR/2007/REC-soap12-part0-20070427/`.

[79] Bian Wu, Alf Inge Wang, und Yuanyuan Zhang. *Experiences from Implementing an Educational MMORPG.* In *2nd International IEEE Consumer Electronics Society's Games Innovations Conference*, 2010.

[80] Stefan Zörner. *Software Architekturen Dokumentieren und Kommunizieren.* Hanser, München, 2012.

# Ehrenwörtliche Erklärung

Hiermit erkläre ich,

- dass mir die Promotionsordnung der Fakultät bekannt ist,
- dass ich die Dissertation selbst angefertigt habe, keine Textabschnitte oder Ergebnisse eines Dritten oder eigenen Prüfungsarbeiten ohne Kennzeichnung übernommen und alle von mir benutzten Hilfsmittel, persönliche Mitteilungen und Quellen in meiner Arbeit angegeben habe,
- dass ich die Hilfe eines Promotionsberaters nicht in Anspruch genommen habe und daß Dritte weder unmittelbar noch mittelbar geldwerte Leistungen von mir für Arbeiten erhalten haben, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen,
- dass ich die Dissertation noch nicht als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht habe.

Bei der Auswahl und Auswertung des Materials sowie bei der Herstellung des Manuskripts haben mich folgende Personen unterstützt:

.....

Ich habe die gleiche, eine in wesentlichen Teilen ähnliche bzw. eine andere Abhandlung bereits bei einer anderen Hochschule als Dissertation eingereicht:
Ja / Nein (Zutreffendes unterstreichen)
Wenn Ja, Name der Hochschule: .....
Ergebnis: .....

..... .....
Ort, Datum Unterschrift