wirtschafts
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Fakultät für
Wirtschafts-
wissenschaften
Business School

Auswahl eines Frameworks für Multiagentensysteme zum Einsatz in Forschung und Lehre

Stefan Selle

Professor für Wirtschaftsinformatik
Fakultät für Wirtschaftswissenschaften
Hochschule für Technik und Wirtschaft des Saarlandes

Saarbrücken, 06.01.2018

Kurzfassung

Software-Agenten sind Computerprogramme, die stellvertretend für Menschen und weitgehend autonom Aufgaben übernehmen und durchführen können. Beispielsweise sorgen Assistenzsysteme im Automobil für ein sicheres und komfortables Fahren. Digitale Assistenten in Form von Smartphone-Apps oder intelligente Lautsprecher helfen bei alltäglichen Problemen. Maschinen und Roboter unterstützen in der Industrie bei der Herstellung von Produkten. Werden Software-Agenten zusammengeschlossen, um eine komplexe Aufgabe zu bewältigen, dann spricht man von einem Multiagentensystem (MAS). Insbesondere in den Bereichen *Internet of Things*, Industrie 4.0, autonomes Fahren und *Smart Home* werden Multiagentensysteme zukünftig eine wichtige Rolle spielen.

Diese Arbeit vermittelt einerseits die theoretischen Grundlagen von Multiagentensystemen und stellt andererseits ein Framework vor, mit dem sich MAS-Anwendungen implementieren lassen. Die Auswahl eines geeigneten Frameworks erfolgt dabei in zwei Schritten. Zunächst wird eine intensive Recherche durchgeführt und bereits diejenigen Frameworks gefiltert, die gängige Standards einhalten. Im zweiten Schritt wird dann eine Nutzwertanalyse zur Bewertung und zum Vergleich dieser Frameworks durchgeführt. Die Präsentation einer Beispielanwendung zeigt schließlich auf, wie sich das Framework verwenden lässt. Zukünftig soll dieses MAS-Framework in Forschung und Lehre an der htw saar eingesetzt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AAL	Ambient Assisted Living
ABM	Agentenbasierte Modellierung
ABS	Antiblockiersystem
ACC	Agent Communication Channel
ACL	Agent Communication Language
AMS	Agent Management System
AMUSE	Agent-based Multi-User Social Environment
API	Application Programming Interface
B2B	Business-to-Business
B2C	Business-to-Consumer
BDI	Belief-Desire-Interaction
BT	British Telecom
CPS	Cyber-physisches System
DF	Directory Facilitator
ESP	Elektronisches Stabilitätsprogramm
EVA	Eingabe-Verarbeitung-Ausgabe
FIFO	First In - First Out
FIPA	Foundation for Intelligent Physical Agents
GPL	GNU General Public License
GUI	Graphical User Interface
GUID	Global Unique Identifier
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
JADE	Java Agent Development Framework
JAMEL	Java Agent-based MacroEconomic Laboratory
JVM	Java Virtual Machine

KI Künstliche Intelligenz

LGPL GNU Lesser General Public License

M2M Maschine-zu-Maschine

MAS Multiagentensystem

MASON Multi-Agent Simulator Of Neighborhoods

MTS Message Transport System

MTP Message Transport Protocol

MVC Model View Controller

OASIS Optimal Aircraft Sequencing using Intelligent Scheduling

OWL Web Ontology Language

RDF Resource Description Framework

REPAST Recursive Porous Agent Simulation Toolkit

SeSAm Shell for Simulated Agent Systems

TCO Total Cost of Ownership

VKI Verteilte Künstliche Intelligenz

VSIS Verteilte Systeme und Informationssysteme

WADE Workflows and Agents Development Environment

XML Extensible Markup Language

1 Einleitung

Vielen Menschen ist der Begriff Software-Agent nicht bekannt, selbst wenn sie mit diesen bereits in Berührung gekommen sind. Wenn man beispielsweise eine Service-Hotline eines Dienstleistungsunternehmens anruft, dann meldet sich fast immer zunächst nur ein Computerprogramm, das die Anfrage entgegen nimmt und je nach Absicht des Anrufers die Weiterleitung zu einem Service-Mitarbeiter gezielt steuert. In diesen Fällen ist es offensichtlich, dass der Kommunikationspartner eine Maschine ist. Im weitesten Sinne könnte man hier von einem Software-Agenten sprechen, der stellvertretend für einen Menschen eine Dienstleistung erbringt. Nicht ganz so offensichtlich ist die Interaktion mit *Chatbots*. Der Begriff setzt sich zusammen aus den englischen Wörtern *Chat* für Plaudern und *Robot* für Roboter bzw. Maschine. In sozialen Netzwerken gibt es Nutzerprofile, die nicht von Menschen sondern von Maschinen gepflegt werden. Diese Chatbots können textbasierte Dialoge mit anderen Teilnehmern führen. Negative Schlagzeilen haben Chatbots vor allem in Form von sogenannten Propaganda-Bots während des letzten US-Präsidentenwahlkampfs erlangt. Mit Hilfe von Millionen dieser künstlichen *Follower* wurden Twitter-Meldungen der Kandidaten Donald Trump und Hillary Clinton weiterverteilt. Dadurch können generell die politische Willensbildung erschwert und Wähler gezielt manipuliert werden [Ron16]. Vorbild aller Chatbots ist ELIZA, ein Computerprogramm, das bereits 1966 von Joseph Weizenbaum entwickelt wurde, um die Kommunikation zwischen Mensch und Maschine zu demonstrieren [Wei66]. ELIZA agiert dabei als virtuelle Psychotherapeutin. Sie benutzt ein strukturiertes Wörterbuch und eine Sammlung von typischen Phrasen zu speziellen Themen. Ähnlich wie bei einer realen Psychotherapeutin stellt ELIZA Fragen oder gibt Aufforderungen, z.B. „Erzählen Sie mir mehr zu dem Thema!“. Dem Gesprächspartner wird dabei nur vorgetäuscht, die Aussagen tatsächlich zu verstehen oder gar Verständnis für seine Probleme zu haben. ELIZA und die aktuellen Chatbots würden den Turing-Test[1] allerdings nicht bestehen. Im Zeitalter des *Mobile Computing* ist mittels Smartphone auch eine Kommunikation per natürlicher Sprache zwischen Mensch und Maschine möglich. Diese Systeme sind mit einer Spracherkennungssoftware ausgestattet und liefern die Antwort per Sprachausgabe an den Benutzer zurück. Bekannte digitale Assistenten sind bspw. Apples Siri, Amazons Alexa, Microsofts Cortana oder Googles Assistent. Diese Software-Agenten liefern ggf. sinnvolle Antworten auf unsere alltäglichen Fragen, können aber auch bereits Produkte im Internet bestellen oder vernetzte Geräte in unserem Haushalt steuern. Man kann also zweierlei feststellen: Die Agenten werden immer intelligenter und die Interaktionen zwischen Menschen und Maschinen nehmen zu.

Wenn man nun mehrere Software-Agenten zusammenschließt, bekommt man ein MAS, also ein Multiagentensystem. In einem solchen verteilten System können die Agenten auch untereinander kommunizieren. Maschine-zu-Maschine (M2M)-Kommunikation ist also

[1] Der Turing-Test ist benannt nach dem englischen Mathematiker Alan Turing (23.06.1912 - 07.06.1954). In einem Gedankenexperiment führt ein Mensch per Tastatur und Monitor ein Gespräch mit einem Gegenüber, ohne diesen zu sehen oder zu hören. Der Gegenüber ist entweder ein Mensch oder eine Maschine. Der Turing-Test gilt für eine Maschine dann als bestanden, wenn nach einer intensiven Befragung nicht zweifelsfrei entschieden werden kann, dass es sich bei dem Gegenüber um eine solche handelt. In diesem Fall würde man der Maschine also menschenähnliches Denkvermögen und damit Intelligenz attestieren.

ein typischer Bestandteil solcher Systeme. Dieses Merkmal ist auch eine wesentliche Voraussetzung, damit die Agenten gemeinsam eine komplexe Aufgabe lösen, die sie alleine nicht bewältigen könnten. Ganz ähnlich funktionieren Unternehmen bzw. Organisationen, allerdings werden diese normalerweise zentral gesteuert. In einem MAS existiert dagegen keine zentrale Steuereinheit, sondern die Agenten agieren weitgehend autonom, weshalb das System sehr robust ist. Das Internet basiert auf einem ähnlichen Konzept und wurde so entwickelt, dass es ausfallsicher ist[2]. Im *Internet of Things (IoT)* werden vor allem Dinge, also Maschinen, miteinander vernetzt. Die technische Voraussetzung ist dabei der Einsatz des *Internet Protocol (IP)* in der Version 6, welches das veraltete IPv4 zunehmend ablöst. Nur so ist eine hoch dimensionale Vernetzung von Geräten möglich. Daraus ergeben sich diverse Anwendungsfelder, die im Folgenden skizziert werden.

Das *Smart Home* ist bereits kurz angesprochen wurden. Hierbei lassen sich Geräte im Haushalt vernetzen und steuern: Heizung, Rollläden, Licht, Radio, TV, Waschmaschine, Kühlschrank, Alarmanlage, Backofen usw. Durch Prozessautomatisierung lässt sich der Wohnkomfort steigern, bspw. indem die Rollläden je nach Lichtsituation bzw. Tageszeit automatisch gesteuert werden. Ein anderes Beispielszenario handelt von der intelligenten Waschmaschine. Diese entscheidet selbstständig, wann sie wäscht, vorausgesetzt sie ist mit schmutziger Wäsche und Waschmittel gefüllt. Die Maschine berücksichtigt dabei die Daten der Photovoltaikanlage auf dem Dach, des Solarstromspeichers im Keller sowie Wetterinformationen aus dem Internet, um einen möglichst effizienten Waschvorgang durchzuführen. Die von der Photovoltaikanlage überschüssig produzierte Energie kann ggf. auch in das Stromnetz eingespeist werden. Allgemein bilden diese dezentralen Energielieferanten das sogenannte *Smart Grid*. Dieses kann ebenfalls als Multiagentensystem betrachtet werden. Elektrischer Strom ist eine sehr spezielle Ware, weil sich diese nur schlecht oder mit erheblichen Verlusten speichern lässt und deshalb normalerweise nach der Erzeugung direkt verbraucht werden sollte. Wenn der Haushalt neben dem Solarstromspeicher im Keller aber auch noch über ein Elektroautomobil verfügt, dann kann dieses, genauer gesagt dessen Akkus, ebenfalls als Energiespeicher fungieren.

Moderne Automobile fahren nicht nur mit Strom als Kraftstoff, sondern sie sind auch rollende Computersysteme. Jedes Auto für sich betrachtet ist bereits ein Multiagentensystem. Es ist mit einer Vielzahl von Softwarekomponenten ausgestattet, insbes. mit Assistenzsystemen, um Sicherheit und Fahrkomfort zu erhöhen. Dazu gehören bspw. Antiblockiersystem (ABS), Elektronisches Stabilitätsprogramm (ESP), Bremsassistent, Reifendruckkontrollsystem, Abstandsregeltempomat, Spurwechselassistent, Adaptiver Fernlichtassistent, Einparkhilfe, Navigationssystem usw. Diese Systeme benutzen verschiedene Sensoren, um ihre Umwelt wahrzunehmen. Jedes dieser Systeme kann als Software-Agent betrachtet werden. Deren Zusammenspiel ist dann eine Form eines Multiagentensystems. Zukünftig werden Automobile immer intelligenter werden und selbstständiger agieren, bis hin zum vollständig autonomen Fahren. Eine Vernetzung der Fahrzeuge untereinander und mit der Verkehrsinfrastruktur (Ampeln, Verkehrszeichen) wird bereits entwickelt [Con07]. Die sogenannte Car2X-Kommunikation soll helfen, die Verkehrssicherheit zu erhöhen und den Verkehrsfluss optimal zu steuern. In einem solchen Multiagentensystem wird dann jedes Fahrzeug und jede Ampel als ein Agent betrachtet. Im Bereich der Luftfahrt wurde bereits in den 1990er Jahren das Multiagentensystem Optimal

[2] Eine Hundertprozentige Sicherheit ist prinzipiell nicht gegeben, aber ein kompletter Ausfall wäre sehr unwahrscheinlich. Denn falls ein Netzknoten ausfällt, dann springt normalerweise ein anderer für ihn ein, sodass nicht das komplette Netz ausfällt, sondern nur ein anderer Weg benutzt wird, um die Daten vom Absender zum Empfänger zu transportieren (*Routing*).

Aircraft Sequencing using Intelligent Scheduling (OASIS) zur Luftverkehrskontrolle am Flughafen von Sydney in Australien entwickelt [LL92].

Neben dem Anwendungsbereich Transport & Logistik sind Multiagentensysteme auch in der Produktion zu finden. ARCHON ist bspw. ein MAS, welches zur Produktionsplanung und -steuerung eingesetzt wird [Wit92]. Ein Trend in der Produktion ist die immer stärker werdende Individualisierung der Produkte. D.h. die Produktvielfalt nimmt zu, im Extremfall wieder zu Losgröße Eins. Um diesen Kundenanforderungen gerecht zu werden und flexibel produzieren zu können, wird ein Cyber-physisches System (CPS) benötigt. Dies ist ein Netzwerk aus technischen Komponenten, sogenannten *Embedded Systems*, dessen Charakteristik die dezentrale Steuerung ist. Beispielsweise kann das zu fertigende Produkt als Werkstück mit Hilfe eines integrierten RFID-Chips die Informationen zu dessen Bearbeitung selbst zur Maschine bringen. Eine solche *Smart Factory* kann mit einem Multiagentensystem beschrieben werden. Forschungs- und Industrieprojekte in diesem Bereich werden auch mit dem Schlagwort Industrie 4.0 versehen und gehören damit zur Hightech-Strategie der Bundesrepublik Deutschland.

Ein anderer Bereich, in dem Agenten zum Einsatz kommen, ist der Internet-Handel. Shopping-Bots durchsuchen das Internet nach Händlern zu einem bestimmten Produkt und liefern als Ergebnis eine nach Präferenzen (Preis, Bewertung usw.) sortierte Liste an den Nutzer zurück. Um komplexe Bestellprozesse abzuwickeln, können sogar agentenbasierte Marktplätze in Form eines Multiagentensystem aufgebaut werden, sowohl als Business-to-Business (B2B)- als auch als Business-to-Consumer (B2C)-Plattform.

In allen diesen Anwendungsbeispielen werden aufgrund der Kommunikation zwischen den Maschinen und ggf. auch durch die Interaktion mit den Benutzern sehr viele Daten generiert. Die auf diese Art generierten und gesammelten Daten werden von den Systemen wiederum genutzt, um die Prozesse zu optimieren, in denen sie eingesetzt werden. Man spricht daher auch von selbstlernenden Systemen. Somit stehen Themen wie Künstliche Intelligenz (KI), *Big Data*, IT-Sicherheit und Datenschutz ebenfalls im Zusammenhang mit Multiagentensystemen.

Das Ziel dieser Arbeit ist das Suchen und Finden eines geeigneten Frameworks für Multiagentensysteme, das möglichst universell bei der Entwicklung und Simulation der beschriebenen Anwendungen eingesetzt werden kann. Ein systematisches Vorgehen ist notwendig, um dieses Ziel zu erreichen, welches sich daher auch im strukturellen Aufbau dieser Arbeit widerspiegelt. In Kap. 2 werden zunächst die theoretischen Grundlagen gelegt, die wesentlichen Begriffe definiert und Zusammenhänge zwischen diesen dargestellt. Insbesondere werden hier auch gängige Standards im Zusammenhang mit Agenten und MAS beschrieben. In Kap. 3 werden dann die MAS-Frameworks vorgestellt, welche diese Standards verwenden und notwendigen Anforderungen genügen. Mit Hilfe einer Nutzwertanalyse werden in Kap. 4 diese gefilterten MAS-Frameworks bewertet und miteinander verglichen. Hierzu werden ausgewählte Merkmale der Frameworks gemäß weiterer, hinreichender Anforderungen gewichtet. Das Framework mit dem höchsten Nutzwert wird in Kap. 5 dann näher vorgestellt. In diesem Kapitel wird auch eine Beispielanwendung in Form eines Buchhandels präsentiert und die Installation, Konfiguration und der Betrieb des Frameworks dargestellt. Daraufhin wird ein kurzes Fazit bezüglich dieser ersten Erfahrungen und betreffend des ursprünglichen Ziels gezogen. Im letzten Kapitel wird die Arbeit schließlich zusammengefasst und ein Ausblick auf die weiteren Entwicklungen gegeben.

2 Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen zum Thema Multiagentensystem (MAS) behandelt. Zunächst wird allgemein der Begriff des Systems eingeführt und definiert, dann werden Merkmale von Systemen beschrieben und verschiedene Systeme voneinander abgegrenzt. Anschließend wird der Begriff des Agenten definiert. Das Besondere dabei ist, dass ein Agent bereits selbst ein System darstellt. Merkmale und Klassen von Agenten werden vorgestellt, insbesondere wird das Konzept des Belief-Desire-Interaction (BDI)-Agenten erläutert, ein Standard und die Basis für einen intelligenten Agenten. Danach wird das Kernthema MAS behandelt und auf die Merkmale eingegangen. Abschließend wird die Referenzarchitektur der Foundation for Intelligent Physical Agents (FIPA) präsentiert.

2.1 Systeme

Systeme helfen uns einerseits dabei, die Welt, in der wir leben, zu erklären, und andererseits unterstützen sie uns dabei, unsere Aufgaben in dieser Welt zu erfüllen. Deshalb begegnen uns zahlreiche Systeme in der Natur und den Wissenschaften, z.B. Sonnensystem, Atomsystem, Periodensystem, Ökosystem, Zahlensystem, Softwaresystem usw. Das Wort *System* muss aber nicht notwendigerweise selbst im zugrundeliegenden Begriff vorkommen. Beispielsweise sind das Alphabet, Maschinen und Unternehmen ebenfalls Systeme. Ursprünglich kommt das Wort System aus dem Griechischen (*sýstēma*) und bedeutet [Dud17]

> *« aus mehreren Teilen zusammengesetztes und gegliedertes Ganzes. »*

Allgemein kann der Begriff System wie folgt definiert und erklärt werden [FG17]:

> *« Ein System ist eine Menge von geordneten Elementen mit Eigenschaften, die durch Relationen verknüpft sind. Die Menge der Relationen zwischen den Elementen eines Systems ist seine Struktur. Unter Element versteht man einen Bestandteil eines Systems, der innerhalb dieser Gesamtheit nicht weiter zerlegt werden kann. Die Ordnung bzw. die Struktur der Elemente eines Systems ist im Sinn der Systemtheorie seine Organisation. Die Begriffe der Organisation und der Struktur sind also identisch. »*

Die Elemente eines Systems werden auch Entitäten genannt; dies sind unterscheidbare Dinge bzw. Objekte. Zwischen den Elementen (Entitäten) bestehen Beziehungen (Relationen), die sich in Anzahl und Art unterscheiden. Hierdurch manifestiert sich eine Systemgrenze, durch die sich das System von seiner Umwelt abgrenzt.

Man unterscheidet **offene** und **geschlossene Systeme**. Ein offenes System enthält mindestens ein Randelement, d.h. ein Element, das zu Elementen anderer Systeme in Beziehung steht. Ein geschlossenes System enthält keinerlei solcher Randelemente. Reale Systeme sind immer offene Systeme, näherungsweise lassen sie sich jedoch als geschlossene Systeme beschreiben. Systeme lassen sich in Teilsysteme unterteilen, wobei ein Teilsystem ein Randelement des übergeordneten Systems darstellt.

Ein **dynamisches System** (z.B. Ökosystem) zeigt im Gegensatz zu einem **statischen System** (z.B. Alphabet) ein zeitabhängiges Verhalten. Auf der Mikroebene können alle Elemente dauerhaft in Bewegung sein, auf der Makroebene, also von außen betrachtet, kann ein dynamisches System einen stationären Zustand einnehmen. **Stabile Systeme** sind solche dynamischen Systeme, die wieder von selbst einen stationären Gleichgewichtszustand erreichen, wenn sie eine Störung erfahren.

Komplexe Systeme sind vielschichtig und sie bestehen aus vielen, unterschiedlichen Elementen, die sehr viele Beziehungen untereinander haben. Eine einzelne dieser Beziehungen kann auch kompliziert sein. Unternehmen und Softwaresysteme sind Beispiele für komplexe Systeme. Es sind sozio-technische Systeme, die u.a. Mensch-Maschine-Schnittstellen beinhalten. Um die Komplexität in solchen Systemen zu reduzieren, wird oft das sogenannte Teile-und-Herrsche-Prinzip anwendet: Man unterteilt das System zunächst in einzelne Teile und muss dann die Schnittstellen zwischen diesen Teilen beherrschen. Viele komplexe Systeme sind hierarchisch aufgebaut. In Abb. 2.1 ist der Aufbau eines Softwaresystems schematisch dargestellt. Hierarchisch strukturierte Systeme sind durch **Emergenz** gekennzeichnet. Auf der Makroebene bilden sich Eigenschaften heraus, die auf der Mikroebene nicht vorhanden sind. Nach Aristoteles gilt [ArioJ]:

> *« Das Ganze ist mehr als die Summe seiner Teile. »*

Die **Ontologie** beschäftigt sich damit, Systeme formal mit begrifflichen Mitteln zu beschreiben. Die **Modellierung** bildet ein System ab, wobei Methoden zur Abstraktion (z.B. Generalisierung) und Reduktion (z.B. Hierarchien) eingesetzt werden, um mit Komplexität umzugehen. Modelle können beispielsweise grafische Abbildungen aber auch mathematische Formeln sein. Zu einem Modell lassen sich **Simulationen** durchführen, um das Verhalten des zugrundeliegenden Systems zu analysieren. Für mathematische Modelle können solche Systemanalysen mit Hilfe von Computerprogrammen durchgeführt werden.

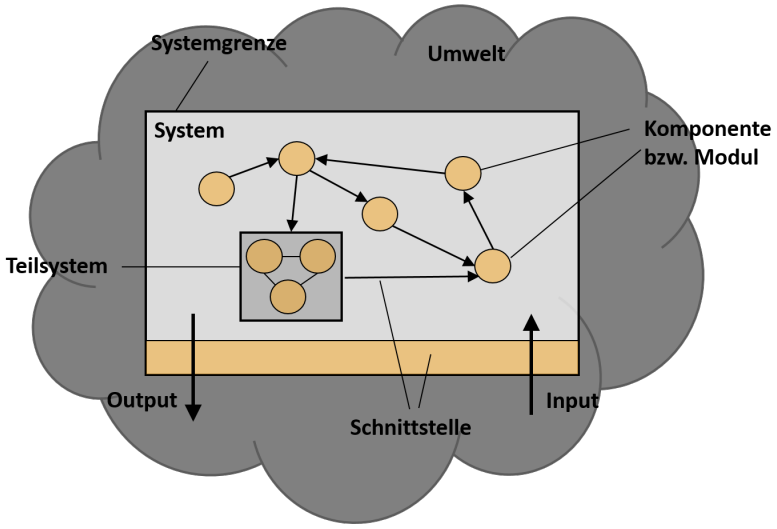

Abbildung 2.1: Schematische Darstellung eines Softwaresystems

2.2 Agenten

Je nach Forschungs- und Anwendungsbereich (z. B. Künstliche Intelligenz, Robotik, Entwicklung verteilter Systeme) wird der Begriff des Agenten unterschiedlich definiert. Eine sehr allgemeine Definition stammt von Russel und Norvig [RN03]:

> *« An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators. »*

In dieser Definition wird insbesondere die Wechselwirkung des Agenten mit seiner Umwelt herausgestellt. Der Agent nimmt dabei die Umwelt über Sensoren wahr. Sein Handeln über Aktoren beeinflusst wiederum die Umwelt (vgl. Abb. 2.2).

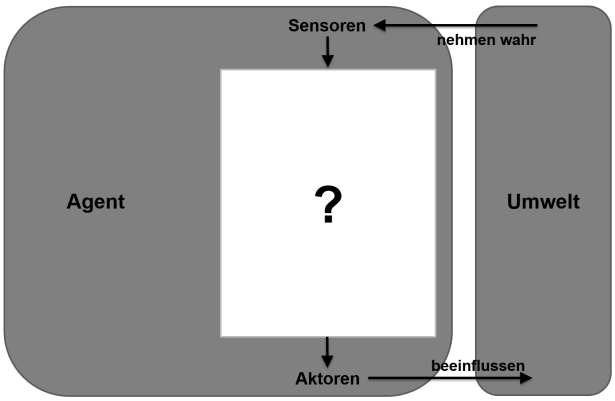

Abbildung 2.2: Ein Agent interagiert mit seiner Umwelt – In Anlehmung an [RN03]

2.2.1 Merkmale von Agenten

Woolridge und Jennings [WJ95] konkretisieren in ihrer Definition einige Aspekte:

> *«An agent is an encapsulated computer system that is situated in some environment, and that is capable of flexible autonomous action in that environment in order to meet its design objectives. »*

Zunächst wird spezifiziert, dass es sich bei dem Agenten selbst um ein System handelt (vgl. Abschnitt 2.1). Ein Computersystem besteht aus materieller Hardware und immaterieller Software. Beispielsweise ist ein Roboter eine Maschine (Hardware), die von einem Computerprogramm (Software) gesteuert wird. Bei einem sogenannten Software-Agenten handelt es sich zwar nur um ein virtuelles Computerprogramm, dieses benötigt aber eine Hardwareumgebung, um ausgeführt zu werden. Auch Software-Agenten können mobil sein. Hierbei wird das Computerprogramm auf eine andere Hardware-Umgebung migriert, d.h. übertragen. Insofern ist **Mobilität** ein optionales Merkmal eines Agenten.

Außerdem erfahren wir aus der Definition, dass das Computersystem gekapselt ist. Unter **Kapselung** des Systems versteht man, dass der Agent eine in sich abgeschlossene Einheit ist. Diese besitzt eine innere Struktur, deren genauer Aufbau aber ggf. verborgen bleibt und auf deren Elemente man keinen direkten Zugriff hat. Stattdessen werden jedoch definierte Schnittstellen zur Verfügung gestellt, um **Interaktionen** mit der Umwelt zu ermöglichen. Letzteres impliziert, dass der Agent über **soziale Fähigkeiten** verfügt, diese auch einsetzt und mit seiner Umwelt **kommuniziert**.

Des Weiteren wird die **Situiertheit** des Agenten in der Umwelt betont. Normalerweise ist der Agent nicht allwissend, sondern er nimmt über seine Sensoren nur einen Teil seiner Umwelt wahr. Begrenztes Wissen und Unsicherheit bestimmen sein Handeln. Er benötigt Fähigkeiten, um seine aktuelle Situation in seinen Entscheidungsfindungsprozess zu integrieren. Insofern ist eine gewisse **Flexibilität** des Agenten erforderlich. Er kann **reaktiv** auf seine Umwelt reagieren oder sich **proaktiv** verhalten. Jedenfalls agiert der Agent eigenständig bzw. eigenverantwortlich ohne direkten Eingriff von außen (**Autonomie**). Schließlich wird der Agent zu einem bestimmten Zweck konstruiert. In seinen Entscheidungsfindungsprozess werden diese Entwurfsziele ebenfalls berücksichtigt. Somit verhält sich der Agent **zielorientiert**.

Weitere optionale Merkmale eines Agenten sind **Lernfähigkeit** und **Rationalität**. Voraussetzung für die Lernfähigkeit ist, dass der Agent in der Lage ist, seine aktuelle Situation und seine bisherigen Aktionen hinsichtlich der erreichten Ergebnisse und der Entwurfsziele zu bewerten. Dann kann er nämlich verschiedene Lernmethoden anwenden, um sein zukünftiges Verhalten bezüglich dieser Ziele zu optimieren. Dies ermöglicht eine Anpassung an sich verändernde Umweltzustände und ist die Basis für **adaptives** Verhalten. Wenn der Agent sich rational verhält, dann wird er diejenigen Aktionen auswählen, von der er erwartet, dass ihm diese langfristig zum gewünschten Ziel bringen. Ein rationaler Agent wird also immer das Richtige tun. Lernfähigkeit und Rationalität sind Eigenschaften von **intelligenten** bzw. **kognitiven** Agenten.

2.2.2 Klassifikation von Agenten

Ausgehend von den Merkmalen (vgl. Abschnitt 2.2.1) lassen sich nach Russel und Norvig [RN03] Agenten hinsichtlich ihres inneren Aufbaus und ihres Verhaltens klassifizieren.

Der **einfache Reflex-Agent** besitzt kein Gedächtnis. Er verfügt über ein Regelwerk, aus dem hervorgeht, welche Aktion durchzuführen ist, falls eine bestimmte Umweltsituation als Bedingung eingetreten ist. Ausgehend von der Wahrnehmung seiner Umwelt wählt er somit gemäß dieses Regelwerks genau die entsprechende Aktion und führt diese aus. In Abb. 2.3 ist der einfache Reflex-Agent schematisch dargestellt. Rechtecke symbolisieren den aktuellen internen Entscheidungsprozess, während die abgerundeten Symbole Hintergrundinformationen darstellen, die in diesen Entscheidungsprozess einfließen.

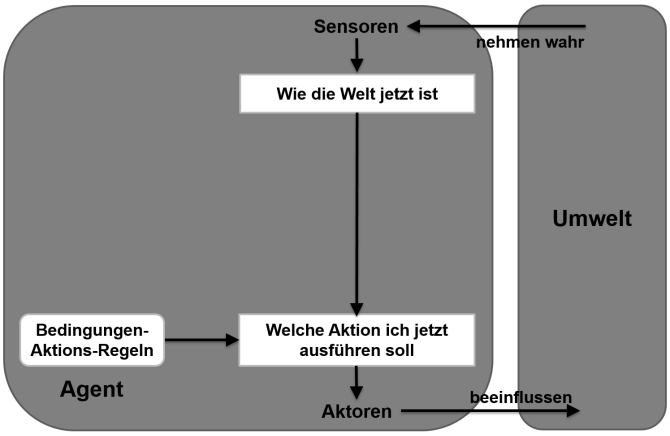

Abbildung 2.3: Schematischer Aufbau des einfachen Reflex-Agenten – in Anlehnung an [RN03]

Gleiche Wahrnehmungen führen immer zu gleichen Aktionen. Solche Verhaltensmuster nennt man Reflexe. Biologisch betrachtet ist ein Reflex eine stereotypische, also automatische und gleichartige, Reaktion eines Organismus auf einen bestimmten Reiz. Agenten, die reflexartig reagieren, sind sehr einfach aber auch sehr effizient, denn die Reaktion erfolgt unmittelbar. Dies setzt jedoch voraus, dass alle möglichen Umweltzustände als Bedingungen und die zugehörigen Aktionen im Regelwerk des Agenten hinterlegt sind. Außerdem muss die Umwelt vollständig beobachtbar sein. Wenn bereits eine dieser beiden Voraussetzungen nicht gegeben ist, dann findet sich der einfache Reflex-Agent nicht mehr in seiner Umwelt zurecht.

Der **modellbasierte Reflex-Agent** ist eine Erweiterung des einfachen Reflex-Agenten und er besitzt, wie der Name schon sagt, ein Modell der Welt. In seinem internen Zustand werden die früheren Aktionen und deren Auswirkungen gespeichert. Weiterhin benötigt er Informationen, wie sich die Umwelt unabhängig von ihm weiterentwickelt. Außerdem muss der Agent noch wissen, wie seine eigenen Aktionen die Welt beeinflussen werden (vgl. Abb. 2.4). Daraus kann er dann Rückschlüsse ziehen und sich besser in einer nur teilweise beobachtbaren Umgebung zurechtzufinden als der einfache Reflex-Agent. Dafür benötigt er jedoch ein komplizierteres Auswahlverfahren zu den im Regelwerk festgelegten Bedingungen und Aktionen.

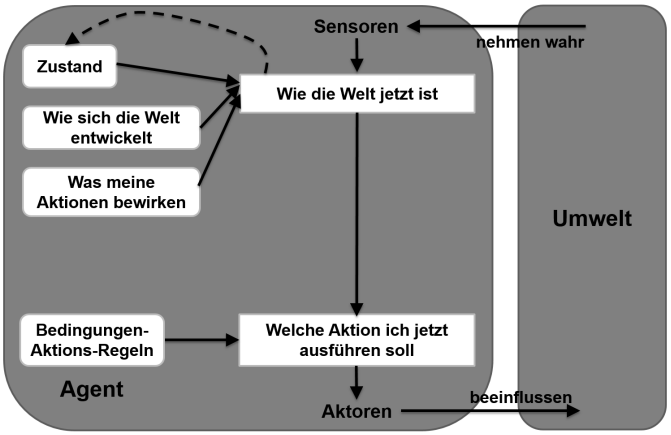

Abbildung 2.4: Schematischer Aufbau des modellbasierten Agenten – in Anlehnung an [RN03]

Der **zielbasierte Agent** verfügt über ein oder mehrere Ziele, die er erreichen möchte (vgl. Abb. 2.5). Diese Ziele lassen sich in kurz-, mittel- und langfristige Ziele einteilen, d.h. es gibt eine festgelegte Zielhierarchie. Kurzfristige Ziele lassen sich ggf. in einem Schritt mittels einer Aktion erreichen. Bei der Verfolgung von langfristigen Zielen muss der Agent dagegen seine Aktionen schrittweise im Voraus planen und immer wieder überprüfen, ob Teilziele bereits erreicht wurden. Zum Planen gehört auch das Suchen von Aktionen, welche den Agenten nach deren Ausführung näher zum Ziel bringen. Diese Art des Entscheidungsfindungsprozesses ist grundsätzlich verschieden zu dem des Reflex-Agenten. Dies bietet sehr viel mehr Flexibilität als ein festes Regelwerk. Allerdings verursachen das Suchen und Planen eine höhere Komplexität und mehr Aufwand, sowohl bei der Modellierung als auch beim Einsatz dieser Typen von Agenten.

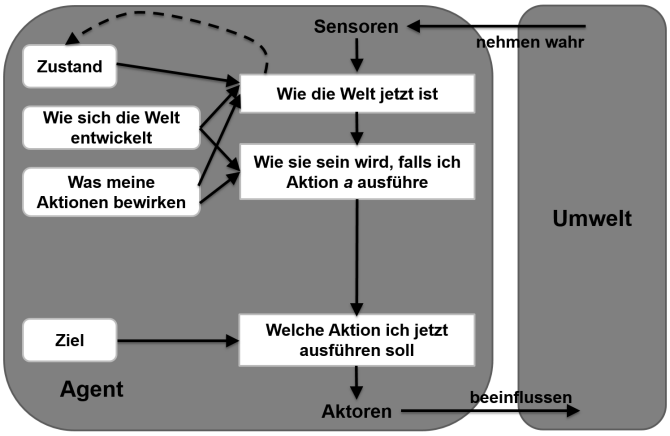

Abbildung 2.5: Schematischer Aufbau des zielbasierten Agenten – in Anlehnung an [RN03]

Der **nutzenbasierte Agent** hat kein eigenes Ziel, sondern verfügt stattdessen über eine Nutzenfunktion, die jedem Zustand einen Nutzenwert zuordnet (vgl. Abb. 2.6). Der Agent wird diese Nutzenfunktion und den damit verbundenen Nutzen maximieren. Hierdurch lassen sich auch Entscheidungen treffen, falls Ziele in Konflikt zueinander stehen. Der nutzenbasierte Agent verhält sich also rational. Die Berechnung der Nutzenfunktion kann allerdings sehr aufwendig sein, wenn sehr viele Zustände betrachtet werden müssen. Außerdem sind möglicherweise nicht alle Informationen vorhanden, um diese zu berechnen, weil der Agent nicht allwissend ist und sich seine Beobachtungen nur auf einen Teil der Umgebung erstrecken.

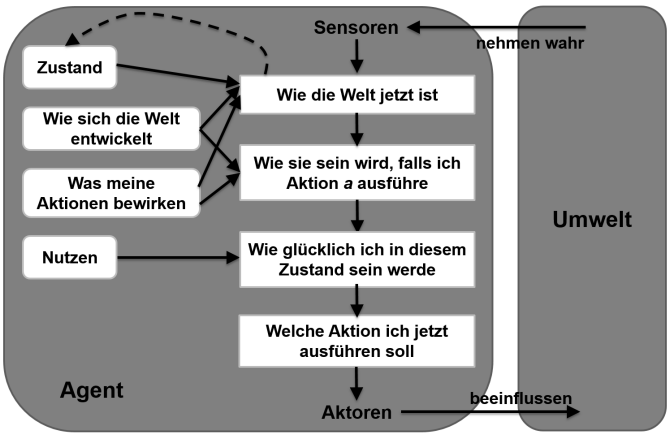

Abbildung 2.6: Schematischer Aufbau des nutzenbasierten Agenten – in Anlehnung an [RN03]

Der **lernende Agent** ist im Kern einer der bisherigen Agenten. Das sogenannte Leistungselement enthält die bisherigen Komponenten (Regelwerk, Modelle, Ziele und Nutzenfunktionen). Der lernende Agent kann diese Komponenten nun verändern und zwar mit Hilfe des Lernelements. Die Kritik-Komponente gibt dabei dem Lernelement Feedback, ob das Verhalten des Leistungselements hinsichtlich der externen Leistungsstandards erfolgreich war oder nicht. Die positive oder negative Rückmeldung kann mit eine Belohnung oder einer Bestrafung verbunden sein. Der leistungsorientierte Agent wird sich

nun zukünftig so verhalten, dass er die neu gelernten, sehr erfolgreichen Aktionen bevorzugt einsetzen wird. Dadurch wird aber möglicherweise Potenzial verschenkt, denn einige der kurzfristig suboptimalen Aktionen könnten auf lange Sicht zu noch besseren Ergebnissen führen. Deshalb wird die Problemgenerator-Komponente benötigt (vgl. Abb. 2.7). Diese generiert neue Aufgaben, die der Agent mittels Experimentieren lösen soll. Die so gemachten neuen Erfahrungen und daraus gewonnenen Erkenntnisse lassen sich dann zukünftig in den Entscheidungsfindungsprozess bei der Auswahl der Aktionen einsetzen. Dieses Wissen verhilft dem Agenten, in noch unbekannten Situationen, erfolgreich zu agieren.

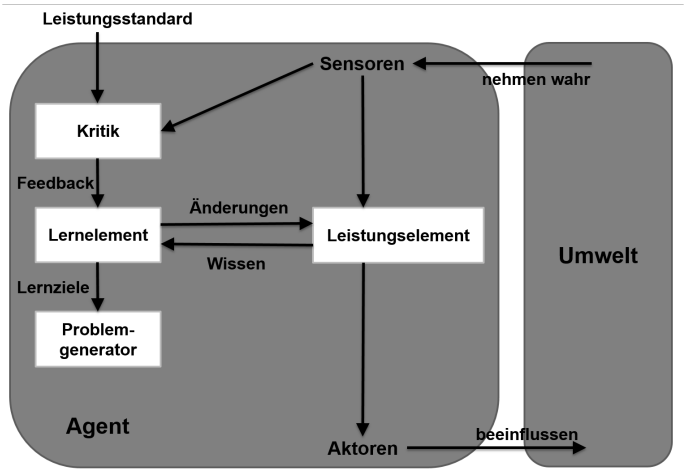

Abbildung 2.7: Schematischer Aufbau des lernenden Agenten – in Anlehnung an [RN03]

Woolridge und Jennings unterscheiden nur drei Klassen von Agenten [WJ95]. Der **reaktive Agent** entspricht dem einfachen Reflex-Agenten. Er besitzt kein internes, symbolisches Modell seiner Umwelt. Somit kann er keine komplexen Schlussfolgerungen ziehen, sondern seine Entscheidungen basieren ausschließlich auf den lokalen Informationen zu seiner Umwelt. Er besitzt keine Pläne und kann sich auch nicht adaptiv verhalten. Dafür ist der reaktive Agent sehr kompakt, robust und effizient.

Der **deliberale Agent** zeichnet sich im Gegensatz zum reaktiven Agenten durch reflektierendes Verhalten aus. Er verfügt über Domänenwissen, also ein Modell der Welt, kann daraus Schlüsse ziehen und seinen internen Zustand anpassen (Adaption). Aufgrund von Ziel- und Planer-Komponenten lassen sich auch rationale Entscheidungsprozesse abbilden. Somit kann ein deliberaler Agent auch als intelligenter bzw. kognitiver Agent bezeichnet werden. Er entspricht also dem lernenden Agenten von Russel und Norvig.

Der **hybride Agent** ist eine Kombination aus einem reaktiven und einem deliberalen Agenten, der die jeweiligen Vorteile auf sich vereint.

2.2.3 BDI-Agent

Der Belief-Desire-Interaction (BDI)-Agent ist ein deliberaler Agent. Es werden drei mentale Komponenten unterschieden, aus die sich sein interner Zustand zusammensetzt [RG95]:

Beliefs	—	Überzeugungen, Annahmen
Desires	—	Wünsche, Motivationen, Ziele
Intentions	—	Intentionen, Absichten, Vorhaben

Ein BDI-Agent besitzt ein internes Modell über die Welt, welches in der Komponente **Beliefs** verarbeitet wird. Dieses Wissen, also die Annahmen über die Welt, kann sich zeitlich ändern und erweitert werden. Das Wissen über die Welt beinhaltet auch das Wissen über andere Agenten (Beziehungswissen) und von anderen Agenten (*Second Hand Knowledge*). Durch Kommunikation mit diesen Agenten kann also Wissen erworben werden. Durch interne Schlussfolgerungen des Agenten lässt sich das vorhandene Wissen ebenfalls erweitern. Die Komponente **Desires** verarbeitet die verschiedenen Handlungsmöglichkeiten des Agenten. Diese können auch widersprüchlich zueinander sein, da sie nicht zwingend ausgeführt werden müssen. Die Komponente **Intentions** verarbeitet die Absichten des Agenten. Es wird immer diejenige Absicht gespeichert, auf die sich ein Agent festgelegt hat (*Commitment*). Das Ziel hinter dieser Absicht wird solange verfolgt, bis es erreicht ist oder verworfen wurde. Der Prozess der Entscheidungsfindung nennt man *Practical Reasoning*. Er lässt sich grob in zwei Schritte unterteilen. Zunächst wird ermittelt, **welche** Ziele der Agent erreichen soll (Effektivität) und anschließend wird bestimmt, **wie** sich diese Ziele erreichen lassen (Effizienz).

Häufig wird dieses Konzept noch um eine Planer-Komponente ergänzt. Die Funktionsweise eines BDI-Agenten lässt sich in Pseudocode folgendermaßen darstellen:

```
wiederhole {
        // Eingabe (Sensoren)
        w := wahrnehmen(r);
        // Verarbeitung
        b_neu := aktualisieren(b_alt, w);          // Beliefs
        d_neu := ermitteln(d_alt, b_neu);          // Desires
        i_neu := filtern(i_alt, d_neu, b_neu);     // Intensions
        p_neu := auswaehlen(p_alt, i_neu, b_neu); // Plans
        // Ausgabe (Aktoren)
        a := agieren(p_neu);
}
```

Listing 2.1: Pseudocode BDI-Agent

Der Arbeitsablauf des BDI-Agenten wiederholt sich ständig:

1. Gemäß des Eingabe-Verarbeitung-Ausgabe (EVA)-Prinzips werden Eingaben zu Ausgaben verarbeitet. Dabei nimmt der Agent über seine Sensoren zunächst den Reiz r aus der Umwelt wahr. Das Ergebnis wird als Wahrnehmung w gespeichert.

2. Dann wird der Umweltzustand b_{neu} aktualisiert, wobei der alte Umweltzustand b_{alt} und die Wahrnehmung w als Argumente herangezogen werden (Beliefs-Komponente).

3. Danach werden die neuen Wunschzustände d_{neu} ermittelt, wobei die alten Wunschzustände d_{alt} und der neue Umweltzustand b_{neu} als Argumente dienen (Desires-Komponente).

4. Jetzt wird die neue Absicht i_{neu} aus einer Menge von Möglichkeiten ermittelt. Dabei werden sowohl die alten Absichten i_{alt} betrachtet, als auch Absichten generiert, die sich aus den neuen Wünschen d_{neu} ergeben. Der neue Umweltzustand b_{neu} wird ebenfalls als Argument benötigt, damit nicht Absichten ausgewählt werden, die sich gar nicht realisieren und ausführen lassen. Somit wirkt diese Funktion wie ein Filter (Intentions-Komponente).

5. Die Absicht muss nun noch in eine Aktion umgesetzt werden. Dies ist aber nicht immer in einem Schritt möglich und deshalb gibt es eine Planer-Komponente mit einer zugehörigen Bibliothek. In dieser sind Pläne mit Vorschriften enthalten, um von einem Zustand in einen anderen Zustand zu kommen. Ein neuer Plan p_{neu} wird ausgewählt, indem die alten Pläne p_{alt}, die neue Absicht i_{neu} und das neue Weltbild b_{neu} betrachtet werden. Der neue Plan muss einerseits zum neuen Weltbild passen (Vorbedingung) und andererseits muss das Ergebnis dieses Plans nach der Ausführung zur neuen Absicht passen (Nachbedingung).

6. Schließlich wird dieser ausgewählte Plan p_{neu} umgesetzt, indem die entsprechende Aktion a über den Aktoren des Agenten ausgeführt wird.

2.3 Multiagentensysteme

Dem Namen nach ist ein Multiagentensystem (MAS) ein System, das aus mehreren Agenten besteht. Eine genauere Definition liefert das Glossar der Foundation for Intelligent Physical Agents (vgl. Abschnitt 2.3.2) [FIP03]:

> *« A Multi-Agent System is a system composed of a great number of autonomous entities, named agents, having a collective behavior that allows to obtain the desired function / service. »*

Gemäß dieser Definition gibt es keine zentrale Einheit, welche die Agenten steuert. Somit gibt es keine globale Kontrolle über das System, stattdessen agieren alle Agenten autonom. Die Agenten verhalten sich aber nicht vollständig unabhängig voneinander, sondern sie zeigen auf der makroskopischen Ebene ein kollektives Verhalten. Sie stehen beispielsweise im Wettbewerb zueinander (Konkurrenz) oder arbeiten zusammen im Team (Kooperation). Dies impliziert Koordinations- und Kommunikationsmechanismen. Dahinter verbirgt sich auch die Annahme, dass jeder Agent nur beschränkte Fähigkeiten besitzt und über unvollständiges Wissen über die Welt verfügt. Somit müssen Agenten zusammenarbeiten, um bestimmte Probleme überhaupt lösen zu können. Es wird auch kein globaler Takt vom System vorgegeben, sondern die Berechnungen und Aktionen der Agenten finden nebenläufig statt. Die Autonomie der Agenten verlangt auch, dass jeder Agent über eigene Ressourcen verfügt und seine Daten (z. B. seinen internen Zustand) lokal speichert und verarbeitet. Das MAS wird zu einem bestimmten Zweck konstruiert. Letztendlich liefert das Zusammenwirken der vielen Agenten das gewünschte Ergebnis der Problemlösung in Form einer Funktionalität bzw. eines Dienstes.

2.3.1 Merkmale von MAS

Kommunikation ist ein zentraler Aspekt eines MAS. Agenten können kommunizieren, indem sie Nachrichten austauschen. Dieser Austausch kann synchron wie bei einem Telefongespräch oder asynchron wie bei einer E-Mail erfolgen. Er kann direkt zwischen den beteiligten Agenten erfolgen oder indirekt mit Hilfe einer gemeinsam genutzten Komponente, in der die Nachrichten abgelegt und entnommen werden können. Die erfolgreiche

Kommunikation setzt voraus, dass die Agenten eine gemeinsame Sprache sprechen und über ein gemeinsames Grundwissen verfügen. Zu der verwendeten Sprache muss die Syntax (vereinfacht: Grammatikregeln) und Semantik (vereinfacht: Bedeutung) definiert sein. Die Ontologie (vereinfacht: Vokabular) definiert die Begriffe und die Beziehungen zwischen diesen zum Zweck der Wissensrepräsentation [Woo09].

Ein MAS kann als **Verteilte Künstliche Intelligenz (VKI)** betrachtet werden. Einerseits kann jeder lernende Agent bereits über Intelligenz verfügen (vgl. Abschnitt 2.2), andererseits kann das Zusammenwirken von reaktiven Agenten auf der Makroebene neue Strukturen hervorbringen, die als eine Art von Schwarzintelligenz betrachtet werden können. Diese durch Selbstorganisation entstehenden geordneten Strukturen bezeichnet man auch als Emergenz (vgl. Abschnitt 2.1). Die **Agentenbasierte Modellierung (ABM)** (engl. *Agent-based Modeling*) beschäftigt sich mit der computergestützten Modellbildung und Simulation von genau solchen Systemen. Das Ziel einer ABM ist es, Erklärungen für das kollektive Verhalten der Agenten zu finden. Dabei werden die Agenten normalerweise als reaktive Agenten modelliert, die sich wie einfache Reflex-Agenten verhalten (vgl. Abschnitt 2.2.2). Somit ist die ABM ein Spezialfall von MAS.

Die Vorteile der **Kooperation** von Agenten sind:
- Bearbeitung von Aufgaben, die ein Agent hätte allein nicht durchführen können
- Steigerung der Produktivität jedes einzelnen Agenten
- Erhöhung der Gesamt-Effizienz (Anzahl bearbeiteter Aufgaben pro Zeit)
- Verbesserung der Ressourcennutzung

2.3.2 FIPA-Standard

Die Foundation for Intelligent Physical Agents (FIPA) ist die wichtigste Organisation hinsichtlich der Standardisierung von Multiagentensystemen. Sie wurde 1996 gegründet und hat eine Referenzarchitektur zu einem verteilten MAS entwickelt, das in 25 Spezifikationen näher beschrieben ist [Pos07]. Mit diesem Ansatz können sogar Agenten aus unterschiedlichen MAS miteinander kommunizieren und kooperieren. Die Referenzarchitektur besteht aus folgenden Komponenten (vgl. Abb. 2.8):

Das **Agent Management System (AMS)** verwaltet die Agenten in einem Agentenverzeichnisdienst, vergleichbar mit einem Telefonbuch (*White Pages*). Jeder Agent wird hier registriert und bekommt dabei eine plattformübergreifende eindeutige Kennung. Der Lebenszyklus der Agenten wird vom AMS ebenfalls überwacht und besteht aus den Phasen Initialisierung, Aktivierung, Suspendierung, Migration und Beenden. Jede Plattform eines MAS benötigt ein eigenes AMS. Mit diesem Mechanismus können Agenten während der Laufzeit hinzugefügt, verschoben (migriert) oder entfernt werden, wodurch ein flexibles, offenes System gekennzeichnet ist.

Der **Directory Facilitator (DF)** ist vergleichbar mit den Gelben Seiten (*Yellow Pages*) und unterstützt die Bekanntgabe und Suche von bestimmten Diensten, welche die Agenten anbieten. Denn jeder Agent verfügt über bestimmte Fähigkeiten und diese kann er nutzen, um bestimmte Funktionen durchzuführen. Für die Kooperation von Agenten ist dieser Dienst sehr hilfreich.

Das **Message Transport System (MTS)** ist die zentrale Komponente zur Kommunikation. Sie ist vergleichbar mit einem E-Mail-Dienst. Mittels Message Transport Protocol (MTP) werden in Agent Communication Language (ACL) verfasste Nachrichten über den Agent

Communication Channel (ACC) versendet. Die Kommunikation kann auch zwischen Agenten verschiedener Plattformen erfolgen.

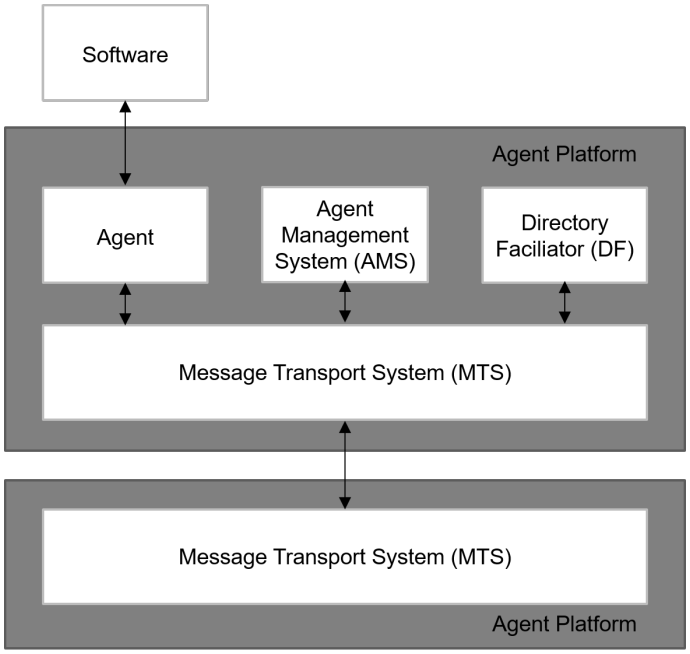

Abbildung 2.8: FIPA-Referenzarchitektur [FIP04]

Die Spezifikation der Sprache ACL sieht einige wichtige Parameter für den Nachrichtenaustausch vor [FIP02a]:

performative	—	Absicht hinter der Nachricht
sender	—	Absender
receiver	—	Empfänger
reply-to	—	Antwort an diesen Agenten statt an den Absender
content	—	Nachrichteninhalt
language	—	Sprache der Nachricht
encoding	—	Codierung der Nachricht
ontology	—	Ontologie der Nachricht in Verbindung mit der Sprache
protocol	—	Interaktionsprotokoll für den Nachrichtenaustausch
conversation-id	—	Identifizierung für einen längeren Konversationsverlauf
reply-with	—	Kennzeichen zu einer speziellen Nachricht
in-reply-to	—	Antwort bezieht sich auf Nachricht mit *reply-with*
reply-by	—	Datum & Uhrzeit für die späteste Antwort

Der Agent kann mit dem Parameter *performative* bereits seine Absichten ausdrücken, was die Kommunikation erleichtert. In Tabelle 2.1 sind zu einigen möglichen Werten von *performative* exemplarische Nachrichten zur Veranschaulichung dargestellt [FIP02b]. Diese einfachen Beispiele zeigen, dass beide Agenten bzw. Gesprächspartner über dasselbe Wissen verfügen müssen. Falls unklar ist, was mit dem Begriff Tür gemeint ist, dann scheitert die Konversation bereits sehr früh. Weiterhin muss für beide klar sein, um welche Tür es sich handelt. Eine Tür kann drei Zustände einnehmen: offen, geschlossen und verschlossen. Zum Verschließen einer Tür wird außerdem ein Schlüssel benötigt. Dieses Vokabular

performative	**Bedeutung**	**Beispielnachricht**
query	Frage stellen	Ist die Tür offen?
request	Anfrage stellen	Öffne die Tür (für mich).
agree	Anfrage zustimmen	Einverstanden! Ich öffne die Tür (für Dich).
inform	Informationen teilen	Die Tür ist offen.
failure	Problem berichten	Ich kann die Tür nicht (für Dich) öffnen. (Sie ist verschlossen.)
refuse	Anfrage ablehnen	Ich will die Tür nicht (für Dich) öffnen.
call-for-proposal	(Generellen) Aufruf starten	Kann jemand die Tür (für mich) öffnen?
proposal	Vorschlag machen	Ich kann die Tür (für Dich) öffnen. Zum Preis von 100 Euro.
accept-proposal	Vorschlag annehmen	Einverstanden! Der Preis ist in Ordnung.
reject-proposal	Vorschlag ablehnen	Nein! Das ist zu teuer.
not-understood	Nachricht nicht verstanden	Tür? Was ist das? Ich verstehe das nicht.

Tabelle 2.1: Absichten mit Beispielnachrichten

und Wissen muss in der Ontologie abgebildet werden. Hierzu kann die Web Ontology Language (OWL) mit dem Resource Description Framework (RDF)-Schema benutzt werden, das auf Extensible Markup Language (XML) basiert. Im folgenden Beispiel-Listing stellt Mary eine Anfrage an John, die beiden Türen zu öffnen.

```
(request
 :sender Mary
 :receiver John
 :content (
  <?xml version="1.0"?>
  <rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
           xmlns:fipa="http://www.fipa.org/schemas#">
      <fipa:Action rdf:ID="JohnAction1">
        <fipa:actor>John</rdf:actor>
        <fipa:act>open</rdf:act>
        <fipa:argument>
          <rdf:bag>
            <rdf:li>door1</rdf:li>
            <rdf:li>door2</rdf:li>
          </rdf:bag>
        </fipa:argument>
      </fipa:Action>
  </rdf:RDF>)
 :language fipa-rdf0
)
```

Listing 2.2: ACL-Beispiel mittels RDF [FIP01]

3 Frameworks

In diesem Kapitel werden Frameworks zur Modellierung und Simulation von Multiagentensystemen vorgestellt. Zunächst wird aber kurz der Begriff des Frameworks erklärt, das MVC-Konzept dargestellt und dann werden die Anforderungen bei der Auswahl der Frameworks erläutert.

Ein **Framework** ist ein Programmiergerüst, das auf einer Softwarearchitektur basiert und bereits viele Softwarekomponenten zur Verfügung stellt. Der Entwickler muss also nicht die Anwendung von Grund auf selbst erstellen, sondern er kann auf diese Bausteine direkt zugreifen. Das verkürzt die Entwicklungszeit und verringert den Entwicklungsaufwand. Im Unterschied zu einer Software-Bibliothek bietet ein Framework aber nicht nur wiederverwendbare Komponenten an, sondern diese Komponenten arbeiten bereits innerhalb der vorgegebenen Rahmenstruktur zusammen. Meistens ist bereits eine Ablaufsteuerung der Anwendung im Framework integriert.

3.1 MVC-Konzept

Ein sehr verbreitetes Entwurfs- und Architekturmuster, das viele Frameworks als *de facto* Standard einsetzen, ist das sogenannte Model View Controller (MVC)-Konzept, welches schematisch in Abb. 3.1 dargestellt ist.

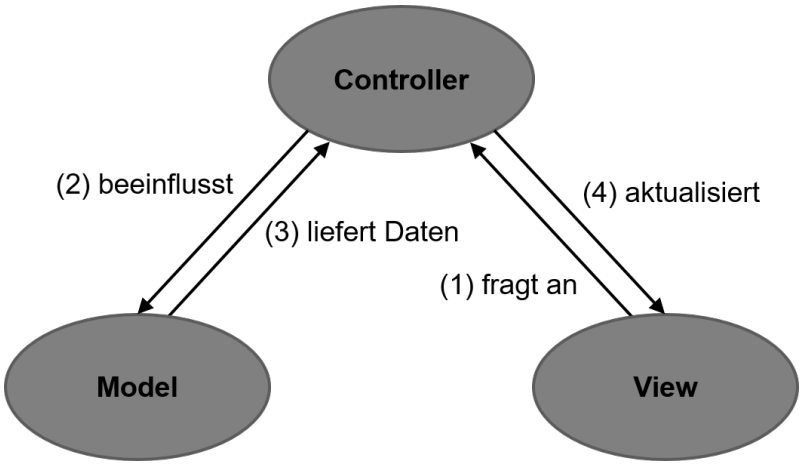

Abbildung 3.1: MVC-Schema

Das **Modell** (engl. *Model*) enthält die zu verarbeitenden Daten und die Geschäftslogik.

Die **Präsentation** (engl. *View*) stellt die Daten aus dem Modell für den Anwender entsprechend dar. Eine Präsentation kann sehr leicht ausgetauscht werden. Beispielsweise kann für jedes Endgerät (Laptop, Tablet, Smartphone usw.) eine eigene Präsentation erstellt werden. Neben der Darstellung der Daten, ist die Präsentation auch für die Entgegennahme der Interaktionen des Anwenders zuständig. Je nach verwendetem Endgerät können

die Interaktionen unterschiedliche sein: Eingaben per Tastatur und Maus, berührungssensitive Gesten usw.

Die **Steuerung** (engl. *Contoller*) verwaltet mindestens eine Präsentation. Sie bekommt die Eingaben des Anwenders von der Präsentation als Anfrage übergeben (Schritt 1), wertet diese aus und ruft dann entsprechende Aktionen auf oder leitet die Eingaben des Anwenders als Daten an das Modell weiter (Schritt 2). Wenn sich nun im Modell, d.h. in den Daten, etwas ändert, dann wird dies der Steuerung mitgeteilt (Schritt 3) und diese Änderungen werden dem Anwender angezeigt, indem die Präsentation aktualisiert wird (Schritt 4).

3.2 Anforderungen

Es werden in dieser Arbeit nur MAS-Frameworks betrachtet, die diese fünf grundlegende Eigenschaften bzw. Anforderungen erfüllen:

Generisch	—	Das Framework muss für beliebige Zwecke verwendet werden.
Unabhängig	—	Das Framework muss sich auf allen Plattformen betreiben lassen.
Offen	—	Das Framework muss offen sein für (eigene) Erweiterungen.
Standardisiert	—	Das Framework muss die üblichen Standards erfüllen.
Intelligent	—	Es müssen sich auch intelligente Agenten modellieren lassen.

Anforderung #1: Einige Frameworks wurden für spezielle Einsatzzwecke entwickelt und optimiert. Beispielsweise kann Java Agent-based MacroEconomic Laboratory (JAMEL) [Sep17], wie der Name schon sagt, für makroökonomische Analysen verwendet werden. Ein Blick in das Application Programming Interface (API) dieses Frameworks zeigt, dass sehr viele Java-Klassen zu diesem Kontext gehören: *Banks*, *Firms*, *Housholds* usw. Es ist unklar, ob solch ein Framework auch für andere Zwecke eingesetzt werden kann. Falls doch, dann stellt sich die Frage, wie hoch der Anpassungsaufwand wäre. Deshalb werden nur Framewoks berücksichtigt, die für allgemeine Anwendungszwecke entwickelt wurden und sich somit generisch für beliebige Modellierungen und Simulationen von MAS einsetzen lassen.

Anforderung #2: Die nächste Anforderung ist nicht ganz korrekt formuliert. Mit dieser Anforderung ist gemeint, dass das Framework auf gängigen Plattformen wie Windows, MacOSX und Unix/Linux funktionieren muss. Die Unterstützung von weiteren, insbesondere mobilen Plattformen, wie Android, iOS usw. ist wünschenswert. Viele MAS-Frameworks sind in Java programmiert. Java ist sowohl eine moderne, objektorientierte Programmiersprache, als auch eine Middleware, die auf den meisten Systemen als Java Virtual Machine (JVM) installiert werden kann. Aus diesem Grund ist es sehr leicht, Java-Programme zu portieren, d. h. von einem auf ein anderes System zu übertragen.

Anforderung #3: Die dritte Anforderung zielt auf Flexibilität und Erweiterungsfähigkeit des Frameworks ab. Es müssen sich andere Lösungen in das Framework integrieren oder zumindest per geeigneter Schnittstelle ansteuern lassen. Auch eigene Ideen und Implementierungen müssen sich einbauen lassen. Im Idealfall bedeutet dies, dass das Framework quelloffen ist und die Quelltexte frei zugänglich sind, man spricht dann auch von

Open Source. Das bedeutet jedoch nicht, dass der Einsatz des Frameworks komplett kostenlos ist. Zwar entfallen Lizenzgebühren, aber gemäß des Total Cost of Ownership (TCO)-Konzepts entstehen bspw. auch Kosten durch Hardware, Installation, Konfiguration, Betrieb, Pflege und Wartung, also für Sachmittel, Personal und Energie. Das Framework JACK Intelligent Agents [Gro17] ist bspw. kein *Open Source* sondern eine proprietäre Softwarelösung, die kommerziell vertrieben wird.

Anforderung #4: Es ist sinnvoll, etablierte Standards zu verwenden, denn im Rahmen des Standardisierungsprozesses ist bereits viel Wissen im Sinne von *Best Practices* in die Spezifikation eingeflossen. Der Standard für Multiagentensystem ist die FIPA-Referenzarchitektur (vgl. Abschnitt 2.3.2). Die untersuchten Softwarelösungen müssen also diesen Standard erfüllen, was streng genommen nur sehr wenige Frameworks tun. Es werden deshalb auch Frameworks betrachtet, die über spezielle Erweiterungen, sogenannte Plug-Ins, die FIPA-Referenzarchitektur berücksichtigen. Ein weiterer Standard ist die BDI-Architektur (vgl. Abschnitt 2.2.3) als Basis für intelligente Agenten (siehe auch Anforderung #5).

Anforderung #5: Im Abschnitt 2.3.1 wurden bereits die beiden Begriffe MAS und ABM voneinander abgegrenzt. Viele Softwarelösungen folgen dem Ansatz der Agentenbasierte Modellierung, wie bspw. MASON [Luk+17], NetLogo [Wil17], REPAST [Alt+17] und Swarm [Rog17]. Der Fokus dieser Anwendungen liegt in der Analyse der emergenten Effekte des kollektiven Verhaltens der Agenten. Mit dem MAS-Framework müssen sich aber auch lernende Agenten modellieren lassen, bspw. als BDI-Agent.

3.3 Auswahl

In diesem Abschnitt werden nun ausgewählte Frameworks in alphabetischer Reihenfolge steckbriefhaft vorgestellt. Diese Auswahl erhebt keinen Anspruch auf Vollständigkeit. Auf Basis einer sorgfältigen Internetrecherche werden diejenigen Frameworks betrachtet, die eine hohe Relevanz bezüglich des Schlagworts MAS haben und alle genannten Anforderungen erfüllen (vgl. Abschnitt 3.2). Hilfreich erwiesen sich dabei die folgenden drei Quellen:

FIPA	Available Implementations of FIPA Specifications	[FIP17]
Kravari & Bassiliades	A Survey of Agent Platforms	[KB15]
Wikipedia	Comparison of Agent-based Modeling Software	[Wik17]

Aufgrund des jeweiligen Untersuchungsdatums finden sich jedoch teilweise widersprüchliche Angaben in den Arbeiten zu diesen Quellen. In dieser aktuellen Recherche (Stand: 05.01.2018) werden diese Widersprüche aufgelöst und die derzeit gültigen Attribute der Frameworks angegeben. Hierfür werden die aktuellen Webseiten zu den betrachteten Frameworks analysiert. Diese Frameworks sind:

1. JADE (Java Agent Development Framework)
2. Jadex
3. Jason
4. SeSAm (Shell for Simulated Agent Systems)
5. ZEUS

3.3.1 JADE

Java Agent Development Framework (JADE) ist ein Framework, das vom Unternehmen Telecom Italia entwickelt wurde und die Implementierung von Multiagentensystem beschleunigen soll. Hierzu kommt eine Middleware zum Einsatz, die auf dem FIPA-Standard basiert (vgl. Abschnitt 2.3.2). Außerdem werden grafische Werkzeuge angeboten, um die Fehlersuche, die Konfiguration und das Ausführen der Agenten zu vereinfachen. Ein JADE-basiertes System kann sogar auf verschiedene Plattformen unterschiedlicher Betriebssysteme verteilt und gesteuert werden. Seit der Version 1.3 vom Februar 2000 ist JADE *Open Source*. Tabelle 3.1 zeigt weitere Merkmale des JADE-Frameworks [Bel+17].

Name	Java Agent Development Framework (JADE)
Organisation	Telecom Italia Lab
Webseite	http://jade.tilab.com
Aktualisierung	08.06.2017
Download	http://jade.tilab.com/download/jade/license/jade-download
Akt. Version	4.5 (08.06.2017)
Lizenz	LGPL Version 2
Open Source	Ja
Plattformen	Microsoft, MacOS, Unix/Linux
Geschrieben in	Java
Community	http://jade.tilab.com/about-us/ http://jade.tilab.com/credits/
Standards	FIPA
Schnittstellen	Maven: http://jade.tilab.com/developers/maven/
Erweiterungen	http://jade.tilab.com/download/add-ons/ http://jade.tilab.com/download/third-party-contributions/
Projekte	WADE: http://jade.tilab.com/wadeproject/ AMUSE: http://jade.tilab.com/amuseproject/
Bücher	Bellifemine, F. L., Caire, G., Greenwood, D.: *Developing Multi-Agent Systems with JADE* Wiley, New York, 2007. http://jade.tilab.com/documentation/book/
Artikel	http://jade.tilab.com/documentation/papers/
Dokumentation	http://jade.tilab.com/doc/api/index.html
Tutorials	http://jade.tilab.com/documentation/tutorials-guides/
Demos	http://jade.tilab.com/documentation/chat-demo/
Beispiele	http://jade.tilab.com/documentation/examples/
API	http://jade.tilab.com/doc/api/overview-summary.html
Wikipedia	https://en.wikipedia.org/wiki/Java_Agent_Development_Framework

Tabelle 3.1: Steckbrief zum JADE-Framework

3.3.2 Jadex

Jadex ist ursprünglich am Institut für Verteilte Systeme und Informationssysteme (VSIS) der Universität Hamburg von Lars Braubach und Alexander Pokahr entwickelt worden. Mit diesem Framework lassen sich intelligente Software-Agenten in XML und Java gemäß der BDI-Architektur erstellen. Außerdem existiert eine Schnittstelle zum MAS-Framework JADE (vgl. Abschnitt 3.3.1), welches den FIPA-Standard erfüllt. Tabelle 3.2 zeigt die wichtigsten Merkmale des Jadex-Frameworks zusammengefasst [PBJ17].

Name	Jadex
Organisation	Actoron GmbH
Webseite	https://www.activecomponents.org/#/project/news
Aktualisierung	03.08.2017
Download	https://www.activecomponents.org/#/download
Akt. Version	3.0.85 (20.12.2017)
Lizenz	GPL Version 3
Open Source	Ja
Plattformen	Windows, MacOS, Unix/Linux
Geschrieben in	Java
Community	https://trac.actoron.com
Standards	BDI
Schnittstellen	JADE
Erweiterungen	-
Projekte	-
Bücher	-
Artikel	https://www.activecomponents.org/#/docs/publications
Dokumentation	https://www.activecomponents.org/index.html#/docs/overview
Tutorials	https://download.actoron.com/docs/releases/latest/jadex-mkdocs/
Demos	-
Beispiele	https://www.activecomponents.org/#/docs/examples
API	https://download.actoron.com/docs/releases/latest/javadoc/
Wikipedia	-

Tabelle 3.2: Steckbrief zum Jadex-Framework

3.3.3 Jason

Jason wurde von Jomi F. Hübner und Rafael H. Bordini entwickelt. Ein wesentliches Werkzeug ist ein Interpreter, der auf AgentSpeak basiert, eine BDI-Agentenorientierte logische Programmiersprache. Jason ist in Java implementiert und kann als Plattform dazu benutzt werden, um MAS zu bauen. Mit Hilfe von JADE (vgl. Abschnitt 3.3.1) lassen sich die Agenten dabei auch über ein Netzwerk verteilen. Jason wird u.a. auch von Michael Wooldridge an der Universität von Oxford verwendet. Tabelle 3.3 zeigt die wichtigsten Merkmale des Jason-Frameworks zusammengefasst [HB17].

Name	Jason
Organisation	-
Webseite	http://jason.sourceforge.net/wp/
Aktualisierung	25.09.2017
Download	https://github.com/jason-lang/jason https://sourceforge.net/projects/jason/files/
Akt. Version	2.3 (19.12.2017)
Lizenz	LGPL
Open Source	Ja
Plattformen	Windows, MacOS, Unix/Linux
Geschrieben in	Java, AgentSpeak
Community	-
Standards	BDI
Schnittstellen	Eclipse
Erweiterungen	JADE
Projekte	http://jason.sourceforge.net/wp/projects/
Bücher	Bordini, R. H., Hübner, J. F. und Wooldridge, M.: *Programming Multi-agent Systems in AgentSpeak Using Jason* Wiley-Blackwell, 2007. http://jason.sourceforge.net/jBook/jBook/Home.html
Artikel	http://jason.sourceforge.net/wp/documents/
Dokumentation	PDF, 31 Seiten (Februar 2007): http://jason.sourceforge.net/Jason.pdf
Tutorials	http://jason.sourceforge.net/mini-tutorial/getting-started/ http://jason.sourceforge.net/mini-tutorial/hello-bdi/
Demos	http://jason.sourceforge.net/wp/demos/
Beispiele	http://jason.sourceforge.net/wp/examples/ http://jason.sourceforge.net/jBook/jBook/Examples.html
API	http://jason.sourceforge.net/api/
Wikipedia	https://en.wikipedia.org/wiki/Jason _(multiagent_systems_development_platform)

Tabelle 3.3: Steckbrief zum Jason-Framework

3.3.4 SeSAm

Shell for Simulated Agent Systems (SeSAm) ist eine generische Umgebung zur Modellierung und Simulation von Multiagentensystemen. Die Software basiert auf der Promotion von Franziska Klügl an der Universität Würzburg. Ursprünglich wurde sie 1998 in LISP programmiert, später dann in Java re-implementiert. Die Forschungsgruppe ist mittlerweile an der Örebro University in Schweden aktiv. Tabelle 3.4 zeigt die wichtigsten Merkmale des SeSAm-Frameworks [KJ17]. Die neuste Version der Software ist zwar aus dem Jahr 2017, die Webseite und die Dokumentation sind aber seit 2012 bzw. 2003 nicht mehr aktualisiert worden.

Name	Shell for Simulated Agent Systems (SeSAm)
Organisation	Universität Würzburg Örebro University
Webseite	http://www.simsesam.de
Aktualisierung	21.05.2012
Download	http://130.243.124.21/sesam/index.php/files/download/25
Akt. Version	06.02.2017
Lizenz	LGPL
Open Source	Ja
Plattformen	Windows, MacOS, Unix/Linux
Geschrieben in	Java
Community	-
Standards	-
Schnittstellen	FIPA
Erweiterungen	FIPA-Plug-In: http://130.243.124.21/mediawiki/index.php/FipaPlugin http://130.243.124.21/mediawiki/index.php/Plugins
Projekte	-
Bücher	-
Artikel	-
Dokumentation	Wiki: http://130.243.124.21/mediawiki/index.php/Main_Page
Tutorials	PDF, 50 Seiten (13.06.2003): http://130.243.124.21/mediawiki/index.php/Tutorials
Demos	-
Beispiele	http://130.243.124.21/mediawiki/index.php/ModelLibrary
API	-
Wikipedia	-

Tabelle 3.4: Steckbrief zum SeSAm-Framework

3.3.5 ZEUS

Das ZEUS Agent Building Toolkit wurde von der Forschungsgruppe Intelligent Systems der British Telecom (BT) entwickelt. Es ist eine integrierte Entwicklungsumgebung (engl. *Integrated Development Environment (IDE)*), um verteilte Agentensysteme zu erstellen. Tabelle 3.5 zeigt die Merkmale der Software [CLT17]. Die ursprüngliche Webseite (s.u.) ist aktuell nicht mehr erreichbar. Die neuste Version der Software ist auch aus dem Jahr 2006, also bereits mehr als 10 Jahre alt.

Name	ZEUS Agent Building Toolkit
Organisation	BT Labs' Intelligent Systems Research Group
Webseite	http://labs.bt.com/projects/agents/zeus
Aktualisierung	07.03.2013
Download	https://sourceforge.net/projects/zeusagent
Akt. Version	2.0 patch 2 (10.01.2006)
Lizenz	BT 'ZEUS' Open Source Licence, Version 1.0
Open Source	Ja
Plattformen	Windows, MacOS, Unix/Linux
Geschrieben in	Java
Community	7 Entwickler
Standards	FIPA
Schnittstellen	-
Erweiterungen	-
Projekte	-
Bücher	-
Artikel	https://link.springer.com/article/10.1023/A:1009673714049 http://www.aaai.org/Papers/Workshops/1998/WS-98-10 http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.26.277
Dokumentation	-
Tutorials	PDF, 9 Seiten (03.02.2001): http://www.upv.es/sma/plataformas/zeus/Zeus-lessonone.pdf
Demos	-
Beispiele	3 Fallstudien im Installationspaket
API	im Installationspaket
Wikipedia	-

Tabelle 3.5: Steckbrief zum ZEUS-Framework

4 Analysen

In Kap. 3 sind fünf MAS-Frameworks vorgestellt wurden, welche fünf spezielle Anforderungen erfüllen. Das Ziel ist es nun, aus diesen Alternativen diejenige auszuwählen, die für den Einsatz in der Lehre und zu Forschungszwecken am geeignetsten erscheint. Hierzu wird eine Nutzwertanalyse durchgeführt.

4.1 Nutzwertanalyse

Nach Zangemeister ist eine Nutzwertanalyse eine [Zan70]:

> *« Analyse einer Menge komplexer Handlungsalternativen mit dem Zweck, die Elemente dieser Menge entsprechend den Präferenzen des Entscheidungsträgers bezüglich eines multidimensionalen Zielsystems zu ordnen. Die Abbildung der Ordnung erfolgt durch die Angabe der Nutzwerte (Gesamtwerte) der Alternativen. »*

Mit einer Nutzwertanalyse lässt sich also herausfinden, welche Alternative den höchsten Nutzen bietet. Sie wird auch Scoring-Verfahren oder Scoring-Modell genannt. Mathematisch betrachtet, gilt es die Zielfunktion:

$$f(x_1, ..., x_n) = w_1 f_1(x_1) + ... + w_n f_n(x_n) \tag{4.1}$$

zu maximieren. Dabei sind x_i das zu bewertende Merkmal i, w_i das Gewicht zum Merkmal i und $f_i(x_i)$ die Bewertungsfunktion des i-ten Merkmals, wobei n die Anzahl der Merkmale angibt. Es gilt: $0 \leq w_i \leq 1$ und die Nebenbedingung:

$$\sum_{i=1}^{n} w_i = 1 \tag{4.2}$$

muss erfüllt sein.

4.2 Merkmale, Gewichte und Skalen

Um eine Nutzwertanalyse durchzuführen und gemäß Gl. 4.1 müssen also die Merkmale, die zugehörigen Gewichte und die Bewertungsfunktionen bzw. Skalen definiert werden. In Kap. 3 wurden bereits die Merkmale *Generisch*, *Unabhängig*, *Offen*, *Standardisiert* und *Intelligent* betrachtet und als K.O.-Kriterien zur Filterung der Frameworks verwendet. Nun werden neue Merkmale definiert, die zur Nutzwertanalyse herangezogen werden:

Aktualität	—	Alter und Historie der Software	—	20 %
Funktionen	—	Standards, Schnittstellen, Erweiterungen, Projekte	—	20 %
Hilfe	—	Dokumentation, Tutorials, Beispiele, Demos, API	—	20 %
Reputation	—	Organisation, Publikationen: Bücher, Artikel	—	20 %
Präsentation	—	Webseite, Community, Erster Eindruck	—	20 %

Merkmal #1: Wenn Software nicht regelmäßig aktualisiert wird, dann veraltet sie. Veraltete Software wird irgendwann nicht mehr eingesetzt und verschwindet dann vom Markt. Insofern ist es wichtig, dass Software weiterentwickelt wird. In diesem Zusammenhang spricht man von Pflege und Wartung. Pflege von Software bedeutet, dass Fehler (engl. *Bugs*) korrigiert werden. Unter Wartung versteht man die Anpassung der Software an neue Benutzerwünsche (engl. *Change Requests*), geänderte Systemumgebungen, neue gesetzliche Bestimmungen usw. Das Ergebnis ist eine neue Version der Software (engl. *Update*). Versionsnummern geben Aufschluss über die Historie der Software. Eine Nummer kleiner Eins weißt normalerweise auf eine Beta-Version hin, die also noch nicht ganz ausgereift ist und sich in einem experimentellen Stadium befindet.

Merkmal #2: Jede Software stellt dem Anwender im Kern Funktionen bereit, um damit arbeiten zu können. Diese Funktionen werden in dieser Analyse aber nicht direkt betrachtet. Hierfür müsste die Software installiert, konfiguriert und getestet werden. Da die MAS-Frameworks sehr komplex sind, wäre dies ein sehr hoher Aufwand. Insofern wird die Funktionalität nur indirekt bewertet und zwar anhand der verwendeten Standards, angebotenen Schnittstellen und möglichen Erweiterungen. Außerdem werden Projekte betrachtet, welche die Software bereits erfolgreich einsetzen.

Merkmal #3: MAS-Frameworks sind komplex und vermutlich wenig benutzerfreundlich und intuitiv bedienbar. Wie bereits zu Merkmal 2 beschrieben, wäre es ein sehr hoher Aufwand, jede einzelne Softwarelösung zunächst ausgiebig zu testen, um herauszufinden wie es sich mit der jeweiligen *User Experience* verhält. Generell ist es aber wichtig, dass zu der Software entsprechende Hilfen in Form von Dokumentationen, Tutorials, Beispiele, Demonstrationen und die Programmierschnittstelle (engl. *Application Programming Interface (API)*) bereitgestellt werden, um den Einstieg zu erleichtern und das Arbeiten mit der Software möglichst effizient zu gestalten. Somit werden diese Dokumente als Ersatz für die *User Experience* bewertet.

Merkmal #4: Vertrauen in die Software ist ebenfalls ein wichtiges Kriterium. Selbst wenn die Software *Open Source* ist und daher keine Lizenzkosten anfallen, so entstehen nach dem *Total Cost of Ownership*-Konzept trotzdem Kosten, insbesondere laufende Kosten für den Betrieb (Energie, Personal). Insofern wird hier auch die Investitionssicherheit betrachtet und bewertet. Mit der Entscheidung und Festlegung auf eine Softwarelösung ist es schwierig und teuer, diese später zurückzunehmen und auf eine andere Alternative zu wechseln. Zu diesem Merkmal werden die Organisationen und Mitarbeiter hinter der Software betrachtet. Da es sich um wissenschaftliche Software handelt, wird ein besonderer Augenmerk auf die akademischen Publikationen gelegt.

Merkmal #5: Das letzte Merkmal betrachtet die Verpackung des Produkts. Auch diese sollte ansprechend sein. Hierzu wird zunächst die Webseite analysiert und deren Gestaltung, Infomativität und Aktualität beurteilt. Viele Organisationen, die *Open Source* Software anbieten, stellen diese auf zentrale Plattformen wie Sorceforge oder GitHub bereit. Auch der Auftritt auf diesen Plattformen und insbesondere die dahinterliegende Community wird bewertet. Dort lässt sich beispielsweise erkennen, wie mit Fehlermeldungen und deren Korrektur umgegangen wird.

Die Gewichte für die fünf Merkmale werden jeweils auf 0,2 gesetzt, was 20 Prozent entspricht. Es gibt keinerlei Präferenzen, sondern jedes Merkmal wird als gleich wichtig erachtet. Das ist zwar einfach, aber meistens auch sehr effektiv.

Zur Bewertung der Merkmale der Alternativen wird das Schulnotensystem verwendet, wobei jede Note einer Punktzahl zwischen null und fünf zugeordnet ist (vgl. Tab. 4.1).

Note	Bedeutung	Punkte
1	sehr gut	5
2	gut	4
3	befriedigend	3
4	ausreichend	2
5	ungenügend	1
6	mangelhaft	0

Tabelle 4.1: Punkteskala für die Bewertung der Alternativen

4.3 Ergebnisse

Die Verteilung der Punkte von null bis fünf für die einzelnen Merkmale der Alternativen ist nie ganz objektiv. Am Einfachsten ist sicherlich die Bewertung der Aktualität der Software, weil hier die Historie und das Datum der aktuellsten Version als quantitative Daten herangezogen werden können. Bei allen anderen Merkmalen wird die Qualität subjektiv bewertet. Die einzelnen Bewertungen, bei 5 Alternativen und 5 Merkmalen sind es insgesamt 25, werden nicht weiter erläutern. Ausgehend von den Steckbriefen (vgl. Abschnitt 3.3) werden Schulnoten vergeben und diese dann direkt in die Punktzahlen umgerechnet. In Tab. 4.2 ist das Ergebnis der Nutzwertanalyse dargestellt. Den höchsten Nutzwert von

Alternativen:		**JADE**		**Jadex**		**Jason**		**SeSAm**		**ZEUS**	
Merkmal	Gew.	Pkt.	NW	Pkt.	NW	Pkt.	NW	Pkt.	NW	Pkt.	NW
Aktualität	20 %	5	1,0	5	1,0	5	1,0	5	1,0	1	0,2
Funktionen	20 %	4	0,8	3	0,6	4	0,8	2	0,4	2	0,4
Hilfe	20 %	5	1,0	4	0,8	5	1,0	3	0,6	2	0,4
Reputation	20 %	5	1,0	3	0,6	4	0,8	2	0,4	5	0,8
Präsentation	20 %	5	1,0	5	1,0	4	0,8	3	0,6	1	0,2
Nutzwerte (NW):			**4,8**		**4,0**		**4,4**		**3,0**		**2,0**

Tabelle 4.2: Scoring-Tabelle zur Nutzwertanalyse

4,8 bekommt das Framework JADE, gefolgt von Jason und Jadex mit jeweils 4,4 bzw. 4,0 Gesamtpunkten. Interessant ist dabei, dass die diese drei Frameworks auch Schnittstellen zueinander haben. Der Fokus von Jason und Jadex liegt auf der Implementierung des BDI-Agenten, während JADE sich auf den FIPA-Standard konzentriert. Die Kombination dieser Frameworks könnte somit weitere Vorteile mit sich bringen.

Zu SeSAm gibt es vergleichsweise wenig Informationen und Hilfen. Die Einhaltung des FIPA-Standards ist nur über ein spezielles Plug-In als Erweiterung möglich. Das Framework ZEUS ist veraltet und wird offensichtlich nicht weiterentwickelt. Die zugehörige Webseite der British Telecom existiert nicht mehr.

5 Anwendung

In Kap. 3.3.1 wurde das MAS-Framework JADE bereits kurz mit den wichtigsten Merkmalen vorgestellt. In diesem Kapitel soll es nun eingesetzt werden. Hierzu ist es zunächst notwendig, die grundlegenden Konzepte von JADE zu verstehen. Anhand eines Anwendungsbeispiels werden die Konzepte vertieft. Außerdem wird beschrieben, wie die Software installiert, konfiguriert und das Anwendungsbeispiels gestartet wird. Abschließend wird ein Fazit bezüglich des Einsatzes von JADE in der Forschung und Lehre gezogen.

5.1 Konzepte

JADE ist komplett in Java programmiert, verwendet den FIPA-Standard und setzt somit auf das FIPA-Referenzmodell auf (vgl. 2.3.2). Mit dem Agent Management System (AMS) lassen sich die Agenten steuern. Hierzu werden u.a. die vier folgenden Java-Methoden verwendet:

`create()`	—	Agent erzeugen
`execute()`	—	Agent starten
`suspend()`	—	Agent anhalten
`terminate()`	—	Agent entfernen

Jeder Agent muss eindeutig identifizierbar sein. Hierzu wird die Klasse `jade.core.AID` zur Verfügung gestellt. Sie sorgt dafür, dass eine Global Unique Identifier (GUID) generiert wird. Außerdem werden Adressen erzeugt, damit die Agenten verschiedener Plattformen miteinander kommunizieren können. Der Agent selbst wird durch eine Java-Klasse beschrieben, welche von der Basisklasse `jade.core.Agent` abgeleitet ist. Wesentliche Methoden, die sich implementieren lassen, sind:

`setup()`	—	Agent initialisieren (u.a. GUID erzeugen)
`takeDown()`	—	Ressourcen wieder freigeben [Gegenstück zu `setup()`]
`doDelete()`	—	Agent beenden, d.h. terminieren

Die Implementierung eines Hallo-Welt-Agenten ist in dem folgenden Listing dargestellt. Dieser einfache Agent sagt nur kurz Hallo, meldet sich mit seinem Namen und verlässt dann das System schon wieder.

```
public class HelloWorldAgent extends Agent {
 protected void setup() {
  System.out.println("Hello World! My name is "+getLocalName());
  doDelete();
 }
}
```

Listing 5.1: Java-Klasse HelloWorldAgent – in Anlehnung an [Bel+17]

In der Methode `setup()` können aber auch die sogenannten *Behaviours* hinzugefügt werden, um das Verhalten des Agenten zu definieren. Die *Behaviours* sind von der Klasse `jade.core.behaviours.Behaviour` abgeleitet. Die Methode `action()` führt eine Aktion zu dem Verhalten aus, während die Methode `done()` prüft, ob das Ziel der Aktion erreicht wurde. Jeder Agent wird als eigenständiger Java-Thread, d.h. auf Betriebssystemebene als Prozess bzw. Task, ausgeführt. Er kann mehrere *Behaviour*-Objekte enthalten. Diese werden aber nicht parallel als eigene Java-Threads ausgeführt, sondern der Programmierer kann steuern, welches Verhalten ausgeführt wird. In Abb. 5.1 ist der Lebenszyklus eines Agenten schematisch dargestellt.

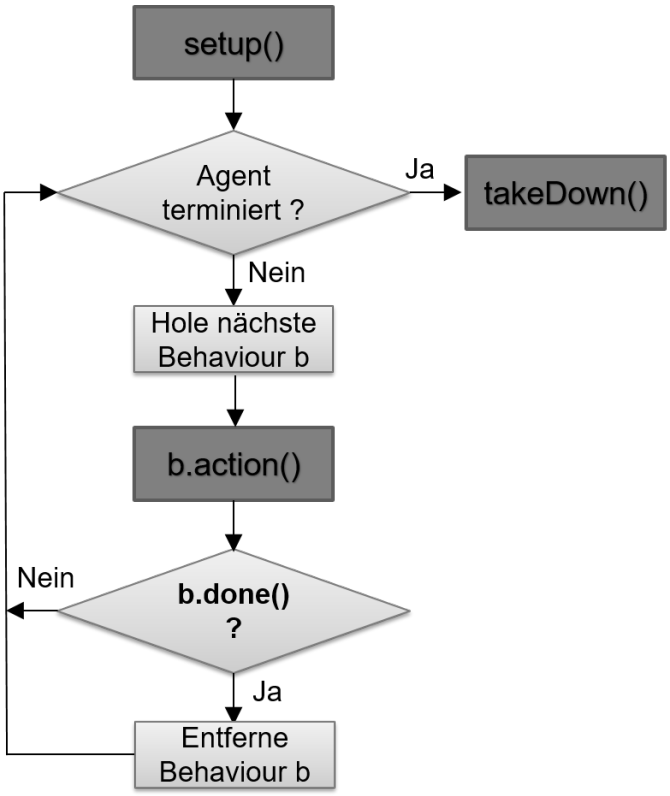

Abbildung 5.1: Lebenszyklus eines JADE-Agenten – in Anlehnung an [BCG07]

Zu Beginn wird der Agent erzeugt und mittels `setup()` initialisiert. Dabei wird auch ein Pool von *Behaviours* angelegt. Gemäß des First In - First Out (FIFO)-Prinzips werden die *Behaviours* `b` abgearbeitet, d.h. aus dem Pool geholt und mittels `b.action()` ausgeführt. Falls die Methode `b.done()` der aktuellen Behaviour `b` den Wahrheitswert `true` liefert, d.h. die Aktion erfolgreich beendet wurde, dann wird diese *Behaviour* wieder aus dem Pool entfernt. Falls der Pool leer ist oder explizit die Methode `doDelete()` aufgerufen wurde, der Agent also beendet werden soll, dann wird die entsprechende Methode `takeDown()` aufgerufen, die Ressourcen also wieder freigegeben und der Agent aus dem System entfernt. Der Lebenszyklus des Agenten ist damit beendet.

Die Implementierung eines einfachen Agenten mit unterschiedlichem Verhalten ist beispielhaft im folgenden Listing dargestellt.

```
public class SimpleAgent extends Agent {

 protected void setup() {

  System.out.println("Agent "+getLocalName()+" started.");

  // 1. add the OneShotBehaviour
  addBehaviour(new OneShotBehaviour(this) {
   public void action() {
    System.out.println("One-shot");
   } // action()
  }); // addBehaviour

  // 2. add the CyclicBehaviour
  addBehaviour(new CyclicBehaviour(this) {
   public void action() {
    System.out.println("Cycling");
   } // action()
  }); // addBehaviour

  // 3. add the generic behaviour
  addBehaviour(new ThreeStepBehaviour());

 } // setup()

 // inner class ThreeStepBehaviour
 private class ThreeStepBehaviour extends Behaviour {

  private int step = 1;

  public void action() {
   switch (step) {
    case 1: // perform operation 1
     System.out.println("Operation 1");
     step++;
     break;
    case 2: // perform operation 2
     System.out.println("Operation 2");
     step++;
     break;
    case 3: // perform operation 3
     System.out.println("Operation 3");
     step++;
     break;
   } // switch
  } // action()

  public boolean done() {
   return step == 4;
  } // done()

 } // inner class ThreeStepBehaviour

} // SimpleAgent
```

Listing 5.2: Java-Klasse SimpleAgent – in Anlehnung an [Bel+17]

Anhand des Quelltextes wird bereits deutlich, dass es unterschiedliche Möglichkeiten gibt, Standard-Verhalten zu implementieren. Hierzu stehen verschiedene Klassen zur

Verfügung, u.a.:

`OneShotBehaviour()`	—	Einmalige Aktionen
`CyclicBehaviour()`	—	Zyklische Aktionen (ggf. endlos)
`WakerBehaviour()`	—	Eingeplante Aktionen
`TickerBehaviour()`	—	Wiederholte Aktionen mit fester Frequenz

Zur Kommunikation von Agenten ist der ACL-Standard implementiert (vgl. Kap. 2.3.2). Es existiert eine Klasse `jade.lang.acl.ACLMessage` mit entsprechenden Methoden zum Nachrichtenaustausch, u.a.:

`setContent()`	—	Nachrichtentext erstellen
`addReceiver()`	—	Empfänger hinzufügen
`send()`	—	Nachricht senden
`receive()`	—	Nachricht empfangen

Zwecks Koordination von Agenten zur Bewältigung von komplexen Aufgaben gibt es die Gelben Seiten, d.h. den Directory Facilitator (DF) zur Registrierung und Suche von Agenten, die einen bestimmten Service anbieten. Die Registrierung sollte normalerweise in der Methode `setup()` erfolgen, während das Austragen aus den Gelben Seiten in der Methode `takeDown()` durchgeführt wird. Das Anwendungsbeispiel (vgl. Abschnitt 5.2) wird von diesen Möglichkeiten Gebrauch machen.

5.2 Beispiel

Das Anwendungsbeispiel zur Demonstration von JADE wurde von der Telecom Italia entwickelt und wird bereits zusammen mit den Installationsdateien ausgeliefert [BCG07]. Zur besseren Lesbarkeit und Präsentation des Quelltextes wurde jedoch ein Refactoring vorgenommen, d.h. dieser wurde modifiziert, ohne dass sich die Funktionalität ändert. Die Anwendung beschreibt einen Buchhandel. Im Wesentlichen gibt es zwei unterschiedliche Agenten: einen Verkäufer, den `BookSellerAgent`, der Bücher zum Verkauf anbietet, und einen Käufer, den `BookBuyerAgent`, der ein bestimmtes Buch möglichst günstig kaufen möchte. Prinzipiell kann es natürlich viele Verkäufer und viele Käufer geben, sodass verschiedene Marktsituationen simuliert werden können. Für den deutschen Markt ist dieses Anwendungsbeispiel allerdings nicht relevant, denn hier gilt die Buchpreisbindung. Die einzelnen Klassen, deren Aufbau und Verwendung werden im Folgenden anhand der Quelltextauszüge vorgestellt.

BookSellerAgent Das Grundgerüst der Klasse `BookSellerAgent` ist in dem folgenden Listing dargestellt. Zu den Methoden und inneren Klassen sind nur die Signaturen angegeben. Die detaillierte Implementierung hierzu wird in weiteren Listings dargestellt.

```
public class BookSellerAgent extends Agent {

 // class variables
 private Hashtable catalogue;
 private BookSellerGui myGui;

 // methods
```

```
  protected void setup() { }
  protected void takeDown() { }
  private void register() { }
  private void deregister() { }
  public void updateCatalogue(final String title, final int price) { }

  // inner classes
  private class OfferRequestsServer extends CyclicBehaviour { }
  private class PurchaseOrdersServer extends CyclicBehaviour { }

}
```

Listing 5.3: BookSellerAgent: Grundgerüst – in Anlehnung an [Bel+17]

Die Klasse `BookSellerAgent` enthält zwei Klassenvariablen. Das Attribut `catalogue` ist eine Hashtable, in der die zu verkaufenden Bücher gespeichert werden. Das Attribut `myGui` ist das Objekt zu einer grafischen Benutzeroberfläche (GUI), mit der während der Laufzeit Bücher zum Katalog hinzugefügt werden können. Die Methoden `setup()` und `takeDown()` wurden bereits erwähnt. Sie benutzen jeweils die privaten Methoden `register()` und `deregister()`, um den Service des Buchverkaufs in den Gelben Seiten Directory Facilitator (DF) an- bzw. abzumelden. Die öffentliche Methode `updateCatalogue()` wird für die GUI benötigt. Die beiden privaten, inneren Klassen `OfferRequestServer` und `PurchaseOrderServer` werden gebraucht, um das zugehörige Verhalten abzubilden, nämlich Bücher anzubieten und den Verkauf abzuschließen. Betrachten wir nun die einzelnen Methoden bzw. inneren Klassen etwas detaillierter.

```
protected void setup() {

  // printout a welcome message
  System.out.println("Seller-agent "+getAID().getName()+" is ready.");

  // create the catalogue
  catalogue = new Hashtable();

  // create and show the GUI
  myGui = new BookSellerGui(this);
  myGui.showGui();

  // register the book-selling service in the yellow pages
  register();

  // add the behaviour serving queries from buyer agents
  addBehaviour(new OfferRequestsServer());

  // add the behaviour serving purchase orders from buyer agents
  addBehaviour(new PurchaseOrdersServer());

}
```

Listing 5.4: BookSellerAgent.setup() – in Anlehnung an [Bel+17]

Immer wenn ein neuer `BookSellerAgent` erzeugt wird, wird die Methode `setup()` ausgeführt. Zunächst meldet sich der Agent mit Namen. Dann wird eine neue, leere Hashtable erzeugt. Die grafische Benutzeroberfläche wird erstellt und angezeigt. Der Service dieses Agenten wird registriert und die beiden Verhalten werden hinzugefügt.

```
protected void takeDown() {

 // de-register from the yellow pages
 deregister();

 // close the GUI
 myGui.dispose();

 // printout a dismissal message
 System.out.println("Seller-agent "+getAID().getName()+" terminating.");

}
```

Listing 5.5: BookSellerAgent.takeDown() – in Anlehnung an [Bel+17]

Wenn der `BookSellerAgent` zerstört wird, müssen die Ressourcen wieder freigegeben werden. Dies geschieht in der Methoden `takeDown()`, in der zunächst der Service wieder abgemeldet, dann die grafische Benutzeroberfläche geschlossen und schließlich noch eine entsprechende Meldung ausgegeben wird.

```
private void register() {
 DFAgentDescription dfd = new DFAgentDescription();
 dfd.setName(getAID());
 ServiceDescription sd = new ServiceDescription();
 sd.setType("book-selling");
 sd.setName("JADE-book-trading");
 dfd.addServices(sd);
 try {
  DFService.register(this, dfd);
 } catch (FIPAException fe) {
  fe.printStackTrace();
 }
}
```

Listing 5.6: BookSellerAgent.register() – in Anlehnung an [Bel+17]

Die Registrierung läuft folgendermaßen ab: Zunächst gibt der Agent als Namen seine AID an. Als Dienstbeschreibung wird der Typ *book-selling* und als Dienstname *JADE-book-trading* angegeben. Eventuell auftretende Fehler bei der Registrierung werden mittels `try-catch`-Block aufgefangen und die Fehlermeldung als `StackTrace` ausgegeben.

```
private void deregister() {
 try {
  DFService.deregister(this);
 } catch (FIPAException fe) {
  fe.printStackTrace();
 } // try, catch
}
```

Listing 5.7: BookSellerAgent.deregister() – in Anlehnung an [Bel+17]

Die De-Registrierung des Agenten bzw. des Dienstes des Agenten ist noch einfacher als die Registrierung. Hierzu muss nur die entsprechende Methode aufgerufen werden und das Agent-Objekt (`this`) als Argument übergeben werden. Eventuell auftretende Fehler werden wieder mittels `try-catch`-Block aufgefangen und die Fehlermeldung als

StackTrace ausgegeben.

```
public void updateCatalogue(final String title, final int price) {
 addBehaviour(new OneShotBehaviour() {
  public void action() {
   catalogue.put(title, new Integer(price));
   System.out.println(title+" inserted into catalogue. Price = "+price);
  } // action()
 } ); // addBehaviour()
}
```

Listing 5.8: BookSellerAgent.updateCatalogue() – in Anlehnung an [Bel+17]

Mit der Methode updateCatalogue() ist es möglich, neue Bücher zum Katalog hinzuzufügen. Zu jedem Buch wird ein Titel und ein Preis benötigt, wobei der Preis der Einfachheit halber als ganze Zahl verarbeitet und gespeichert wird. Intern wird das Hinzufügen von Büchern über eine entsprechende Aktion eines *Behaviours* realisiert.

```
private class OfferRequestsServer extends CyclicBehaviour {
 public void action() {
  MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.CFP);
  ACLMessage msg = myAgent.receive(mt);
  if (msg != null) {
   // CFP Message received. Process it
   String title = msg.getContent();
   ACLMessage reply = msg.createReply();
   Integer price = (Integer) catalogue.get(title);
   if (price != null) {
    // The requested book is available for sale. Reply with the price
    reply.setPerformative(ACLMessage.PROPOSE);
    reply.setContent(String.valueOf(price.intValue()));
   } else {
    // The requested book is NOT available for sale.
    reply.setPerformative(ACLMessage.REFUSE);
    reply.setContent("not-available");
   } // if, else [price]
   myAgent.send(reply);
  } else {
   block();
  } // if, else [msg]
 } // action()
}
```

Listing 5.9: BookSellerAgent: inner class OfferRequestsServer – in Anlehnung an [Bel+17]

Die innere Klasse OfferRequestsServer prüft die Anfrage des BookBuyerAgent und schickt eine Antwort. Sie ist genau genommen ein zyklisches Verhalten. Zunächst wird die Nachricht an diesen Agenten gefiltert, die gemäß des ACL-Standards die Absicht (*performative*) CFP für *call-for-proposal* gesendet hat (vgl. Tab. 2.1 in Kap. 2.3.2). Wie wir später noch sehen werden, benutzt diese Absicht der BookBuyerAgent bei seiner Buch-Anfrage. Wenn es also eine solche Nachricht gibt, dann wird diese weiter analysiert. Als Nachrichtentext wird der Buchtitel erwartet. Im Katalog wird nachgeschlagen, wie der Preis zu diesem Titel ist. Falls das Buch gar nicht im Katalog vorhanden sein sollte, existiert auch kein zugehöriger Preis. In diesem Fall wird die Absicht REFUSE und der Nachrichtentext *not-available* zurückgesendet. Im positiven Fall wird dagegen die Absicht PROPOSE mit dem Preis als Nachrichtentext versendet. Sollte es aktuell keine CFP-Nachricht für den BookSellerAgent

geben, dann wird dieses Verhalten zunächst blockiert. Damit ist gemeint, dass die Aktion erst dann wieder ausgeführt wird, falls es eine neue Nachricht für diesen Agenten gibt. Denn der `OfferRequestsServer` ist ja als endloses zyklisches Verhalten implementiert, welches sehr viel Rechenzeit benötigen würde, wenn ständig nach neuen Nachrichten abgefragt werden müsste.

```
private class PurchaseOrdersServer extends CyclicBehaviour {
 public void action() {
  MessageTemplate mt = MessageTemplate.MatchPerformative(ACLMessage.
      ACCEPT_PROPOSAL);
  ACLMessage msg = myAgent.receive(mt);
  if (msg != null) {
   // ACCEPT_PROPOSAL Message received. Process it
   String title = msg.getContent();
   ACLMessage reply = msg.createReply();
   Integer price = (Integer) catalogue.remove(title);
   if (price != null) {
    reply.setPerformative(ACLMessage.INFORM);
    System.out.println(title+" sold to "+msg.getSender().getName());
   } else {
    // The requested book has been sold to another buyer in the meanwhile
    reply.setPerformative(ACLMessage.FAILURE);
    reply.setContent("not-available");
   } // if, else [price]
   myAgent.send(reply);
  } else {
   block();
  } // if, else [msg]
 } // action()
}
```

Listing 5.10: BookSellerAgent: inner class PurchaseOrdersServer – in Anlehnung an [Bel+17]

Die innere Klasse `PurchaseOrdersServer` wird zum Verkauf des Buches benötigt. Auch diese Klasse ist als zyklisches Verhalten implementiert und hat einen sehr ähnlichen Aufbau wie die Klasse `OfferRequestsServer`. Zunächst wird wieder die Nachricht an diesen Agenten gefiltert, allerdings muss diesmal die Absicht `ACCEPT_PROPOSAL` gesendet worden sein. Im Nachrichtentext steht wieder der Buchtitel. Wird ein Preis zu dem Buchtitel im Katalog gefunden, dann wird das Buch zu diesem Preis an den `BookBuyerAgent` verkauft. Eine entsprechende Information (Absicht `INFORM`) wird zurückgesendet, ansonsten wird ein Fehler (Absicht `FAILURE`) und die Nachricht *not-available* verschickt. Zum `BookSellerAgent` gehört noch eine weitere Klasse `BookSellerGui`, die benutzt werden kann, damit der Verkäufer über eine grafische Benutzeroberfläche (GUI) Bücher und Preise zum Katalog hinzufügen kann. Auf den Aufbau und die Programmierung dieser Hilfsklasse wird an dieser Stelle verzichtet.

BookBuyerAgent: Der `BookBuyerAgent` ist der Gegenspieler vom `BookSellerAgent`. Er sucht ein bestimmtes Buch und möchte dieses möglichst günstig kaufen. Wir nehmen an, dass es mehrere Buchhändler (Verkäufer) gibt, die das gesuchte Buch zu unterschiedlichen Preisen anbieten. Das Grundgerüst zu der Klasse `BookBuyerAgent` ist in dem folgenden Listing dargestellt.

```
public class BookBuyerAgent extends Agent {

 // class variables
 private String targetBookTitle;
 private AID[] sellerAgents;

 // methods
 protected void setup() { }
 protected void takeDown() { }

 // inner classes
 private class RequestPerformer extends Behaviour { }

}
```

Listing 5.11: BookBuyerAgent: Grundgerüst – in Anlehnung an [Bel+17]

Das erste Attribut ist der Buchtitel, den der Käufer kaufen möchte. Das zweite Attribut ist ein Vektor zu den AIDs der Buchverkäufer. Neben den bekannten Methoden wird eine innere Klasse benötigt, um das Verhalten des Käufers abzubilden. Die `setup()`-Methode soll die beiden Attribute initialisieren und das Verhalten definieren.

```
protected void setup() {

 // printout a welcome message
 System.out.println("Buyer-agent "+getAID().getName()+" is ready.");

 // get the title of the book to buy as a start-up argument
 Object[] args = getArguments();
 if (args != null && args.length > 0) {

  targetBookTitle = (String) args[0];
  System.out.println("Target book is "+targetBookTitle);

  // add a TickerBehaviour that schedules a request to seller agents every minute
  addBehaviour(new TickerBehaviour(this, 60000) {
   protected void onTick() {
    System.out.println("Trying to buy "+targetBookTitle);

    // update the list of seller agents
    DFAgentDescription template = new DFAgentDescription();
    ServiceDescription sd = new ServiceDescription();
    sd.setType("book-selling");
    template.addServices(sd);
    try {
     DFAgentDescription[] result = DFService.search(myAgent, template);
     System.out.println("Found the following seller agents:");
     sellerAgents = new AID[result.length];
     for (int i = 0; i < result.length; ++i) {
      sellerAgents[i] = result[i].getName();
      System.out.println(sellerAgents[i].getName());
     } // for
    } catch (FIPAException fe) {
     fe.printStackTrace();
    } // try, catch

    // perform the request
    myAgent.addBehaviour(new RequestPerformer());
```

```
    } // onTick()
   } ); // addBehaviour()
  } else {
   // make the agent terminate
   System.out.println("No target book title specified");
   doDelete();
  } // if, else [args]

}
```

Listing 5.12: BookBuyerAgent.setup() – in Anlehnung an [Bel+17]

Zunächst meldet sich der Agent mit seinem Namen. Der Buchtitel wird dem Agenten als Start-Argument mitgegeben. Fehlt dieses Argument, dann wird der Agent direkt wieder terminiert. Das Füllen des Vektors mit den Verkäufern ist etwas komplizierter. Hierzu wird der Gelbe-Seiten-Dienst verwendet. Da aber zur Laufzeit auch neue Verkäufer hinzukommen können, muss dies als wiederkehrendes Verhalten implementiert werden. In diesem Fall wird das Verhalten von der Klasse `TickerBehaviour` abgeleitet und einmal pro Minute (entspricht 60.000 Millisekunden) die Gelben Seiten abgefragt, und zwar nach dem Typ *book-selling*, den der `BookSellerAgent` bei seiner Registrierung angegeben hat (s.o.). Der Vektor AID wird dann immer neu generiert und mit den Verkäufern gefüllt. Dadurch werden auch Verkäufer gelöscht, die mittlerweile das System wieder verlassen haben. Schließlich wird noch das Verhalten `RequestPerformer` hinzugefügt, welches in der gleichnamigen inneren Klasse implementiert ist.

```
private class RequestPerformer extends Behaviour {

 private AID bestSeller; // the agent who provides the best offer
 private int bestPrice; // the best offered price
 private int repliesCnt = 0; // the counter of replies from seller agents
 private MessageTemplate mt; // the template to receive replies
 private int step = 1;

 public void action() {
  switch (step) {

   case 1: // send the cfp to all sellers
    ACLMessage cfp = new ACLMessage(ACLMessage.CFP);
    for (int i = 0; i < sellerAgents.length; ++i) {
     cfp.addReceiver(sellerAgents[i]);
    } // for
    cfp.setContent(targetBookTitle);
    cfp.setConversationId("book-trade");
    cfp.setReplyWith("cfp"+System.currentTimeMillis()); // Unique value
    myAgent.send(cfp);
    // prepare the template to get proposals
    mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-
        trade"), MessageTemplate.MatchInReplyTo(cfp.getReplyWith()));
    step = 2;
    break;

   case 2: // receive all proposals/refusals from seller agents
    ACLMessage reply = myAgent.receive(mt);
    if (reply != null) {
     // reply received
     if (reply.getPerformative() == ACLMessage.PROPOSE) {
      // this is an offer
      int price = Integer.parseInt(reply.getContent());
```

```
        if (bestSeller == null || price < bestPrice) {
         // this is the best offer at present
         bestPrice = price;
         bestSeller = reply.getSender();
        } // if [bestSeller]
       } // if [reply.getPerformative()]
       repliesCnt++;
       if (repliesCnt >= sellerAgents.length) {
        // we received all replies
        step = 3;
       } // if [repliesCnt]
      } else {
       block();
      } // if, else [reply]
      break;

     case 3: // send the purchase order to the seller that provided the best offer
      ACLMessage order = new ACLMessage(ACLMessage.ACCEPT_PROPOSAL);
      order.addReceiver(bestSeller);
      order.setContent(targetBookTitle);
      order.setConversationId("book-trade");
      order.setReplyWith("order"+System.currentTimeMillis());
      myAgent.send(order);
      // prepare the template to get the purchase order reply
      mt = MessageTemplate.and(MessageTemplate.MatchConversationId("book-
          trade"), MessageTemplate.MatchInReplyTo(order.getReplyWith()));
      step = 4;
      break;

     case 4: // receive the purchase order reply
      reply = myAgent.receive(mt);
      if (reply != null) {
       // purchase order reply received
       if (reply.getPerformative() == ACLMessage.INFORM) {
        // purchase successful. We can terminate
        System.out.println(targetBookTitle+" successfully purchased from
            agent "+reply.getSender().getName());
        System.out.println("Price = "+bestPrice);
        myAgent.doDelete();
       } else {
        System.out.println("Attempt failed: requested book already sold.");
       } // if, else [reply.getPerformative()]
       step = 5;
      } else {
       block();
      } // if, else [reply]
      break;

    } // switch
   } // action()

   public boolean done() {
    if (step == 3 && bestSeller == null) {
     System.out.println("Attempt failed: "+targetBookTitle+" not available
         for sale");
    } // if
    return ((step == 3 && bestSeller == null) || step == 5);
   } // done()

}
```

Listing 5.13: BookBuyerAgent: inner class RequestPerformer – in Anlehnung an [Bel+17]

Generell bildet die Klasse `RequestPerformer` das Verhalten des Käufers in mehreren Schritten ab. Im ersten Schritt werden alle Verkäufer über die Buchsuche informiert. Hierzu wird eine entsprechende Nachricht mit der Absicht `CFP` und dem Buchtitel als Text an alle Verkäufer gesendet. Damit die Antworten besser zugeordnet werden können, wird außerdem das Attribut `ReplyWith` gesetzt und die aktuelle Systemzeit in Millisekunden benutzt. Es folgt automatisch der zweite Schritt. In diesem werden die Antworten der Verkäufer dann analysiert. Interessant sind nur Nachrichten mit der Absicht `PROPOSE`, die den Verkaufspreis als Nachrichtentext enthalten. Dieser Preis wird nun mit dem aktuell besten Preis (`bestPrice`) verglichen, um den günstigsten Verkäufer (`bestSeller`) herauszufinden. Erst, wenn von allen Verkäufern ein Angebot oder eine Absage vorliegt, wird dieser Schritt abgeschlossen. In Schritt 3 wird dann dem günstigsten Verkäufer die Kaufnachricht mit der Absicht `ACCEPT_PROPOSAL` geschickt. Automatisch folgt dann der vierte und letzte Schritt. Hier wird die Antwort des günstigsten Verkäufers analysiert. Die Absicht muss `INFORM` sein, damit der Buchkauf erfolgreich war. In diesem Fall wird direkt `doDelete()` aufgerufen, weil das Ziel erreicht wurde und somit der `BookBuyerAgent` beendet werden kann. Es könnte aber auch sein, dass das Buch zwischenzeitlich bereits (an einen anderen Käufer) verkauft wurde (Absicht `FAILURE`). In beiden Fällen wird die Variable `step` auf 5 gesetzt. Das Verhalten wird in jedem Fall beendet, falls dieser Schritt 5 erreicht wurde. Es muss aber auch beendet werden, falls es das Buch bei keinem der Verkäufer zu kaufen gibt, d.h. es gilt: `step` $=$ 3 und `bestSeller` $=$ `null`. Genau diese Bedingung ist in der Methode `done()` auch implementiert.

Die Methode `takeDown` muss diesmal keine Ressourcen freigeben, sondern meldet nur das Beenden des Agenten.

```
protected void takeDown() {
  // printout a dismissal message
  System.out.println("Buyer-agent "+getAID().getName()+" terminating.");
}
```

Listing 5.14: BookBuyerAgent.takeDown() – in Anlehnung an [Bel+17]

5.3 Installation und Betrieb

JADE wird als ZIP-Archiv zum Herunterladen unter der URL http://jade.tilab.com/download/jade/ angeboten. Innerhalb dieses Archivs *JADE-all-4.5.0.zip* sind vier weitere Archive enthalten:

JADE-bin-4.5.0.zip	—	Ausführbares Java-Archiv
JADE-doc-4.5.0.zip	—	Dokumentation, API
JADE-examples-4.5.0.zip	—	Beispiele
JADE-src-4.5.0.zip	—	Java-Quelltext

Im ersten Archiv befindet sich ein ausführbares Java-Archiv in Form eine Datei jade.jar. Mit Hilfe dieser Datei kann das Multiagentensystem gestartet werden. Hierzu sollte zunächst der Pfad zu dieser Datei als Systemvariable `CLASSPATH` gesetzt werden. Bei dem Betriebssystem Windows kann dies unter der Systemsteuerung eingestellt werden. Alternativ kann dies auch über die Kommandozeile (cmd) realisiert werden:

```
set JADE_HOME=C:\Jade
set CLASSPATH=%JADE_HOME%\lib\jade.jar;%JADE_HOME%\classes
```

In diesem Beispiel wurde JADE direkt unter dem Laufwerk `C` installiert. Die ausführbare Bibliothek befindet sich im Unterverzeichnis `lib` und die kompilierten Java-Dateien liegen im Unterverzeichnis `classes`. Mit dem Aufruf

```
java jade.Boot <name>:<class>
```

kann nun ein Agent gestartet werden, wobei mindestens der Name und die Java-Klasse angeben werden müssen. Der Agent wird dann innerhalb eines Standard-Containers (*main container*) als Java-Thread ausgeführt. Falls es Probleme beim Start gibt und die Klasse des Agenten nicht vom System gefunden wird, kann man den Klassenpfad auch direkt mitangeben:

```
java -cp <classpath> jade.Boot <name>:<class>
```

Für die Beispielanwendung aus Kap. 5.2 kann der `BookBuyerAgent` mittels des Kommandos

```
java -cp C:\Jade\lib\jade.jar;C:\Jade\classes; jade.Boot -gui
 Max:examples.bookTrading.BookBuyerAgent("Der Herr der Ringe")
```

gestartet werden. Dabei möchte der Agent *Max* das Buch mit dem Titel *Der Herr der Ringe* kaufen. Die Klasse `BookBuyerAgent` befindet sich im Paket `examples.bookTrading`, welche als Pfad vorangestellt werden muss. Mit der Option `-gui` wird außerdem eine grafische Benutzeroberfläche (engl. *Graphical User Interface (GUI)*) gestartet, mit der das Multiagentensystem gesteuert werden kann (siehe Abb. 5.2).

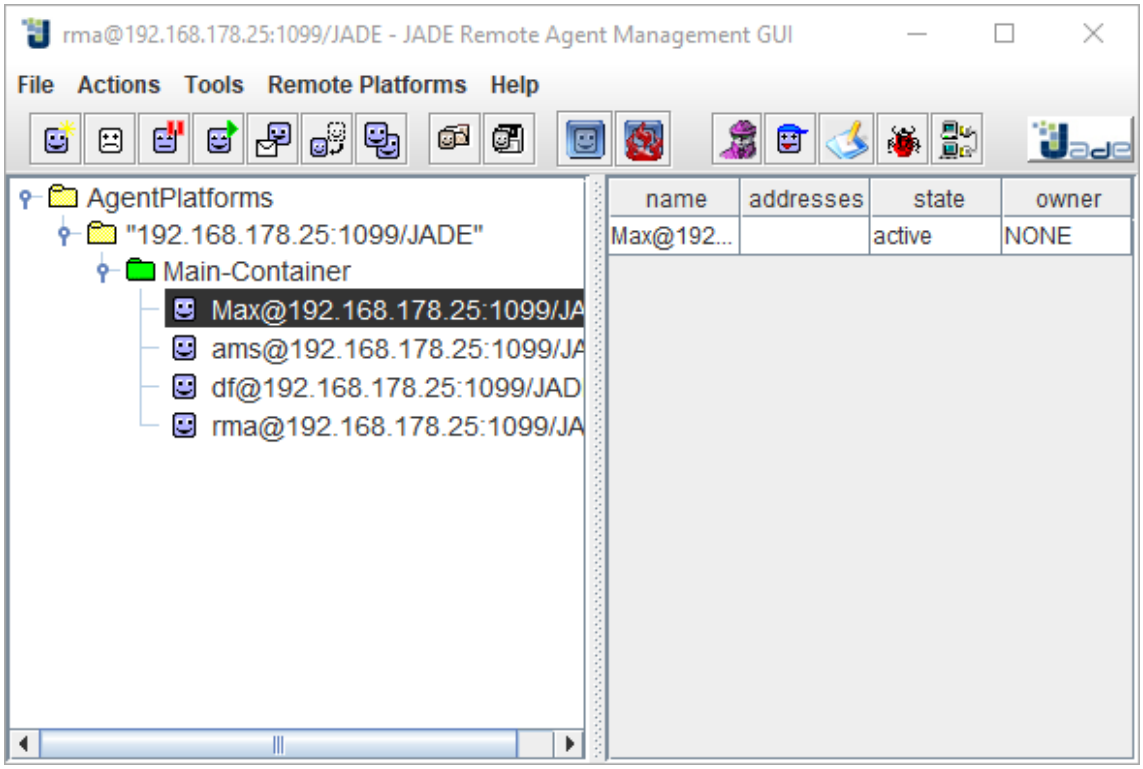

Abbildung 5.2: Grafische Benutzeroberfläche zu JADE

Über diese GUI lassen sich auch weitere Agenten wie z.B. den `BookSellerAgent` hinzufügen (siehe Abb. 5.3).

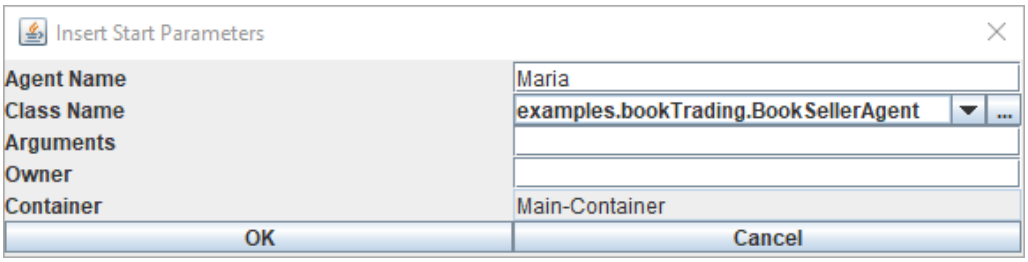

Abbildung 5.3: Anlegen eines BookSeller-Agenten

Nachdem der `BookSellerAgent` mit dem Namen Maria angelegt wurde, öffnet sich ein weiteres Fenster, um Bücher in den Katalog des Verkäufers einzupflegen (vgl. Kap. 5.2, siehe Abb. 5.4).

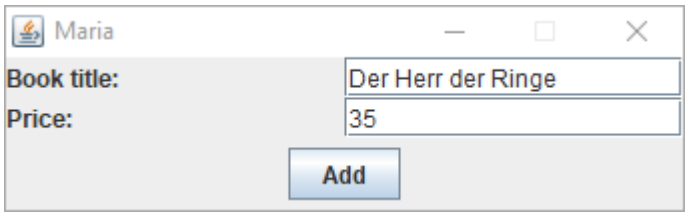

Abbildung 5.4: Hinzufügen eines Buches zum Katalog des Verkäufers

In der Konsole werden die Ausgaben der Java-Befehle `System.out.println()` ausgegeben. Hier sieht man bereits grob den Ablauf des Buchverkaufs (siehe Abb. 5.5).

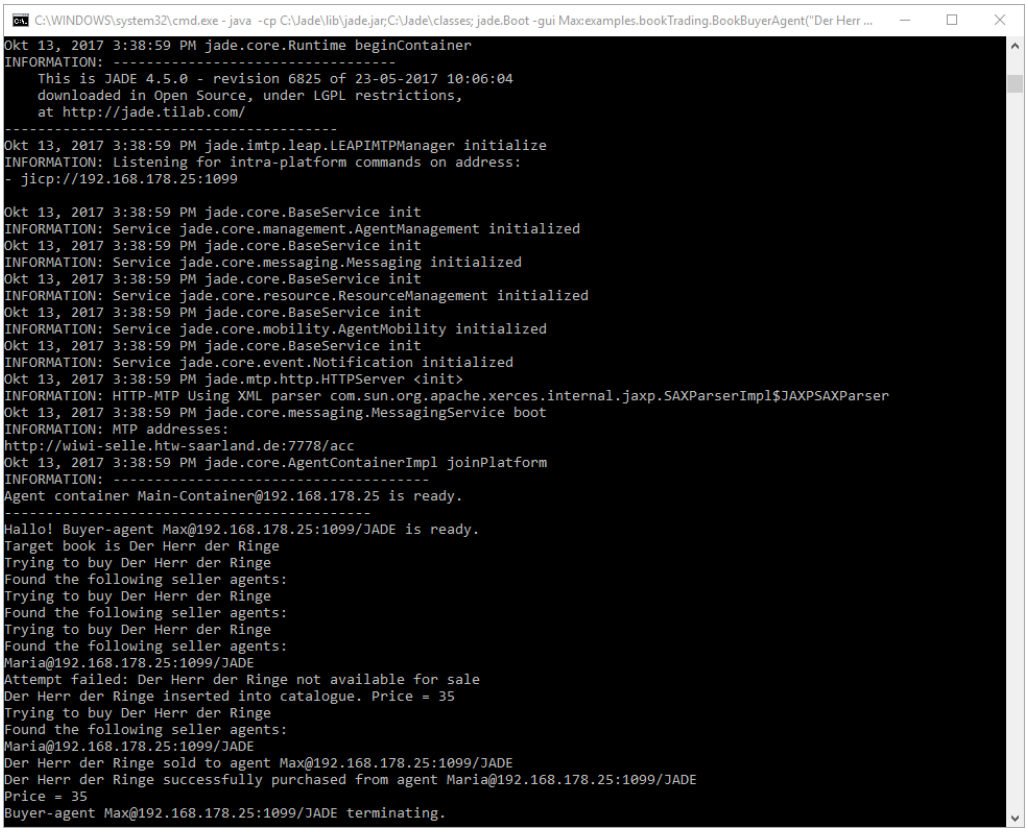

Abbildung 5.5: Konsole mit den Ausgaben der Agenten

Die JADE-Plattform bietet einige Tools, die einem die Arbeit erleichtern, u.a. der sogenannte Sniffer-Agent. Dieser kann ebenfalls über die GUI gestartet werden und mit ihm kann die Kommunikation ausgewählter Agenten überwacht werden. Für das Anwendungsbeispiel ist dies in Abb. 5.6 dargestellt. In diesem Beispiel wurde noch ein weitere `BookSellerAgent` mit dem Namen Marion erzeugt, der das gesuchte Buch etwas günstiger als Maria anbietet.

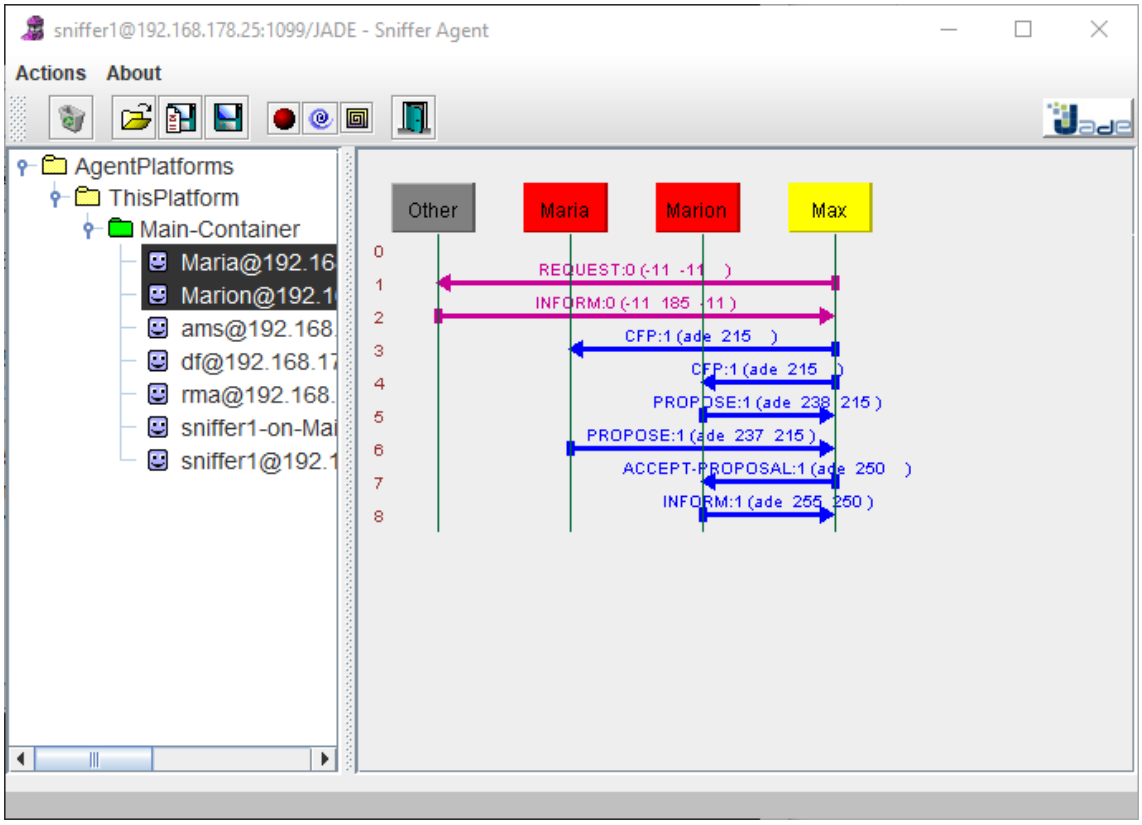

Abbildung 5.6: Sniffer-Agent mit den Kommunikationspfaden

In der Ausgabe der Konsole wird bestätigt, dass Max nun das Buch von Marion gekauft hat (siehe Abb. 5.7). Am Ende wird der Agent Max terminiert, weil er seine Aufgabe erfolgreich erledigt hat. Die Buchverkäufer dagegen können ihre Bücher theoretisch auch noch anderen Agenten zum Kauf anbieten, deshalb bleiben sie im System.

```
Buyer-agent Max@192.168.178.25:1099/JADE terminating.
Hallo! Buyer-agent Max@192.168.178.25:1099/JADE is ready.
Target book is Der Herr der Ringe
Trying to buy Der Herr der Ringe
Found the following seller agents:
Maria@192.168.178.25:1099/JADE
Marion@192.168.178.25:1099/JADE
Der Herr der Ringe sold to agent Max@192.168.178.25:1099/JADE
Der Herr der Ringe successfully purchased from agent Marion@192.168.178.25:1099/JADE
Price = 32
Buyer-agent Max@192.168.178.25:1099/JADE terminating.
```

Abbildung 5.7: Konsolenausgabe zum erweiterten Beispiel: Zwei Verkäufer

Diese Demonstration verdeutlicht den Einsatz des MAS-Frameworks JADE. Das Anwendungsbeispiel ist bewusst einfach gehalten, um zunächst erste Erfahrungen mit der Software zu machen und diese zu dokumentieren. Viele Möglichkeiten, die das Framework

oder deren Erweiterungen bieten, wurden noch nicht betrachtet. Trotzdem konnten bereits interessante Erkenntnisse gewonnen werden, die im folgenden Abschnitt kurz präsentiert werden.

5.4 Fazit

JADE ist ein Framework zu einem Multiagentensystem, welches FIPA-konform in Java entwickelt wurde und dem Anwender eine Middleware und zahlreiche Tools wie bspw. eine grafische Benutzeroberfläche oder den Sniffer-Agenten zur Verfügung stellen, um damit effizient zu arbeiten. Die Installation ist einfach und es gibt zahlreiche Dokumentationen für den Einstieg der Administration der Plattform oder Programmierung der Agenten. Voraussetzung ist hier allerdings, dass man über sehr gute Java-Kenntnisse verfügt. Spezielle Möglichkeiten wie bspw. der Einsatz von Ontologien, umfangreichere Konversationen der Agenten oder das Lösen von komplexen Aufgaben wurden im Rahmen dieser Arbeit noch nicht ausprobiert. Auch die besondere Eigenschaft von JADE, Agenten auf andere Plattformen zu verschieben (Agenten-Mobilität), wurde noch nicht getestet. Das Belief-Desire-Interaction (BDI)-Konzept ist selbst kein Bestandteil von JADE. Diese Art von Agenten lassen sich aber ebenfalls implementieren, wenn hierzu bspw. die Erweiterung Jadex eingesetzt wird. Bei dem Einsatz von JADE und der Beispielanwendung hat sich jedoch ein nicht zu vernachlässigender Nachteil deutlich gezeigt: die schlechte *Performance*, d.h. die Kommunikation zwischen den Agenten erfolgt sehr langsam. Möglicherweise liegt dies am XML-Format der Nachrichten. Ggf. ist es sehr aufwändig, dieses zu verarbeiten. Die Entwicklung von geschäftskritischen Anwendungen von Multiagentensystemen mit dem Framework JADE ist deshalb vermutlich nicht möglich. Insbesondere Anwendungen im Bereich Industrie 4.0 oder Autonomes Fahren, die sehr zeitkritisch und rechenzeitintensiv sind, können also mit JADE nicht realisiert werden. Anwendungen im Bereich von SmartHome oder dem Internet-Handel sind dagegen sicherlich möglich.

Das Ziel dieser Arbeit war es, ein Framework für ein Multiagentensystem zu finden, welches in Forschung und Lehre an der htw saar eingesetzt werden kann. Dieses Ziel wurde erreicht. JADE ist ein solches Framework. Allerdings müssen Einschränkungen gemacht werden. Aus *Performance*-Gründen können mit diesem Framework keine zeitkritischen Anwendungen erstellt werden. Das schränkt die Bereiche in der angewandten Forschung etwas ein. Allerdings lassen sich hier trotzdem noch genügend Themen finden, in denen JADE zum Einsatz kommen kann. Ideen werden hierzu im Ausblick des nächsten Kapitels vorgestellt. Der Einsatz in der Lehre ist ebenfalls möglich. Im Rahmen des Modul BBWL-571 Software Engineering lernen Studierende des Bachelor-Studiengangs Betriebswirtschaft mit dem Schwerpunkt Wirtschaftsinformatik Java als objektorientierte Programmiersprache kennen. Im Anschluss können interessierte Studierende dann das Modul BBWL-622 Projekt 2 belegen. Im Rahmen von Wirtschaftsinformatik-Projekten soll hier das erlernte Wissen praktisch durch Teamarbeit angewendet werden. Mögliche Themen zu diesen Projekten können somit auch aus dem Bereich Multiagentensysteme stammen.

6 Zusammenfassung und Ausblick

Multiagentensysteme werden zukünftig eine wichtige Rolle im Rahmen der Digitalisierung in Verbindung mit Künstlicher Intelligenz spielen. Die Software-Agenten in diesen Systemen werden immer intelligenter und in Zusammenarbeit mit anderen Agenten lassen sich auch komplexe Aufgaben lösen. Das Ziel dieser Arbeit war das Finden eines geeigneten Frameworks für Multiagentensysteme, das bei der Entwicklung und Simulation von typischen Anwendungen eingesetzt werden kann.

In Kap. 2 wurden die theoretischen Grundlagen gelegt, die wesentlichen Begriffe definiert und Zusammenhänge zwischen diesen dargestellt. Die wesentlichen Vorarbeiten zum Thema Agenten stammen von Russel und Norvig [RN03]. Sie haben Agenten hinsichtlich ihres inneren Aufbaus und ihres Verhaltens klassifiziert und dabei folgende Typen definiert: einfacher Reflex-Agent, modellbasierte Reflex-Agent, zielbasierter Agent, nutzenbasierter Agent und lernender Agent. Der Lernende Agent entspricht dem deliberalen Agenten von Wooldridge [Woo09]. Er nimmt seine Umwelt über Sensoren war und kann diese wiederum über Aktoren beeinflussen. Er verfügt aber auch über ein Modell dieser Welt und kann adaptiv seinen internen Zustand anpassen. Aufgrund von Ziel- und Planer-Komponenten lassen sich auch rationale Entscheidungsprozesse abbilden. Somit kann dieser Typ von Agent auch als intelligenter bzw. kognitiver Agent bezeichnet werden. Das Konzept des BDI-Agenten setzt genau hier an. Die Komponenten *Beliefs* (Annahmen), *Desires* (Ziele) und *Intentions* (Absichten) beschreiben den internen Zustand des Agenten. Werden Agenten nun zu einem Multiagentensystem (MAS) zusammengeschlossen, dann sind Kommunikation und Koordination zentrale Aspekte, deren Funktionsweise definiert werden muss. Die Foundation for Intelligent Physical Agents (FIPA) hat zur Standardisierung eine Referenzarchitektur entwickelt. Dort ist sehr genau spezifiziert, wie bspw. die Kommunikation per Agent Communication Language (ACL) abläuft und Nachrichten aussehen.

Standards im IT-Bereich sorgen für Investitionssicherheit. In Kap. 3 wurden Frameworks vorgestellt, die Standards wie FIPA und BDI verwenden. Der Einsatz eines Frameworks hat generell viele Vorteile, u.a. muss durch dessen Verwendung nicht alles komplett neu entwickelt werden. Die Software muss auch einige zusätzliche Anforderungen erfüllen: genereller Einsatzzweck, Plattformunabhängigkeit, Quelloffenheit für flexible Weiterentwicklung. Letztendlich wurden aus zahlreichen Frameworks fünf herausgefiltert, die in Kap. 4 dann näher analysiert werden: JADE, Jadex, Jason, SeSAm und ZEUS. Anhand der Merkmale Aktualität, Funktionsumfang, Hilfe bzw. Dokumentation, Reputation und Präsentation wurden diese bewertet und miteinander verglichen. Hierzu wurde eine Nutzwertanalyse durchgeführt, wobei die fünf Merkmale der Einfachheit halber gleich gewichtet wurden. JADE erzielte bei diesem Vergleich den höchsten Nutzwert. Dieses Framework wurde von Telecom Italia entwickelt und wird seit dem Jahr 2000 als *Open Source* Lösung angeboten und kontinuierlich weiterentwickelt. Es gibt eine große *Community*, sehr gute Dokumentationen und die aktuelle Version ist nur wenige Monate alt.

In Kap. 5 wurde JADE dann detaillierter hinsichtlich des Einsatzes betrachtet. Es wurden die Konzepte erläutert, wie Agenten in der objektorientierten Programmiersprache

Java erstellt werden können. Eine einfache Beispielanwendung mit Kontext Buchhandel wurde ebenfalls vorgestellt. Ein `BookBuyerAgent` möchte ein bestimmtes Buch kaufen, ein oder mehrere `BookSellerAgents` verkaufen Bücher. Der Kommunikationsablauf mittels ACL-Nachrichten wurde dabei erläutert. Die Installation, Konfiguration und der Betrieb von JADE wurde dann exemplarisch anhand dieser Beispielanwendung auf dem Betriebssystem Windows erklärt. Generell kann diese aber auch unter anderen Betriebssystemen ausgeführt werden. Das Framework hat alle Erwartungen erfüllt, lediglich die Performance war nicht optimal. Da die Kommunikation der Agenten asynchron erfolgt und die Agenten als Java-Threads ausgeführt werden, ergaben sich teilweise Wartezeiten im Sekundenbereich während der Agentenkommunikation. Somit sollte JADE nicht für zeitkritische Anwendungen eingesetzt werden.

Das Ziel dieser Arbeit wurde somit erreicht. JADE ist ein Multiagentensystem, das in Forschung und Lehre an der htw saar eingesetzt werden kann, allerdings mit der oben genannten Einschränkung. Zukünftig sollte JADE noch intensiver getestet werden. Viele Möglichkeiten wurden noch gar nicht ausprobiert. Hierzu gehört bspw. das Verschieben von Agenten auf andere Plattformen, insbes. auf mobile Endgeräte wie Smartphones. Die Agenten sollten in ihren Nachrichten Ontologien verwenden. Die Kommunikation könnte wesentlich komplexer werden und auch die Zusammenarbeit lässt sich noch genauer untersuchen [Wei13]. Des Weiteren sollte auch das BDI-Konzept eingesetzt werden. Hierfür kann bspw. die Erweiterung Jadex verwendet werden. Schließlich sollen die Agenten selbst möglichst intelligent im Sinne der künstlichen Intelligenz werden. Dabei könnten spezielle Lernverfahren wie Reinforcement Learning und Deep Learning eingesetzt werden. All diese Themen lassen sich auch im Rahmen von Projektarbeiten (Modul BBWL-622) oder Abschlussarbeiten durch Studierende bearbeiten. Hierzu könnten ggf. auch Raspberry-Pi-Computer benutzt werden. Auch auf diesen Kleinstcomputern kann JADE installiert und ausgeführt werden. Mit Hilfe von Kompensationsmitteln wurden mehrere Raspberry-Pis angeschafft und ein Cluster aufgebaut (vgl. Abb. 6.1), der im Rahmen und zur Verbesserung der Lehre eingesetzt wird.

Abbildung 6.1: Raspberry-Pi-Cluster

Das Multiagentensystem JADE kann zukünftig auch in der angewandten Forschung verwendet werden. Wie bereits dargestellt, sind zeitkritische Anwendungen aber nicht möglich, d.h. es ist nicht sinnvoll, Anwendungen für den Bereich Industrie 4.0 oder dem autonomen Fahren zu entwickeln. In anderen Branchen ist dagegen Potenzial für JADE vorhanden, insbes. im Bereich Internet of Things (IoT) oder Smart Home können entsprechende Anwendungen erstellt werden. Zwei Ideen werden im Folgenden kurz vorgestellt.

Tourismus: Dynamic Packaging Unter *Dynamic Packaging* versteht man die dynamische Zusammenstellung einer Reise aus verschiedenen Bausteinen wie bspw. einem Flug, einem Hotel, einem Transfer vom jeweiligen Flughafen zum Hotel und zurück zum Airport, oder auch einem Mietwagen usw. Hierfür gibt es bereits Online-Portale im Internet, mit deren Hilfe der Kunde seine persönliche Wunschreise zusammenstellen kann. Dies erfordert allerdings immer noch sehr viel Zeit und Interaktion mit dem Online-Portal. Ein Multiagentensystem könnte diese Aufgabe für den Kunden übernehmen und sinnvolle Ergebnisse als Vorschläge generieren, die der Kunde dann bei Gefallen nur noch annehmen braucht. Das MAS kann derart gestaltet werden, dass jeder Agent eine Spezialaufgabe bekommt: `FlugAgent`, `HotelAgent`, `MietwagenAgent`, `TransferAgent`, `VersicherungAgent`, `RestaurantAgent`, `VeranstaltungAgent` usw. Eine Zusammenarbeit innerhalb der Fakultät für Wirtschaftswissenschaften der htw saar wäre hier mit den Kollegen aus dem Bereich des Internationalen Tourismus Managements möglich, z.B. mit Prof. Dr. Achim Schröder.

Smart Home: AAL *Ambient Assisted Living (AAL)* umfasst Methoden, Konzepte, Systeme, Produkte und Dienstleistungen, die das alltägliche Leben älterer bzw. benachteiligter Menschen (z.B. gehbehinderte oder Demenzkranke) situationsabhängig unterstützen. Ziel ist es somit, dass diese Menschen möglichst lange in ihrer gewohnten Umgebung selbstständig leben können. Eine Smart-Home-Plattform kann auch als Multiagentensystem realisiert werden. Jedes Gerät (Lampe, Rollladen, Heizung, Kühlschrank, Herd, Rauchmelder, Multimedia usw.) ist dabei ein Agent. Es muss nicht zwangsläufig eine zentrale Steuerung in diesem System geben. Jeder Agent kann auch autonom agieren und sich gemäß den Wünschen und Anforderungen des Menschen anpassen. Dies setzt die Verwendung von Sensoren voraus. Die Daten, die dabei entstehen, sollten dann allerdings zentral gespeichert werden, damit alle Agenten bei Bedarf auch darauf zugreifen können. Denn diese Daten sind die Basis für selbstlernende Systeme, die für Komfort (Gewohnheit) und Sicherheit (Abweichungen) sorgen können. An der htw saar hat Prof. Dr. Wolfgang Langguth aus der Fakultät der Ingenieurswissenschaften bereits ein bekanntes und großes AAL-Netzwerk etabliert.

Quellenverzeichnis

[Alt+17] M. Altaweel, N. Collier, T. Howe, R. Najlis, M. North, M. Parker, E. Tatara und J. R. Vos. *Repast*. 2017. URL: http://repast.sourceforge.net/repast_3 (besucht am 24. 07. 2017).

[ArioJ] Aristoteles. *Metaphysik VII*. o. J.

[BCG07] F. Bellifemine, G. Caire und D. Greenwood. *Developing Multi-agent Systems with JADE*. 1. Aufl. Chichester, West Sussex, UK: John Wiley & Sons, 2007.

[Bel+17] F. Bellifemine, G. Caire, G. Rimassa, A. Poggi, F. Bergenti, T. Trucco und D. Gotta. *JADE*. 2017. URL: http://jade.tilab.com (besucht am 24. 07. 2017).

[CLT17] J. Collis, L. Lee und S. Thompson. *Zeus Agent Toolkit*. 2017. URL: https://sourceforge.net/projects/zeusagent (besucht am 24. 07. 2017).

[Con07] Car 2 Car Communication Consortium. *Manifesto: Overview of the C2C-CC System*. 2007. URL: https://www.car-2-car.org/index.php?id=31 (besucht am 28. 08. 2007).

[Dud17] Duden. *System*. 2017. URL: http://www.duden.de/rechtschreibung/System (besucht am 15. 07. 2017).

[FG17] E. Feess und R. Gillenkirch. *System*. 2017. URL: http://wirtschaftslexikon.gabler.de/Archiv/3210/system-v12.html (besucht am 15. 07. 2017).

[FIP01] FIPA. *RDF Content Language Specification*. 2001. URL: http://www.fipa.org/specs/fipa00011/XC00011B.html (besucht am 10. 08. 2001).

[FIP02a] FIPA. *ACL Message Structure Specification*. 2002. URL: http://www.fipa.org/specs/fipa00011/XC00011B.html (besucht am 12. 03. 2002).

[FIP02b] FIPA. *Communicative Act Library Specification*. 2002. URL: http://www.fipa.org/specs/fipa00037/SC00037J.html (besucht am 12. 03. 2002).

[FIP03] FIPA. *Glossary*. 2003. URL: http://www.pa.icar.cnr.it/cossentino/FIPAmeth/glossary.htm (besucht am 05. 11. 2003).

[FIP04] FIPA. *Agent Management Specification*. 2004. URL: http://www.fipa.org/specs/fipa00023/SC00023K.html (besucht am 18. 03. 2004).

[FIP17] FIPA. *Publicly Available Implementations of FIPA Specifications*. 2017. URL: http://www.fipa.org/resources/livesystems.html (besucht am 03. 08. 2007).

[Gro17] AOS Group. *JACK*. 2017. URL: http://www.agent-software.com.au/products/jack/ (besucht am 03. 08. 2017).

[HB17] J. F. Hübner und R. H. Bordini. *Jason*. 2017. URL: http://jason.sourceforge.net/wp/ (besucht am 03. 08. 2017).

[KB15] K. Kravari und N. Bassiliades. „A Survey of Agent Platforms“. In: *Journal of Artificial Societies and Social Simulation* 18.1 (2015), S. 11. URL: http://jasss.soc.surrey.ac.uk/18/1/11.html (besucht am 03. 08. 2017).

[KJ17] F. Klügl und R. Junges. *SeSAm*. 2017. URL: http://www.simsesam.de (besucht am 24. 07. 2017).

[LL92] M. Ljundberg und A. Lucas. „The OASIS Air Traffic Management System“. In: *Proceedings of the Second Pacific Rim International Conference on AI (PRICAI-92)*. Seoul, Südkorea, 1992.

[Luk+17] S. Luke, G. C. Balan, K. Sullivan und L. Panait. *Mason*. 2017. URL: http://cs.gmu.edu/~eclab/projects/mason (besucht am 24. 07. 2017).

[PBJ17] A. Pokahr, L. Braubach und K. Jander. *Active Components: Jadex*. 2017. URL: https://www.activecomponents.org (besucht am 03. 08. 2017).

[Pos07] S. Poslad. „Specifying Protocols for Multi-Agent Systems Interaction“. In: *ACM Transactions on Autonomous and Adaptive Systems* 2.4 (2007), S. 15–39.

[RG95] A. S. Rao und M. P. Georgeff. „BDI Agents: From Theory to Practice“. In: *Proceedings of the First International Conference on Multi-Agent-Systems (ICMAS '95)*. San Francisco, USA, 1995.

[RN03] S. Russel und P. Norvig. *Artificial Intelligence – A Modern Approach*. 2. Aufl. Upper Saddle River, New Jersey, USA: Pearson Education International, 2003.

[Rog17] S. Rogers. *Swarm*. 2017. URL: http://www.swarm.org/wiki/Swarm_main_page (besucht am 02. 08. 2017).

[Ron16] G. Rondinella. *Wie moderne Technologien im US-Wahlkampf mitmischen*. 2016. URL: http://www.horizont.net/tech/nachrichten/Chatbots-Roboterjournalisten-und-Co-Wie-moderne-Technologien-im-US-Wahlkampf-mitmischen-143912 (besucht am 07. 11. 2016).

[Sep17] P. Seppecher. *Jamel: Java Agent-based MacroEconomic Laboratory*. 2017. URL: http://p.seppecher.free.fr/jamel/ (besucht am 02. 08. 2017).

[WJ95] M. Wooldridge und N. R. Jennings. „Intelligent Agents: Theory and Practise“. In: *The Knowledge Engineering Review* 10.2 (1995), S. 115–152.

[Wei13] G. Weiss. *Multiagent Systems*. 2. Aufl. Cambridge, Massachusetts, USA: MIT Press, 2013.

[Wei66] J. Weizenbaum. „ELIZA - A Computer Program for the Study of Natural Language Communication between Man and Machine“. In: *Communications of the ACM* 9.1 (1966), S. 36–45.

[Wik17] Wikipedia. *Comparison of Agent-based Modeling Software*. 2017. URL: https://en.wikipedia.org/wiki/Comparison_of_agent-based_modeling_software (besucht am 03. 08. 2017).

[Wil17] U. Wilensky. *NetLogo*. 2017. URL: http://ccl.northwestern.edu/netlogo/ (besucht am 02. 08. 2017).

[Wit92] T. Wittig. *ARCHON: An Architecture for Multi-Agent Systems*. 1. Aufl. Upper Saddle River, New Jersey, USA: Ellis Horwood, 1992.

[Woo09] M. Wooldridge. *An Introduction to MultiAgent Systems*. 2. Aufl. Chichester, West Sussex, UK: John Wiley & Sons, 2009.

[Zan70] C. Zangemeister. „Nutzwertanalyse in der Systemtechnik“. Diss. Technische Universität Berlin, 1970.

Kolophon

Dieses Dokument wurde mit der LaTeX-Vorlage für Abschlussarbeiten an der htw saar der Fakultät für Wirtschaftswissenschaften im Bereich Wirtschaftsinformatik erstellt (Version 1.0). Die Vorlage wurde von Stefan Selle erstellt und basiert weitestgehend auf der entsprechenden Vorlage der Fakultät für Ingenieurswissenschaften, die von Yves Hary, André Miede, Thomas Kretschmer, Helmut G. Folz und Martina Lehser entwickelt wurde.