Universität Hamburg

Fakultät für Mathematik, Informatik
und Naturwissenschaften

Verteilte Systeme und Informationssysteme

Diplomarbeit

Integration von Agentenplattformen in Middleware – am Beispiel von Jadex und Java EE

Hamburg, 25. April 2006

Sven Linstaedt

1linstae@informatik.uni-hamburg.de
Studiengang Informatik
Matr.-Nr.: 5408540
Fachsemester 9

Erstgutachter: Prof. Dr. Winfried Lamersdorf
Zweitgutachter: Dr. Wolf-Gideon Bleek

Danksagung

Ich möchte mich bei allen bedanken, die mir diese Arbeit ermöglicht haben. Insbesondere sind dies Alexander Pokahr und Lars Braubach, die trotz Dissertationsstress immer wieder die Zeit fanden, meine Fragen bezüglich Jadex zu beantworten, mir allzu lässige Formulierungen anzukreiden und rekordverdächtig lange wie auch inhärent unverständliche Sätze mit teilnahmslosen Fragezeichen zu kommentieren. Auch Wolf-Gideon Bleek möchte ich für seine Hilfe und Kritik bei der abschließenden Überarbeitung der Arbeit danken.

Gleichermaßen gilt mein Dank den Entwicklern (und besonders Jeremias Märki) des XSL-FO Prozessors FOP, mit dem diese Arbeit gesetzt wurde. Obwohl noch im Entwicklungsstadium befindlich wird das Open Source Projekt so diszipliniert entwickelt und zügig auf berichtete Fehler reagiert, dass ich immer mit dem aktuellen Trunk arbeiten konnte.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Die vorliegende Diplomarbeit ist am Fachbereich Informatik der Universität Hamburg im Rahmen des Forschungsprojekts *Jadex* entstanden und untersucht die Integrationsmöglichkeiten von Agententechnologie und unternehmensweit akzeptierter Middleware. Basierend auf dieser Untersuchung wird eine Lösung entwickelt, die die Vorteile beider Technologien vereint.

In diesem Kapitel wird eine Einführung in die Problematik gegeben. Dabei wird zur Motivation auf die Ursachen der mangelnden Akzeptanz von Agententechnologie im Unternehmensumfeld eingegangen. Ausgehend von der Problembeschreibung wird danach die Zielsetzung der Arbeit definiert. Abschließend wird die Gliederung der Arbeit näher erklärt.

1.1 Motivation

Der technische Fortschritt in der Entwicklung und Vernetzung von Computersystemen ist Grundlage für einen Trend, den man nach Jean Fourastiés Drei-Sektoren-Hypothese (vgl. [Scha04]) schon in den 70er Jahren beobachten konnte: Durch die Einführung immer verfeinerter, automatisierter Produktionsmethoden und der Ausgliederung (engl. Outsourcing) von Geschäftsbereichen, die nicht zu den Kernkompetenzen der jeweiligen Unternehmen gehörten (z.B. Dienstleistungen wie EDV und Wachdienste), konnte damals immer kostengünstiger produziert werden. Wo auf der einen Seite Arbeitsplätze im Dienstleistungsbereich (tertiären Sektor) entstanden, war die Anzahl der Arbeitsplätze in der Industrie (sekundären Sektor) rückläufig. Die Industriegesellschaft ging in einer Dienstleistungsgesellschaft auf.

Heutzutage ist eine ähnliche Entwicklung im tertiären Sektor zu beobachten: Der Fortschritt ermöglicht eine ganze oder doch zumindest teilweise Automatisierung von Arbeitsfeldern des Dienstleistungssektors. Da es sich bei den automatisierten Dienstleistungen in der Regel um informelle Dienste handelt, bei denen bestimmte Informationen zusammengetragen und aufbereitet werden, kommen häufig komplexe Anwendungen, so genannte *intelligente* Systeme, zum Einsatz. Frühe Vertreter dieser Art von Software sind Expertensysteme, welche monolithisch aufgebaut sind und mit einem bestimmten Wissensbestand sowie Schlussfolgerungsregeln arbeiten, um Probleme zu lösen.

Die Vernetzung von Systemen hat eine neue Art von intelligenten Systemen ermöglicht: Agenten sind deutlich kleinere Softwarekomponenten als beispielsweise Expertensysteme und verfügen deshalb auch nicht über deren Fähigkeiten. Stattdessen handelt es sich bei Agenten um spezialisierte Einheiten, die ihre „Intelligenz“ durch Kommunikation und Interaktion miteinander erhalten. Arbeiten mehrere Agenten zusammen, um ein gemeinsames Ziel zu erreichen, spricht man von einem Agentensystem. Durch die Nutzung von gemeinsamen Verzeichnisdiensten und Kommunikationsprotokollen können solche Systeme dynamisch und verteilt entstehen.

Als Beispiel kann man die Arbeit eines Arbeitsvermittlers anführen: Diese ist in den letzten Jahren aufgrund der angespannten Arbeitsmarktsituation zwar nicht überflüssig geworden, es ist aber zu beobachten, dass immer mehr Arbeitssuchende Stellenangebote über Internetjobbörsen einholen. Viele dieser Jobbörsen bieten mittlerweile über eine aktive Suche hinaus die Möglichkeit, sich nach der Angabe des gewünschten Jobprofils über neue Stellenangebote benachrichtigen zu lassen. Mit dieser Funktionalität avanciert der von den Portalen angebotene Dienst zu einem, den auch die menschlichen Arbeitsvermittler der Bundesagentur für Arbeit wahrnehmen. Da diese im Auftrag des Jobsuchenden arbeiten und seine Interessen vertreten (nämlich ihm einen adäquaten Arbeitsplatz zu suchen), kann man sie ebenso wie die Software der Jobbörsen als Agenten bezeichnen. Tatsächlich wird die entsprechende Funk-

tionalität in einschlägigen Börsen mit dem Terminus *Jobagent* (StepStone) oder *Suchassistent* (Monster) belegt.[1]

Die oben genannten Jobbörsen haben indes eines gemeinsam: Agententechnologie realisiert nur einen kleinen Teil der Anwendung und die Agenten wurden speziell für die Anwendungsdomäne des Systems entwickelt. Die dem System zugrunde liegende Agentenlogik ist nur mit beträchtlichem Aufwand auf andere Domänen übertragbar. Während sich letzteres Problem durch die Nutzung einer der vielen frei erhältlichen Agentenplattformen vermeiden lässt, die eine allgemeine Laufzeitumgebung für Agenten realisieren, würde sich an der geringen Verbreitung von agentenorientiert entwickelter Software insbesondere im Unternehmensumfeld nichts ändern. Die Gründe dafür sind in der fehlenden Unterstützung vieler unternehmenskritischer Anforderungen durch die Plattformen zu suchen. Eigenschaften wie zum Beispiel transaktional geschützte Persistenz, gute Administrierbarkeit und nicht zuletzt die Kompatibilität zu bestehender Software sind maßgebend für den Erfolg von Agententechnologie in Unternehmen.

Es gibt verschiedene Ansätze dieses Problem zu lösen. Die in dieser Arbeit untersuchten Lösungen basieren alle auf der *Java 2 Platform, Enterprise Edition (J2EE)* beziehungsweise *Java Platform, Enterprise Edition (Java EE)*[2] und erweitern diese Middleware um Agentenfunktionalität. Ebenso wie [CaLy03] in ihrem Vergleich von Agentensystemen und der J2EE Plattform, musste der Autor feststellen, dass es bisher kaum Arbeiten zu diesem Thema gibt. Vorhandene Publikationen zum Thema Agententechnologie behandeln die Thematik zumeist aus dem Blickwinkel der Mobilität oder der konzeptuellen Realisierung von Agenten.

1.2 Zielsetzung

Diese Arbeit versucht die Lücke zwischen Unternehmensansprüchen und Agententechnologie zu schließen, indem eine Integration von Agententechnologie in bewährte Middleware untersucht und exemplarisch implementiert wird. Die Idee zu einer solchen Integration ist in einem Universitätsprojekt entstanden, bei dem das ebenfalls an der Universität Hamburg entwickelte Agentenframework *Jadex* zur Realisierung einer Beispielanwendung genutzt werden sollte. Als Middleware-Plattform wurde Java EE gewählt, da diese eine etablierte Größe auf dem Markt darstellt, die von den meisten Unternehmen akzeptiert und unterstützt wird. Mit *Spring*[3] existiert eine weitere Middleware-Plattform, die viele der Probleme von J2EE vermeidet. Die in dieser

1 `http://www.stepstone.de` und `http://www.monster.de`

2 Die aktuell in Entwicklung befindliche Version der Spezifikation ist 5.0 und wird Java EE genannt. Frühere Spezifikationen (bis Version 1.4) sind unter J2EE bekannt. Beide Versionen unterscheiden sich größtenteils nur in der Spezifikation der Enterprise JavaBeans. Daher werden im weiteren Verlauf der Arbeit beide Akronyme soweit nicht explizit darauf hingewiesen gleichbedeutend verwendet.

3 `http://www.springframework.org`

Arbeit vorgestellten Integrationsansätze basieren allerdings alle auf J2EE, da sich existierende Publikationen wie etwa [CaLy03], [TaEO05] und [AgIn05] ebenfalls auf diese Plattform beziehen.

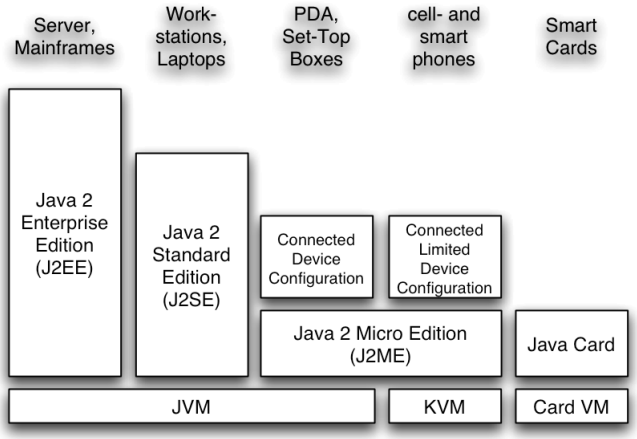

Abb. 1: Übersicht über die Java Editionen nach [Sun00]

Zur Einordnung von J2EE wird in Abbildung 1 ein Überblick über die Java Editionen gegeben (vgl. [Sun00]). Die meisten Agentenplattformen werden auf Basis der Java 2 Platform, Standard Edition (J2SE) implementiert. Für die auf mobile Geräte zugeschnittene Java 2 Platform, Micro Edition (J2ME) existieren mit einer von [Harb04] durchgeführten Portierung von Jadex ebenfalls Agentenplattformen. Für J2EE gibt es bisher – von kommerziellen Produkten abgesehen – nur wenige, für Unternehmen zumeist unzureichende Ansätze, die Agenten auf dieser Java Plattform realisieren.

Ziel dieser Arbeit ist es demzufolge, das Agentenframework Jadex in die J2EE Plattform zu integrieren, um auf dessen Grundlage eine zuverlässige und den Ansprüchen von Unternehmen genügende Laufzeitumgebung für Agenten zu haben. Dabei sollen möglichst viele der von der J2EE Plattform angebotenen Dienste wie beispielsweise Verwaltung, Überwachung und transaktionaler Zugriff in Jadex genutzt werden. Weiterhin soll der wechselseitige Zugriff von Jadex-Agenten und bestehender Software soweit wie möglich vereinfacht werden, um die Interoperabilität zu gewährleisten. Mit Blick auf die Spezialisierung der Entwickler wäre es ideal, wenn weder Agenten- noch J2EE-Entwickler neue Schnittstellen zu lernen hätten und die wechselseitige Nutzung von Diensten vollkommen transparent wäre.

Um dieses Ziel zu erreichen wird eine Bestandsaufnahme existierender Softwareframeworks gemacht, die eine vollständige oder partielle Integration von Agententechnologie in Middleware umsetzen. Das Hauptaugenmerk liegt hierbei sowohl auf der Analyse bestehender Lösungen für Agententechnologie im Unternehmensumfeld im Allgemeinen, als auch auf einer Untersuchung der Integrationsmöglichkeiten von Jadex in bestehende Unternehmenssoftware im Speziellen.

Ausgehend von dieser Analyse wird einer der Integrationsansätze beispielhaft für Jadex realisiert und anhand einer Beispielanwendung validiert.

1.3 Gliederung

An die Einleitung anschließend wird im zweiten und dritten Kapitel ein Überblick über die in dieser Diplomarbeit verwendeten Technologien gegeben. Neben einer allgemeinen Einführung in die Thematiken Agententechnologie und Middleware, werden insbesondere das zugrunde liegende Agentenframework Jadex sowie die für diese Arbeit gewählte Middleware Java EE beschrieben. Dabei stehen die technischen Aspekte im Mittelpunkt, da diese für den Erfolg einer möglichen Integration maßgebend sind.

Im darauf folgenden Kapitel werden die existierenden Integrationsansätze untersucht. Dazu werden zuerst die Anforderungen der Jadex-Agenten an ihre Laufzeitumgebung festgehalten. In der anschließenden Analyse der Integrationsansätze werden diese Anforderungen wiederholt aufgegriffen, um festzustellen wie sie in den jeweiligen Ansätzen realisiert wurden. Die Ergebnisse der Untersuchung werden abschließend zusammengefasst und auf ihre Vor- und Nachteile hin bewertet.

Aufgrund dieser Analyse wird im fünften Kapitel einer der aufgezeigten Ansätze begründet ausgewählt und implementiert. Die Beschreibung der Implementierung wird hier ebenso aufgeführt, wie die bei der Implementierung aufgetretenen Probleme und ihre Lösungen. Das Kapitel schließt mit der Beschreibung eines Beispielszenarios und seiner Umsetzung. Diese Beispielanwendung soll die Vorteile der Integration aufzeigen.

Abschließend werden in einem Resümee die erzielten Ergebnisse mit der vorher definierten Zielsetzung verglichen und die im Zuge der Diplomarbeit aufgetauchten, weiterführenden Ideen kurz umrissen.

Kapitel 2

Agententechnologie

An die Einleitung anschließend wird in diesem und dem nächsten Kapitel ein Überblick über die in dieser Diplomarbeit verwendeten Technologien gegeben. Dazu wird zuerst auf die Agententechnologie eingegangen.

Nach einem Überblick über die grundlegenden Eigenschaften von Agenten wird das BDI-Modell erklärt. Im Anschluss daran wird ein Überblick über Jadex, das in dieser Arbeit verwendete Framework für BDI-Agenten, gegeben. Dabei werden die Konzepte hinter Jadex ebenso erklärt wie deren technische Umsetzung. Letztere ist mit Blick auf die Restriktionen von Applikationsservern relevant und wird entsprechend ausführlich behandelt.

2.1 Hintergrund

Der im Kontext der Informatik benutzte Begriff des Agenten wird als Kurzform für Softwareagent verwendet und bezeichnet ein Computerprogramm, das sich durch bestimmte Eigenschaften auszeichnet (siehe Kapitel 2.2). Ihren Namen haben Softwareagenten in Anlehnung an menschliche Agenten erhalten, da sie, ähnlich ihrem menschlichen Vorbild, im Auftrag eines anderen für dessen Interessen handeln. Das Spektrum des Einsatzes menschlicher Agenten reicht dabei von politischen (klassisches Beispiel: als Geheimagent im Auftrag eines Staates), über wirtschaftlichen (z.B. als Handelsvertreter) bis hin zu privaten Interessen. Nach [MuJo00] haben alle Agenten unabhängig von ihrem Einsatzort aber eines gemeinsam: In der Regel sind sie speziell für ihre Aufgaben ausgebildet worden und haben Zugang zu Ressourcen, um die ihnen aufgetragenen Ziele zu erreichen. Diese Spezialisierung ermöglicht es ihnen, die von ihnen angebotenen Leistungen effizienter zu erfüllen als es der Auftraggeber selbst tun könnte.

2.2 Eigenschaften von Agenten

Ausgehend von der bisherigen Definition eines Agenten[4] könnten viele Computerprogramme als Agenten angesehen werden. Tatsächlich existiert keine allgemein anerkannte Definition für Softwareagenten, so dass viele Softwarehersteller auf die oben genannte für die Informatik unzureichende Definition von (menschlichen) Agenten zurückgreifen und von ihrer Software kurzerhand als Agenten reden. Stan Franklin und Art Graesser fassen in ihrer Arbeit über Klassifikationsmöglichkeiten von Agenten die unterschiedlichen Definitionsansätze zusammen und versuchen die Gemeinsamkeiten aller Ansätze herauszuarbeiten. Ihrer Meinung nach definiert sich ein Agent wie folgt:

> An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future. [FrGr96]

Damit gleicht ihre Definition größtenteils der von Michael Wooldridge und Nicholas Jennings (vgl. [WoJe95]). Beide Definitionen haben die folgenden Eigenschaften gemeinsam:

- **Autonomie** bedeutet für den Agenten, dass alle Handlungen unter seiner Kontrolle liegen und er keinerlei Steuerung von außen erfährt.
- Um mit der **Umgebung interagieren** zu können, muss ein Agent sie wahrnehmen und auf ihren Zustand zumindest ein gewissen Einfluss haben können.

4 Im weiteren Verlauf der Arbeit werden die Begriffe *Agent* und *Softwareagent* der Kürze wegen synonym verwendet.

Nach dieser Definition würden laut [FrGr96] primitive Lebensformen wie Bakterien ebenfalls zu den Agenten zählen. Um die Menge der Agenten noch weiter einzuschränken, haben Wooldridge und Jennings zusätzliche Eigenschaften für Agenten festgelegt, die sich nur noch zum Teil mit denen von Franklin und Graesser decken:

- **Soziabilität:** Agenten sollen mit anderen Agenten kommunizieren können, z.B. um von ihnen Dienste oder Informationen anzufordern, zu denen sie selbst nicht fähig sind beziehungsweise über die sie selbst nicht verfügen. Typischerweise kommt für diese Kommunikation eine spezielle Sprache zum Einsatz, die meist aus Sprechakten bestehen. Eine *Ontologie* bildet dabei eine gemeinsame Grundlage für die Bedeutungsinhalte der Sprache. Nach [Grub93] ist eine Ontologie als „explizite Spezifikation einer Konzeptualisierung“ definiert. Zusammengefasst werden die Kommunikationssprachen unter dem Begriff Agent Communication Language (ACL).
- **Reaktivität:** Diese Eigenschaft bezeichnet die Fähigkeit von Agenten, zeitnah auf auftretende Ereignisse (z.B. Nachrichten anderer Agenten oder Änderungen in der Umwelt) reagieren zu können.
- **Proaktivität:** Neben einem reaktiven Verhalten, sollten Agenten in der Lage sein, von sich aus aktiv zu werden und zielgerichtet zu handeln.

Mit ihren inhärenten Eigenschaften, der asynchronen Kommunikation voneinander unabhängiger Entitäten, stellt die agentenorientierte Softwareentwicklung ein neues Programmierparadigma dar, das den Entwicklungsprozess komplexer Systeme vereinfachen und diese wartbarer machen kann. Die Entwicklung klassischer, nebenläufiger Anwendungen ist dagegen meist fehleranfälliger, da sie von den gängigen Programmiersprachen konzeptionell nicht unterstützt wird.

Über diese Eigenschaften hinaus gibt es noch weitere, die nur für bestimmte Agententypen zutreffen. Man bezeichnet Agenten beispielsweise als *mobil*, wenn sie in der Lage sind, die Plattform zu wechseln. Dies ist notwendig, wenn die Erreichbarkeit der Ursprungsplattform nicht gewährleistet werden kann (bei mobilen Systemen wie etwa Handheld-Computern) und der Agent auf einer anderen Plattform durchgängig mit anderen Agenten kommunizieren könnte.

2.3 BDI-Modell

Die bisherige Definition eines Agenten beschreibt nur seine von außen sichtbaren Verhaltensmuster. Er kommuniziert bei Bedarf mit anderen Agenten und nimmt seine Umwelt war und wirkt aufgrund dieser Wahrnehmung wiederum auf sie ein, um seine Ziele zu erreichen. Dieses Verhalten entspricht ansatzweise dem aus der Psychologie bekannten Prinzip der *Handlungsregulationstheorie* [Kühn01]. Nach [Hack98] und [Volp94] ist das Modell der Handlungsregulationstheorie (siehe Abbildung 2) zyklisch und hierarchisch aufge-

baut und beinhaltet die Phasen Planung, Ausführung und Kontrolle. In der Planungsphase werden die Ziele untersucht und die Handlungen ermittelt, die für das Erreichen der Ziele vorteilhaft sind. Anschließend werden die ermittelten Handlungen (*Transformationen* genannt) ausgeführt. In der letzten Phase werden die Auswirkungen der Handlung validiert. Diese Überprüfung der Ergebnisse sorgt im nächsten Zyklus dafür, dass möglicherweise andere Handlungen erwogen werden, wenn man zum Beispiel den Zielen nicht näher gekommen ist. Bei komplexen Zielen kann in der Planungsphase entschieden werden, das Ziel in mehrere Teilziele zu gliedern, um so die Komplexität zu verringern und die Auswahl möglicher Handlungen zu erleichtern.

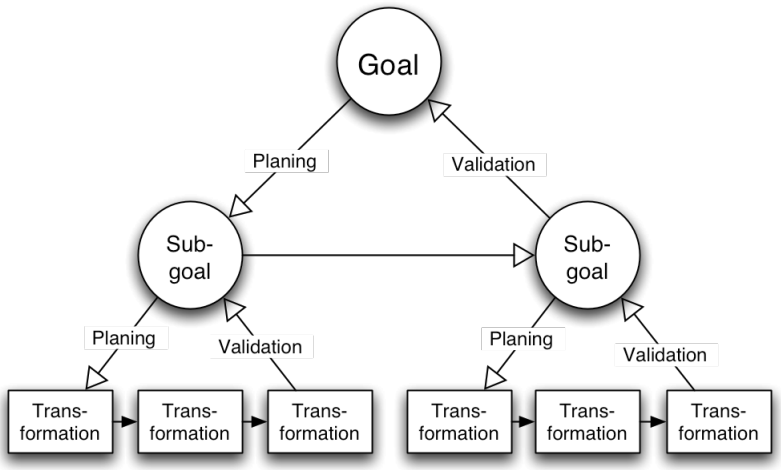

Abb. 2: Modell der Handlungsregulationstheorie

Das Modell der Handlungsregulationstheorie könnte als Grundlage zur Modellierung der internen Verarbeitung von Agenten dienen. Tatsächlich ist die Ähnlichkeit mit dem in dieser Arbeit verwendeten Belief-Desire-Intention (BDI) - Modell bemerkenswert. Dieses wohl am meisten verwendete so genannte *practical reasoning* Modell wurde ursprünglich 1987 von Michael Bratman entwickelt, der sich in seiner Funktion als Professor der Philosophie an der Stanford Universität mit der Entscheidungsfindung bei Menschen beschäftigt hat [WikiBDI06]. *Practical reasoning* (dt. praktisches Schlussfolgern) ist die Fähigkeit der Entwicklung von Lösungswegen für ein gesetztes Ziel. In der ersten Phase der *deliberation* (dt. Überlegung) wird bestimmt, was als nächstes gemacht werden soll, während in der darauf folgenden Phase des *means-ends reasoning* (dt. etwa für "Mittel zum Zweck-Folgern") wie das Ziel unter Ausnutzung der vorhandenen Ressourcen erreicht werden kann (nach [Wool02]). Seine Verbreitung fand es vermutlich deswegen, weil es an die intuitiv leicht erfassbare, menschliche Handlungsregulationstheorie angelehnt ist.

Das BDI-Modell besteht aus den folgenden drei Kernkomponenten (vgl. [GPPTW99] und [Wool96]):

- **Beliefs** (dt. Überzeugungen, Glaube) stellen das Wissen des Agenten über seine Umwelt und seinen eigenen Zustand dar und sind die Grundlage, auf der er Entscheidungen trifft. Abhängig von der Implementierung kommen dafür Variablen, Relationen in Datenbanken oder symbolische Ausdrücke in einer prädikatenlogischen Sprache zum Einsatz. Dieses Wissen über seine Umwelt ist üblicherweise weder vollständig noch korrekt. Der Agent hat zu entscheiden, welche der wahrgenommen Informationen aus seiner Umwelt für seine Entscheidungen relevant sind. Darüber hinaus kann sein Wissen über die Umwelt veraltet und damit nicht mehr korrekt sein.
- **Desires** (dt. Begehren, Wünsche) sind ein weiterer essentieller Teil des Agentenzustands und repräsentieren seine Ziele. Ein solches Ziel kann das Erreichen oder Wahren eines bestimmten Zustands sein. Abhängig vom Ziel führt der Agent bestimmte Aktionen aus, um seinem Ziel näher zu kommen. Er handelt demnach proaktiv (zielorientiert). Im Gegensatz dazu steht die klassische, aufgabenorientierte Softwareentwicklung. In dieser werden Aktionen von einer Software ausgeführt, ohne zu wissen, warum sie ausgeführt werden. Der Vorteil zielorientierten Handelns gegenüber dem aufgabenorientierten liegt darin, dass bei ersterem zum Beispiel Wissen um fehlgeschlagene Aktionen für die weitere Verfolgung der Ziele ausgenutzt werden kann, indem etwa Handlungsalternativen gesucht werden.
- **Intentions** (dt. Absichten, Vorsätze) sind die Handlungen, zu deren Verfolgung sich der Agent entschlossen hat. In der Regel wird ein Agent dabei nicht alle seine Ziele gleichzeitig verfolgen können, selbst wenn die einzelnen Ziele nicht im Widerspruch zueinander stehen. Deswegen ist die Menge der Intentions meist eine Untermenge der Desires. Typischerweise wird ein Agent dabei solange versuchen seine Intentions zu verfolgen wie sie dem Agenten als erreichbar scheinen und sie noch nicht erreicht sind.

Abbildung 3 zeigt den Zusammenhang zwischen den drei Komponenten. Die Beliefs werden normalerweise über Nachrichten anderer Agenten oder Sensoren, die Änderungen der Umwelt überwachen, aktualisiert. Dem gegenüber stehen die Intentions, also die aktiv verfolgten Ziele, aufgrund deren ein Agent mit seiner Umwelt interagiert. Neben diesen Komponenten existieren ein Interpreter als zentrale Einheit und eine Planbibliothek. Letztere beinhaltet alle dem Agenten bekannten Handlungsfolgen, mit denen er auf seine Umwelt Einfluss nimmt, um seine Ziele zu erreichen.

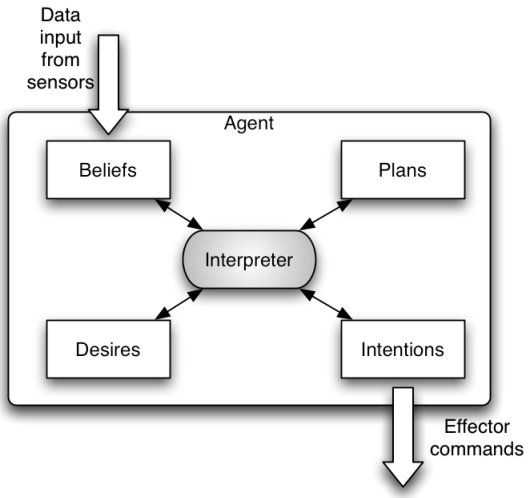

Abb. 3: Zusammenhang der BDI-Architektur

Der Interpreter operiert auf den vier genannten Komponenten. Zum Beispiel generiert er aufgrund aktualisierter Beliefs neue Desires und entscheidet, welche Desires aktiv als Intentions verfolgt werden. Um diese Intentions zu erfüllen, wählt er Pläne aus der Planbibliothek aus, die ihm für das Erreichen des Ziels geeignet erscheinen beziehungsweise bricht eben diese Pläne wieder ab, wenn das Ziel erreicht wurde.

2.4 BDI-Plattformen

Wie Abbildung 4 zeigt, sind zwar mehrere frei verfügbare und kommerzielle Agentenplattformen erhältlich, die meisten fokussieren aber entweder auf dem BDI-Konzept (Agenten als schlussfolgernde Entitäten) oder auf dem Middleware-Aspekt (Interoperabilität zwischen heterogenen Plattformen). Es gibt aber einige Ansätze, die die Kluft zwischen beiden Ansätzen zu schließen versuchen: *Agentis* von der gleichnamigen Firma und Whitesteins *Living Systems Technology Suite* (LS/TS) sind zwei kommerzielle Produkte, die auf der Basis von Java EE Applikationsservern (siehe Kapitel 3.5) ein Agentensystem realisieren. Beide Ansätze werden in Kapitel 4.4.2 (LS/TS) und Kapitel 4.4.3 (Agentis) näher untersucht. Ein anderer Ansatz besteht darin, bereits vorhandene FIPA-konforme Agentenplattformen um das BDI-Konzept zu erweitern. Beispiele dafür sind Jadex, Nuin[5] und FIPA-JACK[6] (vgl. [PBL05a]). Die beiden letzteren sind für diese Arbeit nicht relevant und werden deshalb hier nicht weiter erklärt.

5 http://www.nuin.org/
6 http://www.agent-software.com.au/ und http://www.cs.rmit.edu.au/agents/protocols/

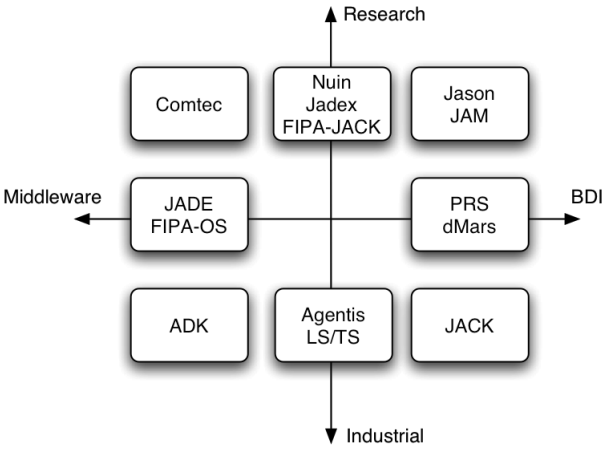

Abb. 4: Klassifikation von Agentenplattformen nach [PBL05a]

Die FIPA (Foundation for Intelligent Physical Agents) ist ein in der Schweiz registrierter, gemeinnütziger Verband, der es sich zum Ziel gesetzt hat, für Interoperabilität zwischen den heterogenen Agentenplattformen zu sorgen. Dazu werden in internationaler Zusammenarbeit Spezifikationen publiziert, die z.B. den Nachrichtenaustausch zwischen Agenten oder die von den Agentenplattformen angebotenen Dienste betreffen. Diese Dienste beinhalten das Starten, Stoppen und Auffinden von Agenten (White Page Service), die vom so genannten Agent Management System (AMS) realisiert werden sowie der Directory Facilitator (DF), welcher die von den Agenten angebotenen Dienste verwaltet und durchsuchbar zur Verfügung stellt (Yellow Page Service).

2.5 Jadex

Jadex (JADE Extension) ist eine an der Universität Hamburg entwickelte BDI Reasoning-Engine, die in Java geschrieben wurde. Ursprünglich als Erweiterung des Java Agent Development Framework (JADE) konzipiert (Kapitel 2.5.4 behandelt die aktuelle Architektur), sollte Jadex die Defizite von JADE beseitigen. Reasoning-Engines und Agentenplattformen setzen zwei unterschiedliche Konzepte um: Während Reasoning-Engines die interne Verarbeitung von Agenten umsetzen, realisieren Agentenplattformen die zum Betrieb der Agenten notwendigen Dienste.

JADE wiederum ist ein von der Telecom Italia Lab (TILAB) komplett in Java entwickelte Agentenplattform, die die Entwicklung von Multi-Agenten Systemen vereinfachen soll. Dazu setzt JADE die Empfehlungen der FIPA um und stellt dem Entwickler grafische Werkzeuge zur Verfügung, mit denen er seine Agenten zur Laufzeit debuggen (dt. von Fehler befreien, testen) kann. Das Framework stellt alle zur Laufzeit der Agenten nötigen Dienste bereit, die nicht Teil der Agenten sind, wie zum Beispiel Transport und Kodierung von Nachrichten sowie AMS und DF. Ausgehend von diesen Diensten könnte man JADE als Middleware für Agenten betrachten [BCPR03].

Da sich die Entwickler von JADE auf die Konformität mit den FIPA-Spezifikationen und die Entwicklung von Werkzeugen zum Testen der Agenten konzentriert haben, blieb die Umsetzung eines internen Agentenkonzeptes bewusst unberücksichtigt [PBL03]. Jadex ist die Lösung dieses Defizits, indem es auf Basis der relativ einfach gestrickten JADE-Agenten ein Framework aufsetzte, das die Entwicklung von BDI-Agenten ermöglichte.

2.5.1 Agent Definition File

Jadex Agenten bestehen aus einem in einer XML-Datei Datei – Agent Definition File (ADF) genannt – gespeicherten Modell, welches die BDI-typischen Komponenten beinhaltet sowie den in Java geschriebenen Plänen. Wie aus Abbildung 5 deutlich wird, beinhaltet dieses Modell neben dem Agententypnamen und dem Java Package, in dem es liegt, die folgenden Attribute (vgl. [PBL05b]):

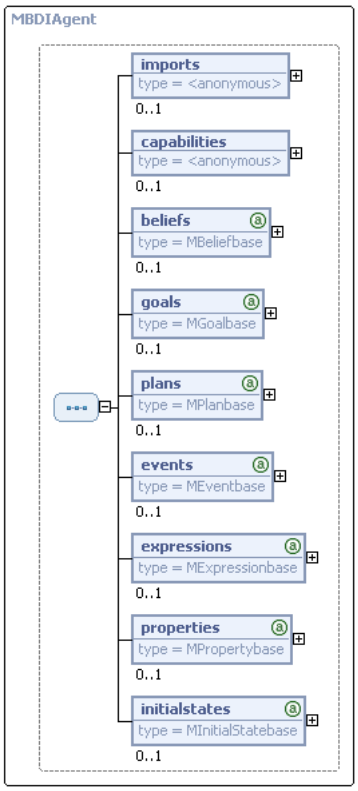

Abb. 5: Aufbau des ADF für Jadex Agenten

Imports

In den Imports können die in diesem ADF verwendeten Java-Klassen angegeben werden, so dass man diese nur mit ihren Klassennamen (d.h. ohne Angabe des Package) ansprechen kann. Dieser Mechanismus stimmt mit dem in Java-Klassen verwendeten Import-Mechanismus überein.

Capabilities

Capabilities (dt. Fähigkeiten) bieten dem Entwickler die Möglichkeit, oft benutzte Funktionalität eines Agenten in einer Capability genannten XML-Datei auszulagern und diese dann an in diesem Attribut zu referenzieren. Das soll die Wiederverwendbarkeit von Agentenfunktionalität unterstützen und die Komplexität der ADFs verringern.

Beliefs

Beliefs (dt. Glauben, Wissen) bieten dem Entwickler die Möglichkeit Wissen abzuspeichern, auf das agentenweit (also auch von allen Plänen) zugegriffen werden kann. Dazu kann man einfache Beliefs und Beliefsets deklarieren und mit einem Javatyp versehen.

- Ein Belief speichert immer genau einen Wert des angegebenen Typs ab (oder `null`), wobei die Typprüfung zur Laufzeit geschieht.
- Beliefsets können mehrere Objekte eines Typs speichern. Entsprechend der Mengenlehre kann ein Objekt dabei immer nur einmal in einem Beliefset vorkommen, wobei die Gleichheit zweier Objekte mittels `equals()` geprüft wird und ein bestehendes Objekt bei Gleichheit von dem Neuen aus dem Beliefset verdrängt wird.

Goals

Die Goals (dt. Ziele) deklarieren die dem Agenten bekannten möglichen Ziele samt ihrer Parameter. Jadex kennt vier Zieltypen:

- Perform Goals (dt. Ausführung, Durchführung) beziehen sich direkt auf eine auszuführende Handlung und sind nach erfolgreicher Ausführung erfüllt.
- Achieve Goals (dt. Erreichen, Erlangen) sind Ziele, die mit dem Erreichen eines bestimmten Umweltzustands erfüllt sind. Gleichzeitig kann eine Abbruchbedingung angegeben werden, bei der das entsprechende Ziel fehlschlägt und verworfen wird.
- Maintain Goals (dt. beibehalten, aufrechterhalten) versuchen dagegen, einen bestimmten Zustand aufrecht zu erhalten. Normalerweise ist diese Art von Zielen nie erfüllt. Das entsprechende Ziel wird bei Erreichen des geforderten Zustands nicht entfernt, sondern nur deaktiviert. Sollte der Zustand irgendwann einmal nicht mehr die geforderten Bedingungen erfüllen, wird das Ziel mit der Absicht wieder aktiviert, die Diskrepanz zwischen Ist- und Soll-Zustand zu beseitigen.
- Query Goals sind Ziele, die ähnlich den Achieve Goals auf einen bestimmten Zustand abzielen. Im Gegensatz zu den Achieve Goals versuchen sie aber einen bestimmten inneren Agentenzustand herbeizuführen, indem sie etwa Informationen von anderen Agenten erfragen.

Plans

Unter Plans (dt. Plan, Entwurf) werden die dem Agenten bekannten Pläne deklariert, sowie deren jeweilige Events (dt. Ereignis) referenziert, bei de-

nen der Plan aktiv werden soll. Tritt das Ereignis ein, wird eine neue Objektinstanz des Plans erzeugt und für die Ausführung in der Agenda (siehe Kapitel 2.5.3) vermerkt. Diese Komponente stellt also die Planbibliothek des Agenten dar. Die Pläne sind Java-Klassen, die abhängig vom Plantyp von bestimmten Klassen erben (`jadex.runtime.Plan` beziehungsweise `jadex.runtime.MobilePlan`) und mittels einer implementierten Methode (`Plan.body()` beziehungsweise `MobilePlan.action(IEvent event)`) in prozeduraler Weise beschreiben, was der Agent tun soll. Die momentane Architektur sieht zwei Arten von Plänen vor:

Der **Standardplan** bietet dem Entwickler die Möglichkeit, den Plan durch die `waitForXXX()`-Methoden von `jadex.runtime.Plan` an einer bestimmten Stelle auf ein Ereignis warten zu lassen (z.B. die Antwortnachricht auf eine Anfrage bei einem Agenten oder die Beendigung eines Teilziels). Von der technischen Seite betrachtet wird jedem Plan statisch ein Thread zugewiesen. Dieser Thread wird zunächst mittels `Object.wait()` angehalten und erst beim Eintreten des angegebenen Ereignis wieder aufgeweckt, um den Plan weiter abzuarbeiten.

Diese Lösung mag programmiertechnisch zwar leicht verständlich sein, da sich mit ihr die aus der objektorientierten Programmierung (OOP) bekannten, synchronen Aufrufmuster von Methoden realisieren lassen (vgl. Kapitel 3.4.3), sie hat aber den Nachteil, dass der Agentenzustand nicht serialisierbar ist solange zumindest ein Plan ausgeführt wird. Der Grund dafür ist in der Sprache Java selbst zu suchen. Aufgrund der Abstraktion von der Rechnerarchitektur kann man mit Java im Gegensatz Sprachen wie C nicht direkt die physikalischen Speicherbereiche des Stacks auslesen. Auch bietet Java kein Application Programming Interface (API) an, um auf den Stack eines Threads zuzugreifen (vgl. [GJSB05]). Dieser Zugriff wäre notwendig, um die lokalen Objekte und primitiven Variablen eines Methodenaufrufs zu sichern beziehungsweise wiederherzustellen. Nach [BHKPB03] ist dieses zwar technisch möglich, in der Praxis würde das aber entweder eine Manipulation der Java Virtual Machine (JVM) erfordern, oder aber die Modifikation des Quelltexts beziehungsweise der kompilierten Klassen via Bytecode-Enhancement. Beide Möglichkeiten haben ihre Nachteile, die sie für eine Integration von Agenten in Applikationsservern ungeeignet erscheinen lassen und deswegen nicht weiter betrachtet werden.

Einen alternativen Ansatz bietet der so genannte **mobile Plan**. Dieser besitzt dieselben `waitForXXX()`-Methoden wie der Standardplan, allerdings haben diese im mobilen Plan eine andere Semantik. Im Gegensatz zum Standardplan bekommt man beim Aufruf der mobilen `waitForXXX()`-Methoden ohne Unterbrechung des Plans ein `jadex.runtime.IFilter` als Rückgabewert. Die Planausführung wird demnach direkt nach dem `waitForXXX()`-Aufruf fortgesetzt. Die entsprechende Planinstanz wird dann allerdings nach Beenden von `MobilePlan.action(IEvent event)` nicht als erfolgreich abgeschlossen betrachtet (und damit für die Garbage-

Collection freigegeben) wie es beim Standardplan der Fall ist. Tritt das entsprechende Ereignis ein, wird dieselbe Planinstanz, die vorher `waitForXXX()` aufrief, wieder über `MobilePlan.action(IEvent event)` aufgerufen. Der Agent kann mit Hilfe des vorher zurückgegebenen Filters prüfen, ob der wiederholte Aufruf der Planinstanz auf das Eintreten des Ereignisses zurückzuführen ist und auf eine entsprechende Behandlung desselben verzweigen. Ein mobiler Plan gilt erst dann als erfolgreich abgeschlossen, wenn `MobilePlan.action(IEvent event)` abschließt, ohne dass eine `waitForXXX()`-Methode aufgerufen wurde. Mögliche Ereignisse werden im folgenden Abschnitt beschrieben. Die entsprechenden Events erben alle von `jadex.runtime.IEvent`.

Events

Mit den Events hat der Entwickler die Möglichkeit, dem Agenten bekannte Ereignisse inklusive aller das Ereignis betreffenden Parameter zu deklarieren, auf die der Agent mit entsprechend zugewiesenen Plänen reagieren kann. Solche Ereignisse sind Nachrichten anderer Agenten und interne Ereignisse:

- Eingehende Nachrichten anderer Agenten resultieren in einem `jadex.runtime.IMessageEvent`. An dieser Stelle werden ebenfalls ausgehende Nachrichten aufgeführt. Diese erzeugen normalerweise kein Ereignis, auf das der Agent reagiert. Vielmehr kann der Entwickler eines Plans ein `IMessageEvent` erzeugen, es mit passenden Parametern (z.B. den Empfänger) ausstatten (Typprüfung der Parameter zur Laufzeit) und abschicken. Die Plattform stellt sicher, dass abgeschickte `IMessageEvents` über einen vom Empfänger der Nachricht abhängigen Transportkanal an die Plattform des Empfängers übermittelt werden, welche die Nachricht wiederum dem Empfänger übergeben würde.
- Interne Ereignisse bieten dem Entwickler die Möglichkeit, einen Mechanismus zur agenteninternen Ereignisverarbeitung zu verwenden. Da keine Informationen über die Verarbeitung des Ereignisses erhalten werden können, stellen interne Ereignisse eine leichtgewichtige Alternative zu Zielen dar.

Expressions

Expressions (dt. Ausdrücke) beinhalten sowohl Ausdrücke, die in einem oder mehreren Ergebnissen resultieren, als auch Bedingungen, die entweder wahr oder falsch sind. Beide werden in einer deklarativen Expression Language (dt. Ausdruckssprache) angegeben, die extra für Jadex entworfen wurde und an die Object Query Language (OQL) angelehnt ist. Mit Hilfe dieser Sprache können zum Beispiel Bedingungen für die Ausführung eines Ziels angegeben werden, die Wissen des Agenten abhängig sind. Innerhalb von Plänen können solche Ausdrücke und Bedingungen ebenso verwendet werden. Da diese Ausdrücke ebenso oft groß wie komplex werden können, kann man sie zur Verbesserung der Übersicht nicht nur in den Java Quelltexten der Pläne angeben, sondern auch im ADF.

Properties

Properties (dt. Eigenschaften) können statische Einstellungen des Agenten aufnehmen. Beispiele hierfür sind unter anderem das Logging Level (dt. Grad der Ausgabe von Debuginformationen).

InitialStates

Abschließend bieten die InitialStates (dt. Anfangszustand) die Möglichkeit, denselben Agententyp unterschiedlich parametrisiert zu starten. Als Anfangszustände sind zum Beispiel anfänglich aktive Ziele oder Pläne naheliegend. Sie können ebenfalls dazu genutzt werden, einen Agenten mit unterschiedlichem Vorwissen zu erzeugen.

2.5.2 Agentenmodell

Wie schon in den vorangegangenen Abschnitt angedeutet, handelt es sich bei dem Modell nur um die *Struktur* eines Agenten und nicht um eine Agenteninstanz. Es deklariert nur die Ziele, Pläne und Ereignisse, die dem Agenten bekannt sind beziehungsweise das Wissen, das er haben könnte. Man kann dieses Modell mit einer aus Java bekannten Klasse vergleichen. Diese enthält in der Regel die Typen der Werte, die den Zustand einer Instanz dieser Klasse ausmachen würden.[7]

Instanziiert man ein Objekt dieser Klasse, wird über einen Konstruktor der initiale Zustand des Objekts festgelegt. Gleiches gilt für einen neu erstellten Agenten, nur dass sein Anfangszustand mit Hilfe der InitialStates ausgewählt werden kann. Zur Laufzeit verhalten sich Objekt und Agent ähnlich. Beide können ihren Zustand ändern. Während dieses beim Objekt über den Aufruf der sichtbaren Methoden geschieht, kann der Agent dagegen auch proaktiv handeln (d.h. ohne Aufruf von außen) und mit diesen Handlungen seinen Zustand ändern.

Auch wenn Agentenmodelle und Klassen zur Laufzeit normalerweise statische Konstrukte darstellen, bieten doch beide die Möglichkeit, ihre Struktur zur Laufzeit zu ändern.[8] Jadex hat für die Modifikation des Modells eine eigene API. Da für jeden neuen Agenten das ADF geladen, auf Fehler überprüft und schließlich in eine dem Agenten zugeordnete Objektmodell überführt wird, sind die Änderungen am Modell nur für den Agenten sichtbar, der diese Änderungen vorgenommen hat. Theoretisch ließen sich aber mit diesem Objektmodell neue Agenten erstellen.

---

7 Statische Variablen ermöglichen zwar Klassen mit einem anwendungsweiten Zustand, seien hier aber außen vorgelassen.

8 Java bietet über die *Reflection API* im Gegensatz zu Sprachen wie z.B. C++ nur lesenden Zugriff die Struktur einer Klasse.

2.5.3 Agenda

Neben dem Modell und den Zustand, den der Agent im Rahmen des Modells einnehmen kann, gibt es noch eine dritte für den Agenten wichtige Komponente: seine Agenda. Diese nimmt nach [PBL05c] alle geplanten, auf ihre Ausführung wartenden Aktionen auf. Abhängig von seiner Selektionsstrategie wählt ein Interpreter Aktionen aus der Agenda aus, die als nächstes ausgeführt werden sollen. Ebenso entfernt er Aktionen aus der Agenda, wenn diese etwa durch ein erreichtes Ziel obsolet geworden sind. Als Aktionen kommen sowohl die Ausführung von Plänen in Frage, als auch die Verarbeitung eingehenden Nachrichten oder Benachrichtigungen der Plattform. Das Fallenlassen von Zielen oder die Suche nach der am besten passenden Bearbeitung eines Ereignisses zählen ebenso zu diesen Aktionen.

2.5.4 Integration über Adapter

Das letzte Release von Jadex (Version 0.94) beinhaltet eine Umstrukturierung, die aus der ursprünglichen JADE-Erweiterung eine unabhängige BDI Reasoning-Engine macht, welche dank geeigneter Schnittstellen über so genannte Adapter auf beliebiger Middleware eingesetzt werden kann (vgl. [PBL05a]). So gibt es mittlerweile neben dem aus dem Refactoring hervorgegangenen JADE-Adapter noch einen Standalone-Adapter, der eine eigenständige Implementierung einer Plattform für Jadex-Agenten darstellt. Im Gegensatz zum JADE-Adapter setzt die Standalone-Plattform die Einhaltung einer FIPA-konformen Kommunikation nicht voraus, sondern bietet verschiedene Möglichkeiten der Kodierung und des Transports von Nachrichten. So lassen sich etwa prinzipiell beliebige serialisierbare Objekte verschicken, während man bei JADE als Plattform für Jadex-Agenten bei Kodierung und Transport auf die Möglichkeiten von JADE eingeschränkt ist.

Als Transportmechanismen bietet die Standalone-Plattform im momentanen Entwicklungsstand proprietäre Implementierungen sowohl eines einfachen auf TCP/IP (Transmission Control Protocol / Internet Protocol) basierten Transportkanals, als auch die Anbindung einer JMS-konformen Message-oriented Middleware (siehe Kapitel 3.5.5). Als Codecs (dt. Kodierung, Verschlüsselung) gibt es dabei neben dem klassischen Serialisierungsmechanismus von Java-Objekten ein Stringcodec und eine XML-basierte Kodierung von Nachrichten.

Abbildung 6 stellt den Zusammenhang zwischen Jadex-Agenten, Adapter und Plattform unter Verwendung der relevanten Interfaces (dt. Schnittstellen) dar. Agenten und Plattform machen der jeweils anderen Seite ihre Dienste über eine Schnittstelle (`jadex.adapter.IJadexAgent` und `jadex.adapter.IAgentAdapter`) zugänglich.

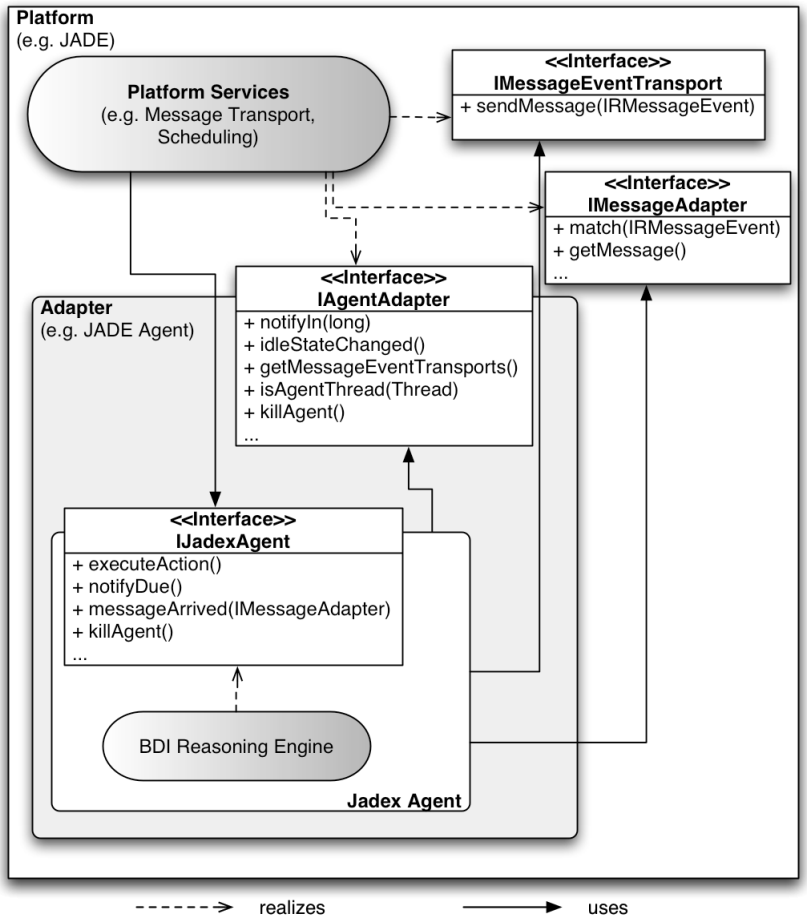

Abb. 6: Zusammenhang von Jadex-Agent, Adapter und Plattform

Der Agent bietet der Plattform nur die für seine Ausführung minimal nötige Schnittstelle an:

- `IJadexAgent.executeAction()` veranlasst den Agenten dazu, eine Aktion (exakt: einen Agendaeintrag) auszuführen. Welcher Natur diese Aktion ist, ist für die Plattform irrelevant. Der Rückgabewert dieser Methode gibt darüber Aufschluss, ob der Agent noch weitere Einträge in seiner Agenda hat, er also weitere Aktionen ausführen könnte.
- `IJadexAgent.notifyDue()` wird von der Plattform aufgerufen, um den Agenten wieder aufzuwecken.
- Mit `IJadexAgent.messageArrived(IMessageAdapter message)` werden dem Agenten eingehende Nachrichten übergeben.
- `IJadexAgent.killAgent()` schließlich informiert den Agenten darüber, dass er sich selbst beenden soll.

Ein Agent kann über seinen Adapter auf die Dienste der Plattform zurückgreifen. Die anzubietenden Dienste sind im Wesentlichen:

- `IAgentAdapter.notifyIn(long millis)` bietet dem Agent die Möglichkeit, sich nach einer bestimmten Zeit von der Plattform wieder aktivieren zu lassen. Diese Methode stellt somit das proaktive Verhalten sicher und wird gewöhnlich dann aufgerufen, wenn die Agenda des Agenten leer ist, er aber unabhängig von äußeren Einflüssen demnächst wieder selbst aktiv werden möchte.
- Mit `IAgentAdapter.idleStateChanged()` benachrichtigt ein Agent die Plattform darüber, dass er aufgrund einer Zustandsänderung wieder ausgeführt werden möchte. Diese Methode wird vom Agenten üblicherweise nach einem Nachrichteneingang oder einer durch eine durch `IJadexAgent.notifyDue()` bedingte Benachrichtigung aufgerufen und soll im Fall der Standardpläne die für die Planausführung verantwortlichen Threads wieder aufwecken.
- `IAgentAdapter.getMessageEventTransports()` stellt dem Agenten die auf der Plattform verfügbaren Kommunikationskanäle zur Verfügung.
- Mit `IAgentAdapter.isAgentThread(Thread thread)` kann ein Agent prüfen, ob ein Thread derjenige ist, der für die Ausführung des Agenten zuständig ist. Damit soll die Synchronisierung des Agenten garantiert und inkonsistente Zustände vermieden werden.
- `IAgentAdapter.killAgent()` wird vom Agenten aufgerufen, um die ihn betreffenden von der Plattform verwendeten Ressourcen wieder freizugeben. Dieses geschieht nur im Fall, dass der Agent beendet werden möchte beziehungsweise beendet werden soll.

Das Interface `jadex.adapter.IMessageEventTransport` bietet als einzige Methode das Senden von Nachrichten an. Auf der anderen Seite ermöglicht `jadex.adapter.IMessageAdapter` den Zugriff auf die Inhalte einer eingehenden Nachricht.

Während `jadex.adapter.IJadexAgent` von der der Reasoning-Engine realisiert wird, müssen Adapter `jadex.adapter.IAgentAdapter` implementieren, um ihre Dienste der BDI-Engine zugänglich zu machen.

2.6 Zusammenfassung

Mit der Entwicklung von komplexen Agenten wird die innere Repräsentation der Agenten als Strukturierungshilfe immer wichtiger. Hier bietet das am menschlichen Handeln angelehnte BDI-Modell Unterstützung, indem es den Agenten in kleinere Module strukturiert, die in festgelegter Art und Weise miteinander interagieren.

Jadex ist ein Vertreter der BDI-Modelle für Agenten und wurde ursprünglich als Erweiterung einer bestehenden Agentenplattform entwickelt, um diese Plattform mit einem kognitiv mächtigeren Agentenmodell auszustatten. Wo viele Agentensysteme auf spezielle Sprachen zur Beschreibung der Agenten setzen, übernimmt Jadex verbreitete Techniken: Der Agent selbst wird mit

Hilfe von XML modelliert. Für die Pläne der Agenten werden einfache Java Klassen benutzt, die prozedural einzelne Handlungsmöglichkeiten der Agenten beschreiben.

Im Laufe der Entwicklung entschieden sich die Entwickler, Jadex neu zu strukturieren und es als eigenständige BDI Reasoning-Engine weiter zu entwickeln. Über eine eigens entwickelte Schnittstelle soll gewährleistet werden, dass sich Jadex in prinzipiell beliebige Agentenplattformen integrieren lässt. Die Plattformen müssen nur bestimmte Basisdienste zur Verfügung stellen, die in dieser Schnittstelle definiert werden.

Kapitel 3

Middleware

Nachdem die Agententechnologie im Allgemeinen und Jadex im Speziellen näher erklärt wurden, bietet dieses Kapitel einen Überblick über Middleware-Technologien.

Dazu wird zuerst die Notwendigkeit von Middleware anhand bestimmter, immer wiederkehrender Unternehmensanforderungen erläutert. Nachfolgend wird ein Modell vorgestellt, dass die Entwicklung von Unternehmenssoftware vereinfacht und in Java EE Anwendung findet. Anschließend wird auf die Applikationsservern zugrunde liegenden Technologien eingegangen. Das Kapitel schließt mit einer umfassenden Erklärung der in der Java EE Spezifikation festgelegten Architektur.

3.1 Hintergrund

Mit der zunehmenden Konkurrenz in der IT-Branche sind Innovativität und Anpassungsfähigkeit obligatorische Eigenschaften für den Erfolg eines Unternehmens geworden. Firmenstrukturen müssen sich den Wettbewerbsbedingungen anpassen, um am Markt überlebensfähig zu sein. Diese Umgestaltungen hinterlassen in den Informationssystemen der Unternehmen ihre Spuren. Neue Hardware und Software wird eingekauft und in die IT-Infrastruktur des Unternehmens integriert. Bestehende Systeme werden weiterentwickelt und an die neuen Gegebenheiten angepasst.

Diese Prozesse führen dazu, dass die größtenteils monolithische IT-Infrastruktur im Laufe der Zeit einer verteilten, heterogenen Struktur gewichen ist. Die damit einhergehende gestiegene Komplexität der Entwicklung interoperabler Geschäftsanwendungen auf der Basis von unterschiedlicher Rechnerplattformen, Betriebssystemen und Netzwerkprotokollen stellt ein Unternehmen vor neue Herausforderungen und bedeutet in der Regel einen nicht unwesentlichen Mehraufwand für Entwickler und Administratoren.

3.2 Anforderungen an Geschäftsanwendungen

Nach [BaGa02] gibt es eine Reihe von nichtfunktionalen[9] Anforderungen an Anwendungen, die in Geschäftsanwendungen[10] zum Einsatz kommen:

- Bei Geschäftsdaten handelt es sich meistens um unternehmenskritische Informationen. Daher werden an **Datenhaltung** und **Zugriff** besonders hohe Ansprüche gestellt. In der Regel werden deshalb Datenbankmanagementsystemen (DBMS) eingesetzt.
- Auch wenn SQL (Structured Query Language) einen Datenbank-unabhängigen Zugriff erlaubt, müssen die erhaltenen Daten zumindest bei den weit verbreiteten, relationalen DBMS noch auf Variablen der Programmiersprache abgebildet werden. Im Fall der objektorientierten Sprachen sind dies Objekte beziehungsweise Objekthierarchien. **Objekt-relationale Mapper (O/R Mapper)** verbinden relationales und Objektmodell, indem sie dem Entwickler eine einheitliche, objektorientierte API für den Datenzugriff anbieten, die transparent für den Entwickler auf der Datenbank arbeitet.
- Die **Konsistenz** der Daten wird zwar durch die ACID-Eigenschaft des DBMS garantiert, sie gilt aber immer nur für lokale Daten. Sind mehre-

9 *Funktionale* Anforderungen legen fest, welche Dienste das System anbieten soll. Sie definieren also die Funktion des Systems. *Nichtfunktionale* Anforderungen sagen dagegen etwas über die Qualität der Dienste aus. Darunter fallen zum Beispiel Performanz, Verlässlichkeit und Sicherheit. Da diese Eigenschaften nichts mit der eigentlichen Funktion des Systems zu tun haben, nennt man sie auch nichtfunktional.

10 Geschäftsanwendungen (*Business Applications* oder *Enterprise Applications* genannt) dienen der Durchführung und Unterstützung von Geschäftsprozessen eines Unternehmens. Dazu gehören zum Beispiel Bestellerfassungen und Personalverwaltung.

re Datenbanken in einer Transaktion beteiligt, muss der Abschluss einer Transaktion koordiniert werden. Diese Aufgabe wird von Transaktionsmonitoren übernommen, die nach dem DTP-Modell (siehe Kapitel 3.4.4) eine verteilte Transaktionsverwaltung sicherstellt.

- Abhängig von den Aufgaben des Benutzers muss dieser unterschiedliche Rechte haben, um mit dem System interagieren zu können. **Authentifizierung** und **Autorisierung** sind deswegen integraler Bestandteil der meisten Geschäftsanwendungen.
- **Performace** (dt. Leistung, Durchsatz) ist eine weitere nichtfunktionale Eigenschaft, welche meist durch Caching (eng. Zwischenspeichern) häufig gebrauchter Daten erreicht wird.
- Wenn ein Unternehmen wächst und die Leistung trotz Optimierung des Systems nicht mehr ausreicht, ist **Skalierbarkeit** gefordert. Durch Verteilung der Arbeit auf mehrere Rechner (so genanntes *Clustering*) kann die Anwendung theoretisch beliebig große Lasten verarbeiten.
- Mit der Verteilung der Last geht eine Redundanz von Rechenkapazität einher, so dass Ausfälle von Systemen unsichtbar für den Benutzer kompensiert werden können. Man spricht in diesem Fall von **ständiger Verfügbarkeit**.
- **Modularisierung** und **Schichtenbildung** innerhalb der Geschäftsanwendung sind letztendlich Anforderungen an die Architektur der Anwendung, die externe Abhängigkeiten reduzieren und die Wartbarkeit verbessern sollen. So sollte zum Beispiel ein Wechsel des verwendeten DBMS so gut wie keine Modifikation der Anwendung nach sich ziehen.
- Letztlich müssen Geschäftsanwendungen **administrierbar** sein. Dazu gehören Werkzeuge, mit denen sich der Systemzustand überwachen und das System als solches verwalten lässt. Üblicherweise werden Protokollierungsmechanismen eingesetzt, die wichtige Systemereignisse aufzeichnen. Ein Administrator kann im nachhinein dieses Protokoll auswerten und dadurch mögliche Probleme erkennen. Andere Werkzeuge ermöglichen es ihm, Einstellungen am System vornehmen kann, um etwa Optimierungen durchzuführen.

3.3 Komponentenbasierte Softwareentwicklung

Der in der Literatur ebenso häufig verwendete wie unterschiedlich definierte Begriff einer *Komponente* soll im Rahmen dieser Arbeit wie folgt definiert werden:

> Eine (Software-)Komponente ist ein vorgefertigter, binärer Softwarebaustein mit einer anwendungsorientierten, semantisch kohärenten Funktionalität. Die strikte Kapselung der Implementierung und die damit verbundene Black-Box-Wiederverwendung führt zu einer gewissen Selbständigkeit der Komponente und ermöglicht somit eine lose Kopplung der Komponente in einer verteilten Umgebung. Mittels vertraglich vereinbarter Schnittstellen

> können kooperierende Komponenten auf die angebotene Funktionalität zugreifen. ([GrTh00], S. 292)

Aufgrund der Kapselung sollten Komponenten nicht oder zumindest nur im geringen Umfang von anderen Komponenten abhängig sein. Dies bedeutet allerdings, dass für jede Komponente die in Kapitel 3.2 genannten Anforderungen redundant implementiert werden müssen. Die entstehenden Komponenten wären zwar unabhängig und damit robust, sie sind aber schwergewichtig, da sie alle benötigten, nichtfunktionalen Eigenschaften selbst realisieren. Für den Komponentenentwickler bedeutet dies, dass er neben der Implementierung der Geschäftslogik auch nichtfunktionale Anforderungen umsetzen muss – ein zusätzlicher Aufwand, den man gerne vermeiden würde. Die fehlende Trennung dieser beiden Belange verschlechtert darüber hinaus die Wartbarkeit und Verständlichkeit des Quelltextes.

Die Alternative besteht darin, allen Komponenten gemeinsame Aufgaben auszulagern. Die Komponenten sind dadurch leichtgewichtiger und der Entwickler könnte sich auf die Implementierung der Geschäftlogik konzentrieren, während die technische Realisierung wie beispielsweise Persistenz bei der Entwicklung außen vor blieben. Dies führt allerdings zu einen deutlich höherem Grad der Abhängigkeit, da externe Funktionalität in die Komponenten eingebunden wird (vgl. [Szyp98]).

In der Praxis wird in der Regel ein Kompromiss gewählt, bei dem die Komponenten in einer Laufzeitumgebung ausgeführt werden. Die dazu notwendigen Abhängigkeiten beschränken sich damit auf diese Umgebung. Man spricht in diesem Zusammenhand von einem *Komponentenmodell*, da die Umgebung den Kontext des Einsatzes von Komponenten festlegt. Dieses wird nach [GrTh00] wie folgt definiert:

> Ein Komponentenmodell legt den Rahmen für die Entwicklung und Ausführung von Komponenten fest, der strukturelle Anforderungen hinsichtlich Verknüpfungs- beziehungsweise Kompositionsmöglichkeiten sowie verhaltensorientierte Anforderungen hinsichtlich Kollaborationsmöglichkeiten an die Komponenten stellt. Darüber hinaus wird durch ein Komponentenmodell eine Infrastruktur angeboten, die häufig benötigte Mechanismen wie Persistenz, Nachrichtenaustausch, Sicherheit und Versionierung implementieren kann. ([GrTh00], S. 293)

Im Gegensatz zur *aspektorientierten* Programmierung (engl. Aspect-oriented Programming (AOP)) ist bei der komponentenbasierten[11] Entwicklung die Erweiterbarkeit der nichtfunktionalen Dienste nicht gewährleistet. Üblicherweise legt man sich mit der Wahl eines Komponentenmodells auf dieses und

11 *Komponentenbasiert* und *komponentenorientiert* haben nach [Mos03] dieselbe Semantik. Während man mit *komponentenorientiert* aber meist einen Entwicklungsprozess oder das Paradigma an sich bezeichnet, spricht bei einer nach diesem Paradigma entwickelten Software von *komponentenbasierter* Software.

die von ihm erbrachten, nichtfunktionalen Dienste fest. AOP ermöglicht dagegen die transparente Nutzung modularer, *Aspekte* genannter Komponenten, die Cross-Cutting Concerns (dt. übergreifende Anforderungen) realisieren. Zu diesen Concerns zählen beispielsweise auch nichtfunktionale Anforderungen. AOP-Frameworks bringen in der Regel einen Satz vordefinierter Aspekte mit, sie ermöglichen aber ebenso die Entwicklung neuer Aspekte.

3.4 Middleware-Technologien

Da es sich bei der Java Platform Enterprise Edition (siehe Kapitel 3.5) um eine Architektur handelt, die ihren Ursprung in klassischen Middleware-Technologien hat, soll zuerst ein kurzer Überblick über die Anfänge von Middleware-Systemen gegeben werden.

3.4.1 Remote Procedure Call

Bei Remote Procedure Calls (RPC) handelt es sich um Protokolle zum synchronen Aufruf von Programmfunktionen über Rechnergrenzen hinweg, die neben den TP-Monitoren (siehe Kapitel 3.4.4) eine der ältesten Middleware-Technologien ist. Ursprünglich von Sun Microsystems etwa 1970 unter dem Namen ONC (Open Network Computing) RPC für das Network File System (NFS) entwickelt, stellte ONC RPC eines der ersten, von der IETF (Internet Engineering Task Force) standardisierten[12] Protokolle dar, das nach dem Client-Server Modell transparentes, verteiltes Rechnen ermöglicht. Um die Schnittstelle eines Dienstes unabhängig von der eingesetzten Programmiersprache veröffentlichen zu können, musste diese mit einer deklarativen Schnittstellenbeschreibungssprache – Interface Description Language (IDL) genannt – beschrieben werden. Neben dem von Sun entwickelten ONC RPC entstanden noch einige andere zueinander inkompatible RPC Protokolle.

3.4.2 Object Request Broker

Object Request Broker (ORB) erweitern das RPC-Konzept von verteilten Prozeduraufrufen um die Möglichkeit einer Kommunikation zwischen verteilten Objekten. Die OMG (Object Management Group)[13] entwickelte 1991 den ersten international anerkannten Standard für einen ORB: Die so genannte Commen Object Request Broker Architecture (CORBA) legt alle für den plattform- und sprachunabhängigen Einsatz von verteilten Objekten erforderlichen Details fest. Zur formalen Spezifikation der Klassen und ihrer Methoden existiert eine IDL. Das General Inter-ORB Protocol (GIOP) bildet das Kommunikati-

12 Siehe RFC 1057 (`http://www.ietf.org/rfc/rfc1057.txt`) und RFC 1831 (`http://www.ietf.org/rfc/rfc1831.txt`)

13 Die OMG ist ein 1989 gegründetes Konsortium namhafter IT-Hersteller wie IBM, Apple und Sun zur Erstellung herstellerunabhängiger Standards.

onsprotokoll[14], welches auf unterschiedlichen Transportprotokollen eingesetzt werden kann. Am weitesten verbreitet ist dabei der Einsatz von GIOP über TCP/IP, das deswegen auch Internet Inter-ORB Protocol (IIOP) genannt wird (vgl. [PuRP06]).

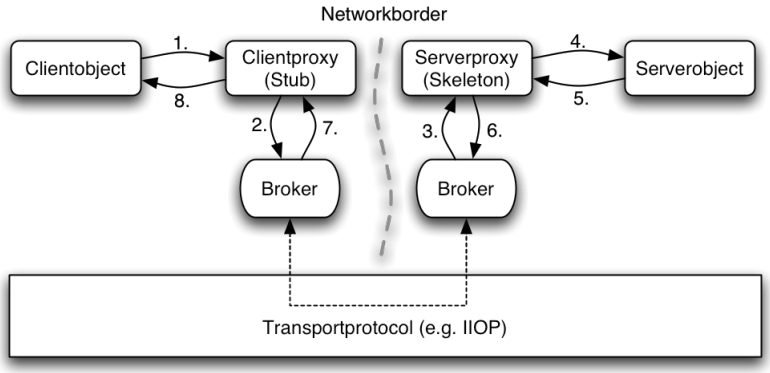

Abb. 7: Kommunikation nach dem Proxy-Entwurfsmuster

Neben CORBA existieren noch weitere ORB. Die bekanntesten von ihnen sind das von Microsoft entwickelte Distributed Component Object Model (DCOM), Java-RMI (Remote Method Invocation) von Sun und RMI-IIOP (RMI über IIOP), das mit CORBA interoperabel ist. Auch wenn alle diese Technologien noch so verschieden sind, haben sie doch eines als Eigenschaft gemeinsam: Die Zugriff auf die verteilten Objekte geschieht weitgehend transparent, das heißt der Aufruf einer Methode an einem entfernten Objekt unterscheidet sich nicht von einem Aufruf an einem lokalen Objekt. Zur Umsetzung der Transparenz bedienen sich alle Technologien dem so genannten Proxy-Pattern (vgl. [GaHJ97]): Wie aus Abbildung 7 ersichtlich wird, existiert für das entfernte Objekt ein lokaler Stellvertreter, Client-Proxy oder Stub genannt. Dieser Stellvertreter besitzt genau dieselbe öffentliche Schnittstelle wie das Zielobjekt und wurde aus der IDL eines angebotenen Dienstes erstellt. Statt aber bei einem Aufruf etwas lokal zu berechnen, wird der Aufruf kodiert (*marshalling* genannt) und an den lokalen ORB weitergeleitet, welcher ihn über einen geeigneten Transportkanal an den Zielrechner schickt. Dort nimmt ein weiterer ORB die Anfrage entgegen und reicht sie an den Proxy des entsprechenden Zielobjektes weiter. Dieser serverseitige Stellvertreter dekodiert den Aufruf (*unmarshalling* genannt) wieder und ruft letztendlich die Methode am Zielobjekt auf. Etwaige Rückgabewerte werden in dem eben geschilderten Verfahren zurück geschickt.

Da einige andere Unternehmen wie etwa Microsoft mit DCOM ebenfalls eigene ORB-Architekturen entwickelten, machte sich Anfang 2002 das W3C (World Wide Web Consortium) ebenfalls daran, einen Standard für verteilte

14 CORBA 1.0 definierte noch kein Protokoll zur Kommunikation von Objekten, so dass ORB-Implementierungen unterschiedlicher Hersteller zueinander inkompatibel waren. Dieser Missstand wurde in CORBA 2.0 mit GIOP behoben.

Objekte unter dem Namen *Web Services* zu entwerfen. Durch den Einsatz gängiger Webtechnologien wie XML und HTTP (Hypertext Transfer Protocol) sollten die Web Services eine einfach einzusetzende und leicht erweiterbare Alternative zu den ressourcenhungrigen CORBA ORBs darstellen. So kommt zum Beispiel als IDL die XML-basierte Web Services Description Language (WSDL) zum Einsatz. Funktionsaufrufe werden meist nach dem Simple Object Access Protocol (SOAP) gekapselt (ebenfalls in XML) und in der Regel über HTTP verschickt [ACK03].

Zwar gibt es dank der Erweiterbarkeit der Web Services die Möglichkeit, die Defizite des bisherigen Standards zu bewältigen, allerdings sind die entsprechenden Standardisierungsprozesse für diese Erweiterungen – soweit sie überhaupt existieren – noch nicht abgeschlossen. Zu diesen Defiziten zählen zum Beispiel die fehlende Unterstützung von Transaktionen[15] und die bisherige Zustandslosigkeit eines Web Services.[16] So kommt es, dass trotz weit reichender Akzeptanz der Web Services CORBA, DCOM und ihre Vertreter noch nicht obsolet geworden sind [KuWö02].

3.4.3 Message-oriented Middleware

Die bisher beschriebenen Technologien setzen alle auf transparente, synchrone Methodenaufrufe. In Abbildung 8 wird dieses Konzept dargestellt. Für den Entwickler macht es demnach keinen Unterschied, ob er eine Methode an einem lokalen Objekt (a) oder an einem entfernten Objekt (b) aufruft. Beide Aufrufe blockieren den Aufrufer solange bis der Methodenaufruf abgeschlossen ist. Während aber im lokalen Fall der die Methode aufrufende Thread ebenfalls die Berechnung des Methodenaufrufs durchführt, wird der im verteilten Fall der aufrufende Thread angehalten. Auf dem Rechner des Zielobjekts muss dagegen ein Thread bereitgestellt werden, der die Berechnung durchführt und den Rückgabewert zurückschickt. Der ursprüngliche Thread wird demnach wieder aktiviert und setzt seinen Programmfluss mit erhaltenem Rückgabewert fort.

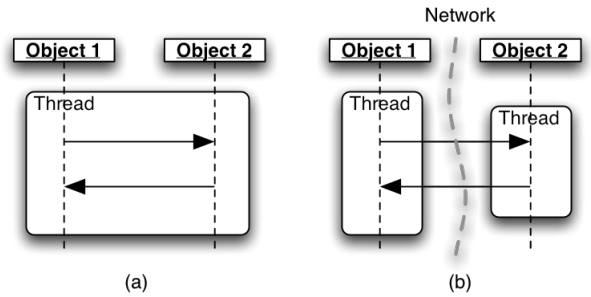

Abb. 8: Synchrone Kommunikation

15 Web Service Atomic Transaction ist eine Spezifikation, die sich diesem Defizit annimmt.
16 Die Lösung dieses Problem wird vom Web Service Resource Framework angegangen.

In manchen Fällen mag es sinnvoll sein, statt das Ende der entfernten Berechnung abzuwarten direkt weitere Berechnungen durchzuführen. Dies ist zum Beispiel der Fall, wenn der Informationsfluss nur in eine Richtung stattfindet, wie etwa bei einer Methode ohne Rückgabewert. Da der aufrufende Thread kein Ergebnis des Methodenaufrufes verwerten kann, könnte er ohne zu blockieren direkt weiterarbeiten. Selbst wenn der Aufgerufene eine Antwort zurückschickt, kann der Aufrufer die Zeit bis zum Eintreffen der Antwort für weitere Berechnungen nutzten, die natürlich unabhängig vom Ausgang des Aufrufs sein müssen.

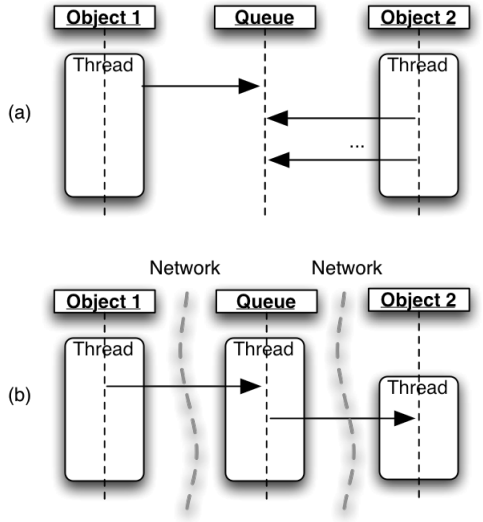

Abb. 9: Asynchrone Kommunikation mittels Queue

Da diese Art der Kommunikation in der Regel nicht über die Methoden einer Schnittstelle abgewickelt wird, kommt eine bestimmte API für die asynchrone Kommunikation zum Einsatz. Aufgrund der Asynchronität des Informationsaustauschs spricht man meistens von einer Nachrichten-orientierten Kommunikation. Abbildung 9 verdeutlicht die Unterschiede zum synchronen Aufruf. Demnach gibt es eine Instanz, die die Nachricht eines Aufrufers entgegen nimmt und sie an das Ziel weiterleitet, wenn dieses Zeit für die Bearbeitung der Nachricht hat. Da diese Instanz die Nachrichten zwischenspeichert, wird sie Message-Queue (dt. Nachrichtenwarteschlange) genannt. Im lokalen Fall (a) würde für eine Implementierung ein synchronisiertes Objekt reichen, auf das von zwei Threads zugegriffen werden kann. Die Threads müssen dabei selbständig prüfen, ob für sie eine Nachricht vorliegt. In einem verteilten System (b) gibt es dagegen eine eigenständige Anwendung, welche die Nachrichtenpufferung übernimmt und die Teilnehmer einer Kommunikation bei Bedarf über nicht verarbeitete Nachrichten informieren kann (*pushing*). Damit müssen Empfänger nicht regelmäßig die Queue auf neue Nachrichten hin überprüfen (*polling*).

Systeme, die eine solche Kommunikation unterstützen, werden allgemein als Message-oriented Middleware (MOM) oder als Message Broker (dt. Nachrichtenvermittler) bezeichnet. Bekannte Vertreter dieser Art von Middleware sind MQSeries[17] von IBM und MSMQ[18] von Microsoft. In der Regel unterstützen MOM-Produkte die persistente Pufferung von Nachrichten zum Beispiel durch Datenbanken. Dieses erhöht die Dienstqualität, da gepufferte Nachrichten im Falle eines Programmfehlers nicht verloren gehen. Nicht sofort zustellbare Nachrichten können gespeichert werden, bis die Empfänger der Nachrichten wieder erreichbar sind. Auf diese Weise können Empfänger und Sender von Nachrichten nicht nur räumlich, sondern auch zeitlich voneinander getrennt sein.

3.4.4 Transaktionsmonitore

In großen Systemen ist es nicht ungewöhnlich, dass mehrere Datenbanken zum Einsatz kommen. Die Datenverteilung geht mit einer Lastverteilung einher und im Falle der redundanten Datenhaltung würde mit jeder weiteren Datenbank ebenfalls die Ausfallsicherheit erhöht werden. Manchmal müssen Daten aber einfach aus gesetzlichen Gründen verteilt gespeichert werden. Unabhängig von den Gründen für diese Verteilung ist das Problem, dass der Zugriff auf diese Datenbestände meistens transaktionsgesichert ablaufen muss. Jedes DBMS für sich unterstützt durch die ACID-Eigenschaft Transaktionen. Falls der Datenzugriff über mehrere Datenbanken erfolgt, muss eine Instanz diesen Zugriff verwalten und kontrollieren. Veranschaulichen kann man sich das an dem einfachen Beispiel einer Überweisung: Das Abbuchen eines Betrages von einem Konto und das Gutschreiben desselben auf ein anderes müssen in einer Transaktion stattfinden. Wird die Konten bei verschiedenen Banken geführt, nehmen die DBMS beider Banken an der Transaktion teil. Man spricht in diesem Fall von einer verteilten Transaktion.

Programme, die verteilte Transaktion ermöglichen, werden gemeinhin als TP-Monitore (Transaction Processing Monitor) bezeichnet. Sie koordinieren unter anderem den transaktionsorientierten Zugriff auf Geschäftsdaten, sind für die Zugriffsrechte verantwortlich, stellen die Kommunikation zwischen allen beteiligten Systemen sicher, dienen zur Lastverteilung und müssen letztendlich nach einem Systemfehler einen konsistenten Betriebszustand wiederherstellen können. In dieser Funktion sind sie zwischen Anwendungen und Datenbanken angesiedelt und gelten deshalb als Middleware. Die Koordination zwischen den Datenbanken basiert häufig auf dem DTP-Modell (Distributed Transaction Processing) von X/Open[19], das verteilte Transaktionen ermöglicht. Bekannte TP-Monitore sind Tuxedo von Bea Systems sowie das Customer Infor-

17 `http://www.ibm.com/software/mqseries/`
18 `http://www.microsoft.com/msmq/`
19 Bei der X/Open handelte es sich um ein 1984 gegründetes Konsortium, das offene Industriestandards entwickeln sollte.

mation Control System (CICS) von IBM, welches seit 1968 entwickelt wird und als der älteste TP-Monitor gilt.

Eine Weiterentwicklung des TP-Monitors ist der Object Transaction Monitor (OTM). Dieser stellt eine Kombination eines klassischen TP-Monitors mit einem ORB dar. Da ein OTM für Komponenten-orientierte Ansätze verwendet werden kann, bezeichnet man ihn häufig als Component Transaction Monitor (CTM). Der Microsoft Transaction Server (MTS) war das erste veröffentlichte Produkt dieser neuen Generation von Middleware. Mittlerweile ist es in COM+ aufgegangen (vgl. [BaGa02]).

3.4.5 Applikationsserver

Mit der Vereinigung von Komponentenserver und TP-Monitoren bilden die CTM die Grundlage für so genannte Applikationsserver. Diese stellen eine Laufzeitumgebung für serverseitige Komponenten zur Verfügung und beinhalten zahlreiche Dienste. Die Komponenten wiederum implementieren mit Hilfe dieser Umgebung die Funktionalität der Geschäftslogik. Mehrere kohärente Komponenten können in Einheiten zusammengefasst werden, welche sich als ganzes auf einen Applikationsserver installieren (*Deployment* genannt) lassen. Die von den Komponenten benötigten Dienste werden deskriptiv vor der Installation auf dem Server festgelegt. Nach [BaGa02] zählen dazu etwa:

- **Verwaltung der installierten Komponenten:** Der Applikationsserver muss die von den Benutzern angeforderten Komponenten instanziieren und sie bei Anfragen immer wieder dem richtigen Benutzer zuordnen. Bei großen Komponentenzahlen muss er darüber hinaus noch häufig angeforderte Komponenten puffern und weniger benutzte bei Bedarf auslagern, um Speicher zu sparen.
- **Kommunikation:** Der Server muss über geeignete Protokolle angesprochen werden können, um von Benutzeranwendungen ebenso wie von anderen Komponenten erreichbar zu sein. So kommt bei Java EE - Applikationsservern zum Beispiel RMI-IIOP zum Einsatz.
- **Sicherheit:** Der Applikationsserver muss den Zugriff auf verwaltete Ressourcen sichern. Dazu gehören eine Benutzerverwaltung samt Authentifizierung sowie die Möglichkeiten, die Benutzer für bestimmte Komponenten zu autorisieren.
- **Verfügbarkeit:** Fehler in der Ausführung müssen erkannt und protokolliert werden, so dass ein Entwickler zu einem späteren Zeitpunkt den Fehler überprüfen und lokalisieren kann. Ebenso muss sichergestellt werden, dass sich ein Fehler nicht auf Komponenten anderer Benutzer auswirkt.
- **Transaktionssicherung:** Applikationsserver enthalten meist einen traditionellen TP-Monitor, der lokale wie verteilte Transaktionen überwacht und somit die Integrität der Daten gewährleistet.

3.5 Java Platform, Enterprise Edition

Java EE und J2EE sind Spezifikationen, die innerhalb des Java Community Process (JCP) von mehreren Unternehmen erarbeitet und anschließend veröffentlicht werden.

Mit der Bezeichnung *Applikationsserver* bezeichnet man oft Java EE konforme Applikationsserver, was auf die derzeitige Dominanz von Java EE am Markt zurück zu führen ist. Der Vollständigkeit halber sollte aber erwähnt werden, dass es auch andere Applikationsserver gibt. So verfolgt etwa Microsofts mit den *.NET Enterprise Services* eine Java EE ähnliche Strategie. Leider existieren außer der Referenzimplementierung von Microsoft, die in Ermangelung äquivalenter .NET Portierungen noch zum großen Teil auf den alten COM+-Diensten basiert und dem freien Applikationsserver *Yuhana*[20], der noch in der Entwicklung befindet, noch keine weiteren Applikationsserver für .NET.

Da Jadex in Java geschrieben ist, bietet sich ein ebenfalls in Java geschriebener Applikationsserver für eine Integration an. Deshalb wird im Folgenden das Java EE Framework detaillierter erklärt.

3.5.1 3-Schichten Architektur

Java EE sieht eine 3-Schichten-Architektur für Geschäftsanwendungen vor, die eine bessere Wartbarkeit als eine monolithische Anwendung verspricht und außerdem den Entwicklungsprozess natürlich zergliedert.

1. Die Client-Schicht umfasst die Benutzeroberfläche, welche die Interaktion mit dem Benutzer ermöglicht. In der Regel werden für die Darstellung Java-Anwendungen, Java-Applets und von Webbrowsern darstellbare HTML-Seiten gewählt.
2. In der mittlere Schicht (engl. middle tier) läuft die in Komponenten umgesetzte Geschäftslogik. Außerdem werden hier die von den Komponenten benutzten, nichtfunktionalen Dienste (siehe Kapitel 3.2) zur Verfügung gestellt. Gegebenenfalls wird in dieser Schicht ebenfalls die Aufbereitung der Daten durchgeführt, da zum Beispiel Webbrowser nur in begrenztem Maße Daten für die Darstellung aufbereiten können. Wenn die Unterstützung durch die Webbrowser sichergestellt ist, könnte die eXtensible Stylesheet Language (XSL) in absehbarer Zeit eine Lösung dieses Problems bieten, da mit ihr Daten und Darstellung derselben absolut unabhängig voneinander zum Browser übertragen werden können.
3. Die Datenschicht (Backend-Schicht genannt) besteht aus einer oder mehreren Datenbanken, beziehungsweise allgemeiner formuliert Enterprise Information Systems (EIS) zu denen etwa auch SAP-Installationen und Enterprise Resource Planing-Systems (ERP) zählen. Diese Schicht persistiert die Daten der Geschäftslogik, sorgt für ihre lokale Konsistenz und optimiert ihren Zugriff.

20 `https://sourceforge.net/projects/yuhana/`

Neben den oben angeführten, offensichtlichen Vorteilen dieser Aufteilung der Architektur gibt es auch weniger offensichtliche: So könnte man annehmen, dass der aus der Schichtenbildung resultierende erhöhte Kommunikationsaufwand die Reaktionszeiten der Benutzeroberfläche verschlechtert. In der Realität sieht es aber meist so aus, dass mittlere und Datenschicht in physikalischer Nähe zueinander befinden, so dass die Latenz zwischen diesen beiden Schichten eher gering ist. Außerdem bereitet die mittlere Schicht nur die für die Clients notwendigen Daten auf und verschickt sie zusammen, was der Reaktionszeit zu gute kommt. In einer zweischichtigen Architektur, in der die Clients selbst die gesamte Geschäftslogik implementieren und eigenständig auf die Datenschicht zugreifen, werden dagegen eventuell überflüssige Daten geschickt oder es müssen mehrere Anfragen hintereinander an die Datenschicht gestellt werden um die notwendigen Daten zu bekommen. All diese Faktoren sorgen letztlich dafür, dass die 3-Schichten Architektur in der Regel schneller reagiert als eine zweischichtige. Davon abgesehen bieten mehrschichtige Architekturen eine bessere Skalierbarkeit, da die einzelnen Schichten nahezu unabhängig von den anderen sind und sie somit über mehrere Rechner verteilt werden können.

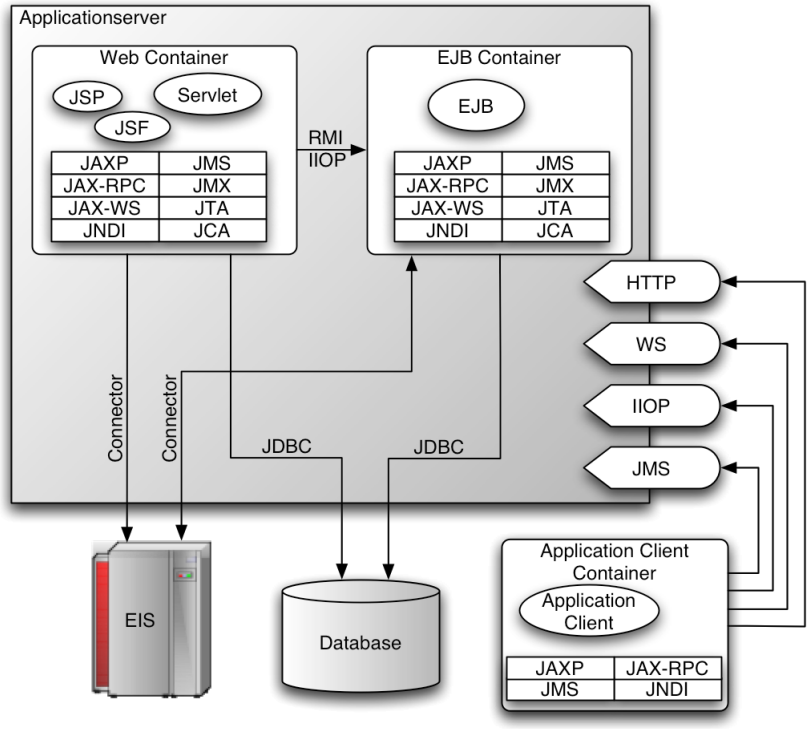

Abb. 10: Java EE Architekturübersicht nach [BCEHJ06]

Abbildung 10 zeigt die Realisierung dieser 3-schichtigen Architektur. Der Applikationsserver arbeitet mit Hilfe von JDBC (Java Database Connectivity) und herstellerspezifischen Konnektoren auf Datenbanken und anderen EIS.

Üblicherweise sollte der Backend-Zugriff nur über Enterprise JavaBeans geschehen, die Architektur ermöglicht den Zugriff auf EIS aber auch von anderen Komponenten wie etwa Servlets. Mit über HTTP verschickten HTML-Dokumenten kann der Applikationsserver Daten mit einfachen Browsern austauschen. Kommt eine eigens entwickelte Clientsoftware zum Einsatz, greift der Applikationsserver dagegen auf RMI-IIOP oder SOAP über HTTP als Kommunikationsmittel zurück.

Intern gliedert sich ein Applikationsserver in zwei Container: Der Enterprise JavaBean Container stellt die schon beschriebene Laufzeitumgebung für die Serverkomponenten (siehe Kapitel 3.5.3 und Kapitel 3.5.4) bereit, die die Geschäftslogik implementieren. Er ist für die Verwaltung aller EJBs verantwortlich und stellt ihnen über bestimmte Schnittstellen seine Dienste bereit.

Der Web Container ist eine Laufzeitumgebung für die dynamische Generierung von HTML-Dokumenten. Dieser Container wird in der Regel benötigt, wenn die Clients über Browser auf den Applikationsserver zugreifen und die angeforderten Daten nicht selbst aufbereiten können. Für die Erstellung der HTML-Dokumente kann ein Entwickler Technologien wie Servlets, JSP und JSF einsetzen. In der Regel greift der Web Container über EJBs auf die Geschäftsdaten zu.

Der so genannte Application Client Container (ACC) ist im eigentlichen Sinne kein Container. Vielmehr stellt er die Java Bibliotheken bereit, die von der Clientsoftware für den Zugriff auf den Applikationsserver benötigt werden. Damit ist der ACC gewissermaßen die Laufzeitumgebung für den Client.

3.5.2 Bestandteile der Spezifikation

Java EE gliedert sich in viele Teile, die ihrerseits Teilaspekte der gesamten Laufzeitumgebung spezifizieren, APIs festlegen und teilweise auch Test-Suites (Programme zum Testen der Spezifikationseinhaltung) und Benchmarks (Programme zur Leistungsbewertung) definieren. Im Folgenden wird ein ausgesuchter Überblick über die wichtigsten Spezifikationen und APIs gegeben und kurz erklärt, welche Bedeutung ihnen in der Architektur zukommt (vgl. [BCEHJ06]).

- **Enterprise JavaBeans (EJB)** bilden den Kern des Komponentenmodells von Java EE. Sie implementieren die Geschäftslogik und werden in Kapitel 3.5.3 und Kapitel 3.5.4 näher beschrieben.
- **Java Servlets** sind von `javax.servlet.http.HttpServlet` erbende Java Klassen, die innerhalb eines Webservers Clientanfragen entgegen nehmen und diese dynamisch bearbeiten. Die Abbildung zwischen Servlets und URL (Uniform Resource Locator) geschieht dabei über eine deskriptive Konfigurationsdatei im XML-Format. Der Webserver ruft anhand dieser Datei eine bestimmte Methode des entsprechenden Servlets mit den vom Client übergegebenen Parametern auf, welche die Anfrage verarbeitet. Zu den Anfangszeiten von Servlets wurde das komplet-

te Ergebnis der Anfrage (meist ein HTML-Dokument) in den Servlets zusammengesetzt.

- **JavaServer Pages (JSP)** sind die Antwort auf das oben beschriebene Problem der Servlets. Da in Servlets zusammengesetzte HTML-Antworten nicht dem MVC-Pattern (Model-View-Controller) folgen und damit zu unwartbaren Code führen, sollte mit der Einführung der JSP-Technologie die Darstellung (View) auf JSPs ausgelagert werden, während die Servlets nur noch die Geschäftlogik (Controller) enthalten sollten. JSP können dabei neben den für die Antwort notwendigen HTML-Tags so genannte Skriptlets enthalten. Dies sind kleine Java-Codefragmente, mit denen man zum Beispiel über Java Collections iterieren kann, um in der Antwort eine HTML-Liste zusammen zu setzen. Über eingebundene Tag-Bibliotheken können JSPs um spezielle Tags und damit auch Funktionalität erweitert werden. Technisch gesehen werden JSPs bei ihrem ersten Aufruf vom Webserver zu Servlets zusammengesetzt und anschließend kompiliert.
- Die **JavaServer Pages Standard Tag Library (JSTL)** ist eine Sammlung von standardisierten Tag-Bibliotheken. Diese bieten häufig benutzte Funktionalität wie Iteration, XML-Transformation und Internationalisierung an. So würde man um etwa auf das obige Beispiel der Erstellung einer HTML-Liste zurück zu kommen keine Skriptlet für eine Iteration benötigen, sondern könnte dies ebenso mit Hilfe der JSTL erledigen. Der Vorteil der JSTL und ähnlicher Tag-Bibliotheken liegt darin, dass man in den JSPs keine Javacode-Fragmente mehr einsetzen muss, um die Darstellung dynamisch anzupassen.
- **JavaServer Faces (JSF)** ist ein Framework, das die Entwicklung der GUI (Graphical User Interface) (dt. grafische Benutzerschnittstelle) vereinfacht. Dazu bietet es abstrakte GUI-Elemente an, die mit Hilfe von Renderern in unterschiedliche Ausgabeformate transformiert werden (z.B. HTML). Zu den Funktionen von JSF gehören unter anderem die Validierung der Benutzereingaben, Datenkonvertierung zwischen Darstellung der Daten und Datenmodell sowie extern konfigurierbare Seitennavigation. Während Servlets und JSPs auf die Trennung von View und Controller des MVC-Pattern abzielten, unterstützt JSF über Tag-Bibliotheken und eine dazugehörige API die Trennung zwischen Datenmodell (Model) und den beiden erstgenannten.
- **Java Message Service (JMS)** ist eine standardisierte API für den asynchronen Nachrichtenaustausch. Da JMS eine im Rahmen dieser Arbeit größere Bedeutung zukommt, wird es in Kapitel 3.5.5 näher erklärt.
- Die **Java Transaction API (JTA)** ermöglicht die standardisierte, vom DBMS unabhängige Nutzung von verteilten Transaktionen in Java.
- **JAXP (Java API for XML Processing)** bietet Unterstützung für die Verarbeitung von XML-Daten. Dazu gehört ebenso das Parsen (dt. zerlegen, analysieren) und Transformieren von XML-Dokumenten sowie der Zugriff auf das Dokument mittels DOM (Document Object Model) und SAX (Simple API for XML).

- **JAX-RPC (Java API for XML-based RPC)** ist eine Implementierung von SOAP und erlaubt den Zugriff auf verteilte Funktionen über HTTP.
- **JAX-WS (Java API for XML Webservices)** basiert auf JAX-RPC und spezifiziert wie innerhalb von Applikationsservern Webservices angeboten und genutzt werden können.
- Die **J2EE Connector Architecture (JCA)** [Sun03] ermöglicht es Unternehmen, ihre EIS-Produkte in Applikationsserver mit Hilfe von *Resource Adapters* genannten Komponenten zu integrieren. Vor der JCA-Spezifikation musste ein EIS-Anbieter für jeden unterstützten Applikationsservern einzeln eine Anbindung realisieren. JCA reduziert diesen Aufwand durch eine standardisierte Architektur und API und ist damit ein Ansatz der Enterprise Application Integration (EAI). Das Zusammenspiel zwischen Resource Adapter und Applikationsserver erfolgt über systemweite Vereinbarungen (engl. contracts), deren Implementierung von der JCA zwingend gefordert wird. Diese Vereinbarungen regeln über entsprechende APIs beispielsweise den Verbindungsaufbau, die Teilnahme an Transaktionen und die Zugriffberechtigungen auf das EIS an sich.
- **JNDI (Java Naming and Directory Interface)** ermöglicht den Entwicklern, Javaobjekte unter einem eindeutigen Namen zur Verfügung zu stellen. In dem lokalen Verzeichnis `java:comp/env` kann man zum Beispiel einzelnen EJBs Konfigurationseinstellungen oder Resource Adapter zugänglich machen, die sie für ihre Aufgabe benötigen. Im Gegensatz zu lokalen Verzeichnissen sind global angelegte Verzeichniseinträge netzwerkweit zugänglich. Üblicherweise werden EJBs global zugänglich gemacht, damit Clients ihre Funktionalität nutzen können. Weil JNDI nur die Schnittstellen für die Verzeichnisnutzung spezifiziert, können über entsprechende Implementierungen beliebige existierende Verzeichnisdienste wie zum Beispiel LDAP (Lightweight Directory Access Protocol) oder DNS (Domain Name Service) genutzt werden.

3.5.3 Enterprise JavaBeans 2.1

Enterprise JavaBeans (in der Version 2.1) sind Teil der von J2EE 1.4 spezifizierten Standardarchitektur für die Ausführung von serverseitigen Komponenten. Diese kapseln die Geschäftslogik. Um ihre Aufgaben zu erfüllen können EJBs wie in Kapitel 3.4.5 beschrieben auf vom Applikationsserver bereitgestellte Dienste zurückgreifen. Grundsätzlich gibt es drei Typen von EJBs, die innerhalb des EJB Containers laufen:

- **Entity Beans** modellieren die persistenten Geschäftsdaten. Um ihre eindeutige Identität zu gewährleisten, müssen sie einen Primärschlüssel (ein oder mehrere Attribute, die das Bean eindeutig identifizieren) besitzen. Über diesen Schlüssel oder mit Hilfe einer OQL ähnlichen Anfragesprache – EJB Query Language (EJB-QL) genannt – können die Entity Beans wieder aufgefunden werden. Wie die Geschäftsdaten persistiert wer-

den, obliegt dem Entwickler. Entweder sorgt er über die entsprechenden Lebenszyklusmethoden selbst für die Persistenz (Bean-managed Persistence (BMP)) oder er überlässt es dem Container, die Daten mit einem von ihm angefertigten Datenbank-Mapping zu sichern (Container-managed Persistence (CMP)). Um die Konsistenz der Geschäftsdaten sicher zu stellen, sollte der Zugriff auf Entity Beans nur innerhalb von Transaktionen erfolgen.

- **Session Beans** implementieren die Geschäftslogik. Typischerweise erledigen sie ihre Aufgaben, indem sie selbst oder andere von ihnen aufgerufene Session Beans Geschäftsdaten manipulieren. Es wird zwischen zustandslosen und zustandsbehafteten Session Beans unterschieden. Während erstere immer nur für genau einen Methodenaufruf genutzt werden, kann ein und dasselbe zustandsbehaftetes Session Bean über mehrere Aufrufe verwendet werden. Der Container ist für die Verwaltung der zustandsbehafteten Session Beans zuständig.
- **Message-driven Beans** werden als Empfänger von asynchronen Nachrichten eingesetzt. Neben JMS unterstützen sie prinzipiell beliebige MOM. Die Nutzung von JMS ist allerdings zu empfehlen, da der Container die Installation der Beans samt ihrer Registrierung an der entsprechenden Nachrichtenwarteschlange unterstützt. Per Spezifikation sind Message-driven Beans zustandslos, d.h. mehrere Nachrichten können nicht an dieselbe Instanz gerichtet werden.

Im Folgenden soll ein Überblick über die Bestandteile eines EJB gegeben werden. Ohne die Nutzung von Hilfswerkzeugen wie etwa XDoclet[21] obliegt die Erzeugung aller Bestandteile dem Entwickler.

- Das **Home-Interface** enthält alle den Lifecycle (dt. Lebenszyklus) des Beans betreffenden Methoden. Dazu gehören für alle Beantypen Methoden zur Erzeugung und zum Entfernen des EJB. Session Beans bieten darüber hinaus noch Methoden zur Aktivierung beziehungsweise Passivierung. Falls der Container zu viele zustandsbehaftete Session Beans verwalten muss, kann er die weniger benutzten Beans passivieren und die entsprechenden Beans danach zum Beispiel serialisiert auf der Festplatte auslagern. In der Passivierungsmethode muss das Session Bean alle Ressourcen schließen, die nicht serialisierbar sind (etwa Socket- oder JDBC-Verbindungen). Bei der Aktivierung können die Verbinden wieder aufgebaut werden. Entity Beans verfügen über zwei weitere Lifecycle-Methoden. Wenn der Entwickler BMP nutzt, kann er in diesen beiden Methoden den Beanzustand in die DB schreiben beziehungsweise diesen laden.

21 XDoclet vereinfacht die Entwicklung von EJB dahingehend, dass im einfachsten Fall nur noch die Beanklasse selbst erstellt werden muss. Alle anderen Bestandteile würde XDoclet erzeugen (`http://xdoclet.sourceforge.net/xdoclet/index.html`).

- Das **Komponenten-Interface** deklariert die eigentlichen Geschäftsmethoden, die von den Clients benutzt werden. So erzeugt ein Client etwa mit Hilfe des Home-Interface ein Session Bean und erhält eine Referenz auf das Bean zurück, dass das Komponenten-Interface implementiert.
- Komponenten- und Home-Interfaces kann es in zwei Varianten geben: **Lokale Interfaces** ermöglichen nur eine lokale Nutzung des Beans in derselben JVM (z.B. von anderen Beans). **Remote Interfaces** bieten dagegen einen netzweiten Zugang zum Bean (über RMI-IIOP). Die Unterscheidung zwischen den beiden Interfaces ist notwendig, da die lokale Nutzung der remote Interfaces einen erhöhten Rechenaufwand durch RMI-IIOP mit sich bringt. Üblicherweise werden Entity Beans nur mit lokalen Interfaces ausgestattet. Der Zugriff auf die Geschäftsdaten geschieht über Session Beans, die als Fassade [GHJV97] für die Entity Beans dienen und dementsprechend wahlweise lokale und/oder entfernte Interfaces verfügen.
- Die **Beanklasse** implementiert die eigentlich Logik, die in den Home- und Komponenten-Interface angegeben ist.
- Der **Deployment-Deskriptor** ist eine deskriptive, XML-konforme Konfigurationsdatei, die dem Container alle nötigen Information zur Installation (engl. deployment) liefert.

3.5.4 Enterprise JavaBeans 3.0

Die meisten Änderungen beim Versionssprung von J2EE 1.4 zu Java EE 5.0 betreffen die Enterprise JavaBeans. Wie im letzten Kapitel ersichtlich wurde, ist die Entwicklung von Serverkomponenten nach J2EE 1.4 sehr aufwendig. Neben der eigentlichen die Logik implementierenden Beanklasse muss der Entwickler noch bis zu vier Interfaces erstellen, die aber nicht direkt von der Beanklasse implementiert werden dürfen. Stattdessen stimmen ihre Methodennamen per Konvention mit denen der Beanklasse überein. Über den Deployment-Deskriptor werden alle Abhängigkeiten zwischen den EJBs festgehalten. Diese Separierung ist zwar wünschenswert, erhöht aber gleichzeitig den Wartungsaufwand (vgl. [EiMe05] und [Pan04]). Ein großer Teil dieser hinderlichen Details soll in Java EE 5.0 mit der EJB 3.0 Spezifikation behoben werden. Im Folgenden werden deshalb einige der Verbesserungen des neuen Komponentenmodells kurz erläutert (vgl. [DeKe05a]):

Die mit J2SE 5.0 eingeführte Möglichkeit, Klassen, Methoden und Instanzvariablen mit so genannten Annotations (dt. Anmerkungen, Notizen) zu versehen, wird in EJB 3.0 genutzt, um die vormals im Deployment-Deskriptor aufgeführten Konfigurationseinstellungen der Komponenten direkt im Quellcode mitzuführen. So können zum Beispiel Transaktionsattribute für Session Bean-Methoden oder Datenbankabbildungen der persistenten Objekte direkt an den Variablen vorgenommen werden. Dies reduziert den Entwicklungsaufwand, weil die Erstellung und Pflege von Deployment-Deskriptoren und externen Mapping-Dateien überflüssig wird. Da die Einstellungen im Quelltext

festgeschrieben sind und ohne Neukompilierung nicht geändert werden können, wurden die Deployment-Deskriptoren nicht ganz abgeschafft. Sie ermöglichen vielmehr Änderungen an den im Quelltext hart kodierten Einstellungen vorzunehmen, um beispielsweise eine Anwendung an ihre Umgebung anzupassen

Ein weiterer neuer Ansatz ist das so genannte *Configuration by Exception*. Soweit möglich gibt es für alle Annotations Standardwerte. Das bedeutet zum Beispiel, dass etwa ein fehlendes Transaktionsattribut einer Session Bean-Methode bedeutet, dass die Methode immer innerhalb einer Transaktion ausgeführt wird. Sollte der Aufruf schon innerhalb einer Transaktion stattfinden, wird diese genutzt. Andernfalls wird eine neue Transaktion erstellt.

Dependency Injection ermöglicht es dem Entwickler, größtenteils auf JNDI-Aufrufe und deren Fehlerbehandlung zu verzichten. Über Annotations kann der Entwickler den Container dazu veranlassen, Variablen eines EJB mit Referenzen zum Beispiel auf andere EJBs oder den Entitymanager (s.u.) zu versehen, so dass das Bean die Referenzen direkt nutzen kann, ohne sich die Referenz selbst über JNDI holen zu müssen.

Die vormalige Notwendigkeit, dass alle EJB-Bestandteile von bestimmten Klassen beziehungsweise Interfaces erben müssen, ist mit EJB 3.0 obsolet geworden. Als EJBs werden einfache POJO (Plain Old Java Object) genutzt.

Die Home-Interfaces sind obsolet geworden. Über JNDI erhält man nun immer sofort eine Referenz auf ein EJB zurück und nicht erst eine Referenz auf das Home-Interface, über das man sich die eigentliche Referenz auf das EJB holen kann. Benötigte Lifecycle-Methoden von EJBs werden über Annotations als solche gekennzeichnet. Gleichzeitig dürfen EJBs in der neuen EJB-Spezifikation ihre Komponenten-Interfaces implementieren wie man es von der objektorientierten Entwicklung gewohnt ist.

Das Persistenzmodell wurde komplett neu entwickelt. Geschäftsdaten können nun in bekannter, objektorientierter Weise modelliert werden. Die Klassen werden über Annotations als persistent markiert. Mit weiteren Annotations werden Variablen der Klasse ausgestattet, um etwa Primärschlüssel und Abbildung auf Datenbankfelder zu bestimmen. Der ehemals über Home-Interfaces realisierte Zugriff auf Entity Beans wird nun über eine zentrale Instanz, den Entitymanager, geregelt. Über ihn können Geschäftdaten gesucht, geladen und persistiert werden. Da die persistenten Klassen keine Schnittstellen mehr implementieren müssen und als klassische POJOs außerhalb des Containers verwendet werden können, bezeichnet man sie in EJB 3.0 nur noch als *Entities* statt Entity Beans [DeKe05b].

Mit Interceptors (dt. Abfänger) wurde das Konzept von AOP für EJBs eingeführt. Sie ermöglichen es dem Entwickler, die schon vorhandenen nichtfunktionalen Dienste des Containers um eigene Aspekte zu erweitern. So können EJBs mit einer Annotation für Interceptors versehen werden. Der Methodenaufruf eines mit dieser Annotation ausgestatteten EJB wird dabei innerhalb des

Interceptors eingebettet. Das bedeutet soviel, dass das EJB nicht direkt aufgerufen wird, sondern zuerst der Interceptor. Innerhalb des Interceptor wird dann die eigentliche EJB Methode aufgerufen.

3.5.5 Java Message Service

Der Java Message Service ist nach [HBSFS02] und [Star04] eine Java API, die MOM-Produkte möglichst vieler Hersteller unterstützen soll und 1998 von Sun vorgestellt wurde. Dazu umfasst die API eine Reihe von Schnittstellen, deren Implementierung speziell auf den eingesetzten Message Broker (JMS-Provider genannt) abgestimmt ist. JMS unterstützt zwei Typen von Nachrichtenwarteschlangen (in JMS *Destinations* genannt) (vgl. Abbildung 11):

- **Queues** (dt. Warteschlangen) entsprechen dem Konzept eines Emailpostfachs. Mehrere Sender können gleichzeitig Nachrichten an eine Queue schicken. Eine Nachricht wird aber immer nur maximal einem Empfänger zugestellt.
- **Topics** (dt. Themen) arbeiten nach dem aus Newsgroups bekannten Publish/Subscribe-Konzept. Hier können ebenfalls mehrere Sender Nachrichten veröffentlichen (engl. to publish). Im Gegensatz zur Queue können aber mehrere Empfänger ein und dieselbe Nachricht verarbeiten. Dazu müssen sie das Topic abbonieren (engl. to subscribe) und bekommen dann fortan entsprechende Nachrichten zugestellt.

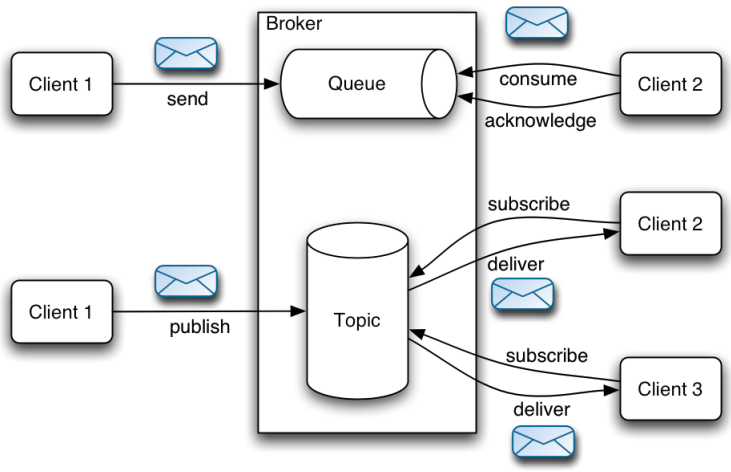

Abb. 11: Kommunikationsmodelle von JMS nach [BaGa02]

Destinations werden von einem Administrator angelegt und sind normalerweise zur Laufzeit nicht modifizierbar. Deswegen werden sie administrierte Objekte genannt. Die administrierten Objekte werden mit JNDI abgelegt, so dass sie für Clients zugänglich sind. Ebenfalls über JNDI können sich Clients mittels einer ConnectionFactory eine Verbindung zum JMS-Provider erstellen lassen.

JMS unterstützt sowohl Polling als auch Pushing von Nachrichten (siehe Kapitel 3.4.3). Beim Pushing muss der Empfänger `javax.jms.MessageListener` implementieren und sich für die automatische Zustellung von Nachrichten an der Destination registrieren.

Im Falle der Nichterreichbarkeit des Empfängers (oder wenn es überhaupt keinen Empfänger für die Queue gibt) speichern Queues die Nachrichten automatisch solange zwischen, bis sich wieder ein Empfänger bei ihnen registriert. Bei Topics sieht das anders aus. Eingehende Nachrichten werden an alle Subscriber weitergeleitet, deren Selektionskriterien auf die Nachricht zutreffen. Die JMS API bietet über so genannte *Message Selectors* einen selektiven Empfang von Nachrichten, wobei als Selektionskriterien alle Nachrichtenattribute in Frage kommen. Sollte kein Empfänger den Topic abboniert haben, würden eine eingehende Nachricht unbearbeitet bleiben. Eine Lösung bieten hier die so genannten Durable Subscriber (dt. dauerhaften Abonnenten). Einmal für ein Topic registriert sichert der JMS-Provider die nachträgliche Weiterleitung von Nachrichten auch dann zu, wenn der Empfänger zeitweise nicht erreichbar ist.

Eine JMS Nachricht besteht immer aus den folgenden drei Bestandteilen:

1. Der **Header** (dt. Nachrichtenkopf) enthält alle notwendigen Attribute einer Nachricht. Dazu zählen etwa Empfänger und Absender, Sendezeitpunkt und Priorität der Nachricht.
2. Die **Properties** (dt. Nachrichteneinstellungen) sind optional und enthalten weiterführende, für den JMS-Provider nicht relevante Informationen, über die ein Empfänger die Nachrichten filtern kann. Als Werte kommen alle primitiven Javatypen und `java.lang.String` in Frage.
3. Der **Body** (dt. Nachrichtenkörper) enthält den eigentlichen Inhalt der Nachricht.

Als Nachrichteninhalte unterstützt JMS fünf verschiedene Typen:

- Eine **StreamMessage** ermöglicht das serielle Übertragen von interpretierten Daten und ist daher mit `java.io.DataOutputStream` vergleichbar
- Eine **MapMessage** enthält Schlüssel/Wert-Paare. Strings bilden dabei die Schlüssel, während als Werte beliebige, primitive Javatypen zum Einsatz kommen können.
- Eine **TextMessage** beinhaltet nur ein `java.lang.String` und bietet sich damit für die Übertragung einfacher Textinhalte an.
- Eine **ObjectMessage** ermöglicht das Senden eines beliebigen, serialisierbaren Objektes (es muss `java.io.Serializable` implementieren).
- Eine **ByteMessage** ermöglicht es letztendlich, eine Folge uninterpretierter Bytes zu senden. Dieser Nachrichtentyp ist sinnvoll, wenn als Kommunikationspartner in anderen Programmiersprachen entwickelte Software eingesetzt wird, welche Daten möglicherweise anders interpretiert.

Queues und Topics sind transaktionale Ressourcen. Das heißt alle Operationen (z.B. das Erstellen oder Verarbeiten von Nachrichten) an ihnen finden innerhalb einer Transaktion statt. Wird eine Transaktion zurückgesetzt, werden in dieser Transaktion erzeugte Nachrichten verworfen und dem Empfänger nicht zugestellt. Wird die Transaktion dagegen bei der Verarbeitung einer eingehenden Nachricht zurückgesetzt, so wird die Zustellung der Nachricht ebenso zurückgesetzt. Nach einer bestimmten Zeitspanne versucht der Broker, dieselbe Nachricht erneut zuzustellen.

3.6 Spring Framework

Spring ist ein von Interface 21 entwickeltes, nicht standardisiertes Framework, das unter der Apache 2.0 Lizenz genutzt werden kann. Es basiert auf den von [John02] beschriebenen Lösungsansätzen der EJB 2.1 Defizite und realisiert eine leichtgewichtige Middleware, die ähnlich wie EJB Container eine Laufzeitzeitumgebung für Komponenten darstellt und innerhalb wie außerhalb von Applikationsservern genutzt werden kann. Da in dieser Arbeit Java EE als Middleware-Plattform genutzt wird und viele der Vorzüge von Spring gegenüber EJB 2.1 auch in EJB 3.0 Anwendung gefunden haben, wird an dieser Stelle nur kurz auf die Unterschiede zur EJB 3.0 Spezifikation eingegangen.

Springs Architektur basiert auf *Dependency Injection*. Im Gegensatz zu EJB 3.0 ermöglicht Spring die Injizierung beliebiger POJOs, die in Spring *Beans* genannt werden. Die Konfiguration der Injizierung wird in einer XML-Datei abgelegt. Denselben Mechanismus nutzt Spring, um den Objekten der Anwendung Middleware-Dienste bereitzustellen – diese werden ebenfalls injiziert. Als Dienste kommen dabei bewährte Software zum Einsatz: Hibernate, JDO und andere O/R Mapper lassen ebenso nutzen wie beispielsweise JAX-RPC als Kommunikationsprotokoll zur Verteilung der Anwendung.

Aufgrund dieser Architekturentscheidung können Dienste flexibel in die Anwendung integriert werden. Während mit den EJB Spezifikation allerdings eine API existiert, die von der Implementierung des Persistenzmechanismus des jeweiligen Herstellers abstrahiert, legt sich der Entwickler bei Spring mit der Nutzung einer bestimmten O/R Mapper-API auf einen Persistenzmechanismus fest.

Die XML-basierte Konfiguration von Spring ist ein mächtiges Werkzeug zur Zusammenstellung von Beans, ihre Komplexität wächst allerdings mit der Größe der Anwendung und erschwert damit ihre Wartbarkeit. Während EJB 3.0 mit *Configuration by Exception* nur die vom Standardverhalten abweichende Konfiguration fordert, muss bei Spring die Konfiguration jedes Beans explizit angegeben werden. Insbesondere deklarative Konfigurationen wie zum Beispiel der Transaktionssteuerung lassen die Nachteile der XML-Konfiguration gegenüber der Java Annotations deutlich werden.

Abschließend lässt sich festhalten, dass Spring aufgrund der flexiblen Integration weiterer Middleware-Dienste eine leichtgewichtige Alternative zu einem

vollständigen Applikationsserver darstellt. Spring Beans lassen sich dadurch beispielsweise einfacher testen als EJBs. Mit Blick auf die langfristige Projektplanung sollte man dessen ungeachtet bedenken, dass der Erfolg des Projekts vom Fortbestand des Spring Frameworks selbst abhängig ist (vgl. [Yuan05]).

3.7 Zusammenfassung

Java EE Applikationsserver bieten eine robuste und skalierbare Laufzeitumgebung für Unternehmenssoftware. Das vom EJB Container umgesetzte Komponentenmodell vereinfacht darüber hinaus die Entwicklung der EJB genannten Softwarekomponenten, da die Komponenten auf viele vom Container angebotenen Dienste (z.B. Persistenz und der verteilte Zugriff auf Komponenten) zurückgreifen können, so dass sich der Entwickler auf die Implementierung der Geschäftslogik konzentrieren kann.

Java EE Applikationsserver vereinen dabei bekannte und bewährte Middlwaretechniken wie beispielsweise Message Broker, ORBs und TP-Monitore. Die dadurch gewonnen Flexibilität ermöglicht den Einsatz von Java EE in einem sehr vielen Kontexten, einerseits weil die EJBs auf viel Funktionalität zurückgreifen können und andererseits, weil sich der Applikationsserver durch die JCA an viele bestehenden Systeme (engl. *Legacy Systems*) anbinden lässt.

Java EE Applikationsserver stellen damit eine ideale Laufzeitumgebung für ebenfalls in Java geschriebene Software wie etwa Jadex dar. Wie sich im nächsten Kapitel zeigen wird, erfüllen EJB Container und damit Applikationsserver alle Vorraussetzungen, um sie als Agentenplattformen einzusetzen.

Kapitel 4

Analyse der Integration

Das vorliegende Kapitel setzt sich mit den möglichen Integrationsarten auseinander und gliedert sich wie folgt: Zuerst werden die technischen Anforderungen von BDI-Agenten-Systemen analysiert, zu denen unter anderem das Jadex-Framework zählt. Auf Grundlage dieser Anforderungen wird untersucht, inwieweit sich Jadex in eine Java EE Umgebung integrieren lässt. Zu diesem Zweck werden existierende Realisierungen solcher Integrationen im Laufe der Untersuchung herangezogen und erläutert. Abschließend werden die Integrationsarten auf ihre Vor- und Nachteile hin bewertet.

4.1 Anforderungen von Agenten

Nach [BrHa02] sind die Unterschiede zwischen Agentenplattformen und Applikationsservern geringer als man erwarten könnte. Beide ermöglichen den Betrieb von verteilten Komponenten (EJBs wie auch Agenten), indem sie Dienste bereitstellen, die etwa das Anbieten und Aufsuchen von Diensten vereinfachen oder die Kommunikation zwischen Komponenten/Agenten sicherstellen. Agentenplattformen fehlen gegenüber den Applikationsservern typische TP-Monitor-Eigenschaften wie Ressourcenverwaltung, Sicherheit und Fehlertoleranz. Davon abgesehen lassen sich EJBs nicht einfach als Agenten bezeichnen, auch wenn ihre Laufzeitumgebung ähnliche Dienste erbringt wie eine Agentenplattform. Ihnen fehlen agententypische Merkmale wie etwa die Fähigkeit, flexibel und selbständig Probleme zu lösen, da es sich bei EJBs in der Regel nur um reaktive Komponenten handelt.

Ausgehend von den drei in Kapitel 2.2 beschriebenen Agenteneigenschaften wird in diesem Kapitel deswegen zusammengefasst, welche technischen Anforderungen ein Agent an seine Laufzeitumgebung stellt. Eine der grundlegenden Eigenschaften von Agenten ist ihre Soziabilität, das heißt ihre Fähigkeit mit anderen Agenten über Nachrichten Informationen auszutauschen. Dieses setzt ein asynchrones Nachrichtentransportsystem zwingend voraus.

Unter der Voraussetzung, dass die Agenten selbständig ihre Ziele verfolgen, muss darüber hinaus noch sichergestellt werden, dass jedem Agenten prinzipiell jederzeit Rechenzeit zur Verfügung steht. In der Regel weisen Plattformen jedem Agenten einen oder mehrere Threads zu, die unter der Kontrolle des Agenten stehen und es ihm ermöglichen, dauerhaft aktiv zu sein. Da Agenten aber nicht permanent aktiv sind, besteht eine Alternative darin, den Agenten bei Bedarf von außen Threads für ihre Arbeit zur Verfügung zu stellen. Hat der Agent vorerst seine Arbeit erledigt und ist momentan inaktiv, kann der Thread für die Ausführung anderer Agenten wieder verwendet werden. Diese Variante setzt einen Scheduler (dt. Zeitplaner) voraus, der die Agenten nach einer von ihnen festgelegten Zeitspanne Rechenzeit zur Verfügung stellt und sie auf diese Weise wieder aufweckt.

Die letzte Anforderung von Agenten stellt eine nicht zu unterschätzende Herausforderung für eine Integration dar: Der Agentenzustand muss irgendwie verwaltet werden. Dies wird insbesondere dann notwendig, wenn Agenten nicht ihre eigenen Threads verwalten, sondern diese nur bei bestimmten Ereignissen von der Plattform zugewiesen bekommen. Die Lösung, alle Agentenzustände dauerhaft im Arbeitspeicher zu halten, würde den Unternehmensanforderungen in Hinsicht auf Skalierbarkeit und Zuverlässigkeit zuwiderlaufen, da einerseits der Arbeitsspeicherbedarf linear mit der Anzahl der Agenten wächst und andererseits Systemfehler zum Verlust der Agentenzustände führen würden. Zumindest vorübergehend inaktive Agenten sollten aus dem Arbeitsspeicher ausgelagert werden können. Optimal wäre eine Lösung, die nur aktuell benötigte Teile des Agentenzustands in den Speicher lädt und nicht benötigte Teile ausgelagert lässt.

Die drei Anforderungen sind nahezu unabhängig voneinander realisierbar. So ist es etwa für den Nachrichtentransport unerheblich, ob die Agenten im Speicher vorliegen oder erst geladen werden oder ob die Threads von den Agenten oder von der Laufzeitumgebung verwaltet werden. Die einzigen Abhängigkeiten ergeben sich durch die Restriktion von Java. So können zum Beispiel sich in Ausführung befindliche Agenten nicht ohne weiteres persistiert werden, da Java keinen Zugriff auf den Stack eines Threads bietet (siehe Kapitel 2.5.2). Aufgrund dieser Unabhängigkeit ist es denkbar, unterschiedliche Ansätze für die jeweiligen Anforderungen von BDI-Agenten zu implementieren, die miteinander kombinierbar sind.

4.2 Integrationsansätze

Nach [BrHa02] werden zwei grundsätzliche Möglichkeiten unterschieden, Agententechnologie in einen Applikationsserver zu integrieren. Beide Ansätze werden in Abbildung 12 im Vergleich dargestellt:

- Eine Variante besteht darin, einen eigenen **Agentencontainer** neben dem Web und dem EJB Container zu betreiben (a). Da die Schnittstellen zwischen Container und Applikationsserver weder standardisiert, noch teilweise offen gelegt sind, wäre diese Lösung proprietär und müsste für andere Applikationsserver, soweit möglich, portiert werden. Da der Container aber von den Spezifika des jeweiligen Applikationsservers abstrahiert, wären die Agenten in dem Container portabel.
- Die Alternative ist eine **komponentenbasierte Integration** mit Hilfe von EJBs (b). Dazu würden die EJBs um Agentenkonzepte erweitert werden, so sie als vollwertige Agenten eingesetzt werden könnten. Da EJBs über standardisierte Schnittstellen verfügen, wäre diese Lösung herstellerunabhängig und könnte auf einem beliebigen Java EE Applikationsserver eingesetzt werden.

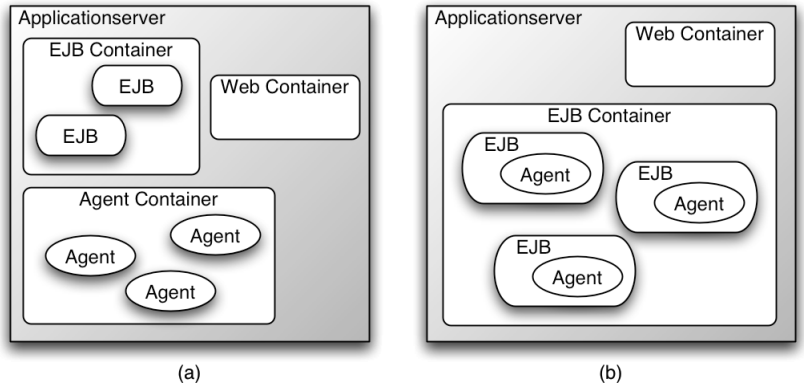

Abb. 12: Container- und komponentenbasierte Integration

In den folgenden beiden Abschnitten wird auf die beiden Ansätze näher eingegangen und anhand von Beispielintegrationen erläutert wie sie technisch rea-

lisiert werden könnten. Im Anschluss an jedes Kapitel findet sich jeweils ein Architekturbeispiel, das veranschaulichen soll wie so eine Integration für Jadex aussehen könnte.

4.3 Containerbasierte Integration

Über die Java EE Spezifikation hinaus bieten viele Applikationsserver eigene, herstellerspezifische Schnittstellen an, über die der Server um weitere Dienste erweitert werden kann. Über diese Schnittstellen werden in der Regel die von der Spezifikation vorgesehen EJB und Web Container an den Server verwaltet. Es wäre also möglich, eine Integration auf dieser Ebene durchzuführen. Neben den beiden schon genannten Containern und möglicherweise anderen serverinternen Diensten würde dann eine Agentenplattform laufen. Da diese Plattform ähnlich wie der EJB Container die Laufzeitumgebung für bestimmte Entitäten darstellt, könnte man in dem Fall von einem vom Applikationsserver verwalteten *Agentencontainer* sprechen, der den Anforderungen von BDI-Agenten gerecht wird.

4.3.1 BlueJADE

Ein Vertreter dieser Art von Integration ist BlueJADE[22] anzuführen. Bei BlueJADE handelt es sich um ein Projekt, welches ursprünglich auf die Integration der Agentenplattform JADE in den Bluestone Applikationsserver von Bluestone Software abzielte. Nach dem Verkauf des Bluestone Applikationsservers an Hewlett-Packard wurde das Projekt für den frei erhältlichen JBoss Applikationsserver angepasst und dahingehend weiter entwickelt. Als die Arbeit an BlueJADE eingestellt wurde, wurde das Projekt als Open Source veröffentlicht (vgl. [NiNy05]).

Der JBoss Applikationsserver[23] wurde von der gleichnamigen Firma JBoss als Open Source Produkt entwickelt und ist Teil der JBoss Enterprise Middleware Suite (JEMS), welche die Entwicklung von Geschäftsanwendungen unterstützen soll. Die Architektur vom JBoss Applikationsserver wurde von Anfang an hochgradig modularisierbar ausgelegt. Dieses wird durch einen leichtgewichtigen Mikrokernel erreicht, der über die Java Management Extensions (JMX) API[24] verwaltet und dessen Funktionalität über zusätzliche, ansteckbare Dienste erweitert werden kann (vgl. [JBoss05]).

Dieser Ansatzpunkt wird von BlueJADE genutzt, um im Kontext des Applikationsserver JADE-Agenten laufen zu lassen. Nach [CGBKR02] bildet BlueJADE dabei den Adapter zwischen JMX-basierter Verwaltungsschnittstelle

22 `http://sourceforge.net/projects/bluejade/`
23 `http://labs.jboss.com/portal/jbossas/`
24 Bei JMX handelt es sich um eine im JCP entstandene Spezifikation zur Verwaltung und Überwachung von Java Anwendungen, Objekten. Dies wird durch die Implementierung einer bestimmten API erreicht.

des Applikationsservers und JADE. Da BlueJADE zwischen beiden vermittelt, gibt es zwei Sichten zu erklären:

- Aus der **Serversicht** bietet BlueJADE die zum Starten und Beenden des Dienstes nötigen Methoden. Darüber hinaus wird gewährleistet, dass von JADE normalerweise auf der Konsole ausgegebene Ereignismeldungen an den Applikationsserver weitergereicht werden, der sie selbst protokolliert.
- Die **Agentensicht** ermöglicht einen JMX-basierten Zugriff auf die Agenten und die Agentenplattform. Über entsprechende Methoden können Agenten wie auch die gesamte Plattform gestartet, gestoppt, pausiert und wieder fortgesetzt werden. Ebenso können der Agenten- beziehungsweise Plattformname sowie der Zustand der beiden abgefragt werden.

Die Serversicht kapselt demzufolge den Zugriff auf den Dienst als solches und ist damit vom Applikationsserverhersteller abhängig. Die Agentensicht ermöglicht die Interaktion mit dem Dienst (respektive Agenten und Plattform) und kapselt deshalb alle dafür nötigen Methoden (vgl. [CoGB02] und [CoGr02]).[25]

BlueJADE delegiert nur die Verwaltung der Agentenplattform an den Applikationsserver und genügt damit nur den Anforderungen an die Administrierbarkeit. Andere der in Kapitel 3.2 genannten Anforderungen werden nicht berücksichtigt. Allerdings gibt es inzwischen eine Erweiterung für JADE, die es erlaubt Agenten, ihre unverarbeiteten Nachrichten sowie Plattformkonfigurationen zu persistieren (vgl. [Rim04]). Mit Hilfe dieser Erweiterung könnte die vorgenannte Integrationslösung zumindest um die Persistierung von Agenten erweitert werden.

4.3.2 Jademx

Jademx[26] gestattet ähnlich wie BlueJADE JMX-basierten Zugriff auf Agenten und Plattform innerhalb eines Java EE Applikationsservers. Darüber hinaus lassen sich mit Jademx auch JADE-Agenten realisieren, bei denen nicht nur der Lebenszyklus verwaltet wird, sondern der vollständige Agent. Entsprechende JADE-Agenten müssen dazu von der Klasse `jade.jademx.agent.JademxAgent` erben, welche wiederum das Interface `javax.management.DynamicMBean` implementiert. Letzteres ermöglich über generalisierte Methoden den lesenden und schreibenden Zugriff auf den Zustand des Agenten. Dadurch bedingt, dass ein Agent durch die Implementierung des Interface ein MBean (Managed Bean) ist, kann er ebenfalls eigene Methoden für den Zugriff über JMX freigeben. Aufgrund des er-

25 Da sich die Dokumentation von BlueJADE zum großen Teil noch auf den Applikationsserver von Hewlett-Packard bezieht, beruht diese Beschreibung von BlueJADE auf einer Analyse des Quelltextes, bei der die genannten Quellen berücksichtigt wurden.

26 `http://jademx.sourceforge.net`

weiterten Zugriffs können somit auch JUnit-Tests[27] für Jademx-Agenten genutzt werden. Bestehende JADE-Agenten lassen sich natürlich immer noch ausführen, allerdings muss dann auf die eben genannte, erweiterte Funktionalität verzichtet werden.

Während BlueJADE nur für den Bluestone, später dann für den JBoss Applikationsserver konzipiert wurde, bietet Jademx zusätzliche Möglichkeiten JADE zu installieren:

- Aufgrund der integrierten JMX Konsole ist eine Installation als JBoss Service und der Zugriff über die Konsole relativ einfach einzusetzen.
- Alternativ lässt sich Jademx als Servlet in einem beliebigen Web Container nutzen. Der Vorteil dieser Lösung besteht in der Portabilität der Integration. Alle Applikationsserver verfügen über einen Web Container, in dem Servlets ausgeführt werden können.
- Innerhalb von J2SE 5.0 lässt sich Jademx ohne Applikationsserver betreiben.

Im Gegensatz zu BlueJADE wird Jademx noch aktiv weiter entwickelt. Mit Blick auf die in Kapitel 3.2 genannten Anforderungen muss aber festgestellt werden, dass Jademx nicht mehr leistet als BlueJADE.

4.3.3 Realisierung für Jadex

Da es sich bei Jadex nur um ein Framework für BDI-Agenten handelt, das auf unterschiedlichen Plattformen eingesetzt werden kann, stellt sich für eine Integration die Frage wie die Anforderungen der BDI-Agenten gelöst werden. Da mit BlueJADE und Jademx schon lauffähige Integrationen von JADE in Applikationsserver existieren, bietet es sich natürlich an, JADE als Laufzeitumgebung für Jadex-Agenten zu nutzen. Allerdings ist auch die Standalone-Plattform als Agentencontainer denkbar, da sie an sich schon modular aufgebaut ist und somit mit relativ geringem Aufwand für den Einsatz im Applikationsserver anpassen lassen müsste.

Abbildung 13 verdeutlicht diesbezüglich am Beispiel von JADE als Plattform für Jadex-Agenten und dem JBoss Applikationsserver, an welcher Stelle Plattform und Agenten in einem Applikationsserver einzuordnen sind: BlueJADE ermöglicht die Installation von JADE als Dienst in dem Server. Statt aber relativ einfach modellierte JADE-Agenten auf der Plattform laufen zu lassen, wird stattdessen das BDI-Agentenframework Jadex eingesetzt, um die Agenten mit einem mächtigeren, kognitiven Modell auszustatten.

27 JUnit ist ein Java Framework zum Testen von Einheiten (engl. Units), welches zumeist Java Klassen sind.

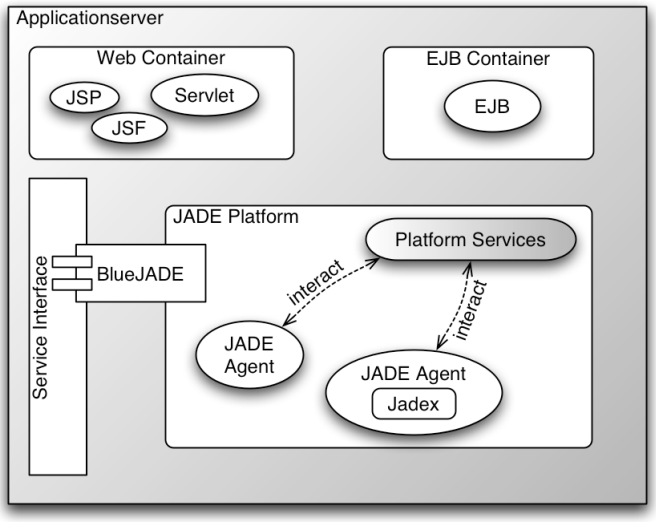

Abb. 13: Beispielarchitektur einer proprietären Integration von Jadex

Der Einsatz einer Agentenplattform bleibt notwendig, da Agenten auf bestimmte Dienste (z.B. Scheduling und Nachrichtentransport) zurückgreifen können müssen. Da die herstellerabhängigen Schnittstellen zur Erweiterung der Applikationsservern hingegen nur die minimale Schnittstelle definieren, um einen Dienst im Verwaltungskontext eines Servers laufen zu lassen, ist jeder Dienst für seine Ressourcenverwaltung (z.B. Threads und Kommunikation) selber verantwortlich.

4.4 Komponentenbasierte Integration

Die Alternative zu einer proprietären Integration besteht darin, die Agenten nicht in einem neuen Container neben Web und EJB Container auszuführen, sondern sie als EJB zu modellieren, so dass sie im EJB Container laufen können. Dadurch können die Agenten über die reine Verwaltung hinaus prinzipiell auf alle vom EJB Container erbrachten Dienste zurückgreifen. Von einer *komponentenbasierten Integration* kann man deshalb sprechen, weil die Agenten aus Sicht des EJB Containers wie andere EJB Komponenten behandelt werden.

4.4.1 Restriktionen für Enterprise JavaBeans

Eine komponentenbasierte Integration ist im Vergleich zu einem Agentencontainer ungleich komplexer. Neben der Erstellung aller zur Installation der Komponenten notwendigen Artefakte wie Deployment-Deskriptoren und Interfaces verlangt die EJB Spezifikation vom Entwickler die Einhaltung bestimmter Programmierrestriktionen. Durch die Einhaltung dieser Restriktionen soll gewährleistet werden, dass die Komponenten portabel sind und in jedem EJB Container installiert werden können. Außerdem wird mit den

Restriktionen das Sicherheitskonzept des Applikationsservers durchgesetzt. Nachfolgend werden deswegen alle für eine Integration relevanten Einschränkungen erklärt (vgl. [DeKe05c]):

- EJBs sollten keine statischen Felder beinhalten, die veränderbar sind. Statische Konstanten sind dagegen unproblematisch. Damit wird die verteilte Ausführung von Komponenten in einem Cluster sichergestellt. Da statische Variablen nur innerhalb einer JVM gelten, würden die Rechner eines Clusters auf verschiedene, statische Variablen zugreifen.
- EJBs sollten nicht selbständig ihren Zugriff synchronisieren. Diese Aufgabe übernimmt der Container, da die Synchronisierungsprimitiva nur innerhalb einer JVM wirken und die Synchronisierung in einem Cluster damit nicht möglich wäre.
- EJBs dürfen keine GUI-Elemente erzeugen. Da Applikationsserver in der Regel auf Rechnern ohne angeschlossene Peripherie wie Tastatur und Monitor laufen, wäre eine Benutzeroberfläche ohnehin nicht nutzbar.
- Der Zugriff auf das `java.io` Package ist nicht erlaubt, da das Dateisystem nicht den Ansprüchen von verteilten, konsistenten Geschäftsanwendungen gerecht wird. Geschäftsdaten sollten über die JCA (meistens JDBC) verwaltet werden.
- EJB dürfen keine Server Sockets erstellen, um Dienste im Netzwerk bereitzustellen. Die Öffnung einer Verbindung zu einem beliebigen Rechner ist dagegen legitim. Die Einschränkung ist damit begründet, dass EJBs nicht dauerhaft ausgeführt werden und ihre Lifecycle vom Container verwaltet wird. Um beispielsweise Altanwendungen Zugriff auf EJBs zu ermöglichen, sieht die JCA eine Möglichkeit für eingehende Verbindungen vor. Damit könnte ein Applikationsserver über die ihm bekannten Protokolle hinaus noch weitere Protokolle implementieren.
- EJBs dürfen die Reflection API nur nutzen, um auf Daten zuzugreifen, die entsprechend gekennzeichnet sind (z.B. öffentliche Variablen). Eine anderweitige Nutzung der Reflection API würde die Sicherheit des Applikationsserver unterminieren.
- Die Nutzung eigener Classloader sowie die Modifikationen der JVM etwa durch `System.exit(int)` ist innerhalb von EJBs untersagt.
- EJBs dürfen keine Threads manipulieren. Threads werden vom Applikationsserver in Pools verwaltet und den EJBs bei Bedarf zur Verfügung gestellt.

4.4.2 Living Systems Technology Suite

Living Systems Technology Suite (LS/TS) ist ein Java basiertes Produkt von Whitestein Technologies und vereinfacht den Entwicklungsprozess von Agentenanwendungen. Abbildung 14 bietet eine Übersicht über die Architektur von LS/TS. Demnach teilt sich die Architektur in zwei separate Teile:

Die *Developement Suite* umfasst eine Sammlung von Werkzeugen, die Modellierung, Entwicklung und Testen von Agenten erleichtern. Ebenso gibt es

Werkzeuge für die Installation und die Überwachung von Agenten. Mit Hilfe einer eigens entwickelten an UML (Unified Modeling Language) angelehnten *Agent Modeling Language* und einem *Agent-oriented Developement Methodology* genannten Entwicklungsprozess soll die Entwicklung vereinheitlicht werden (vgl. [WITG04]). Das Installationswerkzeug erstellt aus dem Agentenmodell die notwendigen, von der Runtime Suite abhängigen Laufzeitartefakte.

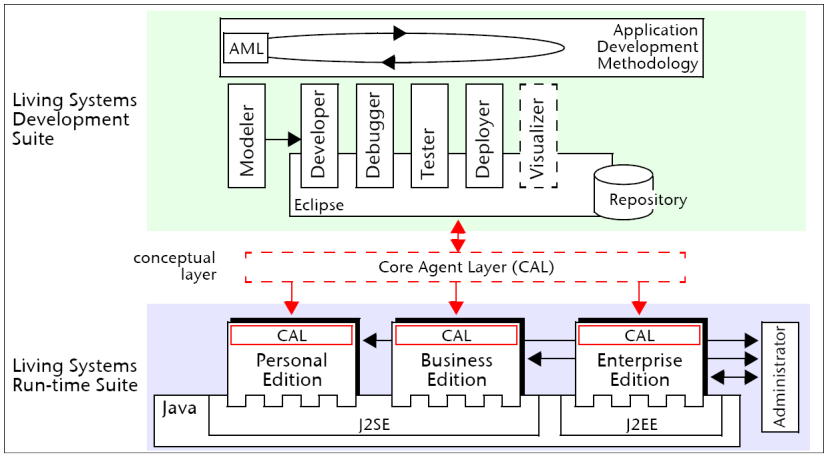

Abb. 14: Architektur von LS/TS nach [RiCK05]

Die *Runtime Suite* stellt dagegen die Laufzeitumgebung der Agenten dar. Diese gibt es in drei Ausführungen. Eine ermöglicht die Evaluation des Produkts (Personal Edition). Die beiden anderen sind kommerziell erhältliche und bieten abgestuft verschieden viele Features. Die Enterprise Edition ermöglicht den Einsatz von LS/TS in J2EE Applikationsservern.

Beide Teile werden durch den so genannten *Core Agent Layer (CAL)* zusammengehalten. Dieser abstrahiert von der zugrunde liegenden Runtime Suite und ermöglicht damit, mit der Developement Suite entwickelte Agenten auf allen drei Editionen ohne Modifikation ausführen zu können. Dank offener Schnittstellen ist der CAL erweiterbar. In der aktuellen Fassung sieht der CAL drei grundlegende Typen von Agenten vor (vgl. [RiCK05] und [WITG05a]):

1. Der **Autonomous Agent** (dt. autonome Agent) kommuniziert asynchron mit anderen Agenten seines Typs. Als einziger Agententyp kann er proaktiv handeln und damit prinzipiell jederzeit aktiv werden. Darüber hinaus verfügt er über einen persistenten Zustand und handelt zielorientiert (vgl. [WITG04]).
2. Der **Servant Agent** (dt. Dienstagent) ist reaktiv und kann nur synchron von anderen Agenten aufgerufen werden. Typischerweise stellt er Dienste bereit, die von mehreren anderen Agenten benötigt werden. Deshalb ist er häufig zustandslos.
3. Der **Data Agent** (dt. Datenagent) ist ebenfalls reaktiv und kann nur synchron angesprochen werden. Er erlaubt den Zugriff auf persistente Daten, die von mehreren anderen Agenten gemeinsam genutzt werden.

Mit dem CAL existiert eine ähnliche Trennung von Agentenmodell und Laufzeitumgebung wie es das Adapterkonzept von Jadex vorsieht (siehe Kapitel 2.5.4). Beide ermöglichen den Einsatz der Agenten auf verschiedener Middleware, indem sie von der Realisierung der von den Agenten benötigten Dienste abstrahieren. Jadex sieht allerdings keine Unterscheidung von Agententypen vor. Von mehreren Jadex-Agenten benötigte Dienste können in einer Capability gekapselt werden, die im jeweiligen ADF eingebunden wird.

Zentrale Datenhaltung, wie sie der Data Agent von LS/TS realisiert, widerspricht eigentlich dem Agentenparadigma. Sie kann aber in Szenarien sinnvoll sein, in denen Agenten in einer Umgebung operieren, die einen von allen Agenten erfassbaren Zustand hat. In Jadex kann man zentrale Daten entweder mit Hilfe von statischen Objekten realisieren oder einen speziellen Agenten modellieren, der den Zustand der Umwelt kapselt.

In Ermangelung einer Enterprise Edition Lizenz von LS/TS konnte die Analyse der Integration leider nur auf Grundlage der Dokumentation durchgeführt werden. Obwohl die Dokumentation von LS/TS fast keine Aussagen darüber macht wie die drei Agententypen in der Enterprise Edition realisiert werden, legt die Dokumentation (siehe [WITG05b]) doch eine Zuordnung gemäß den EJB-Typen (siehe Kapitel 3.5.3) nahe. Dienstagenten würden daher mit Hilfe von Session Beans realisiert werden. Beide sind reaktiv, werden synchron aufgerufen und stellen bestimmte Dienste bereit. Datenagenten scheinen auf Entity Beans abgebildet zu werden. Beide sind ebenfalls reaktiv, werden synchron angesprochen und ermöglichen den Zugriff auf persistente Daten. Die Abbildung von autonomen Agenten auf Message-driven Beans ist dagegen schwerer zu realisieren. Zwar können Message-driven Beans asynchron über JMS kommunizieren, sie sind aber gemäß der Spezifikation zustandslos und reaktiv. Mit dem Einsatz von Message-driven Beans ist von den drei grundlegenden Anforderungen (siehe Kapitel 4.1) also nur die Kommunikation gelöst. Da die Dokumentation keine weiteren Schlüsse über die Integration zulässt, wird eine Lösung dieses Problems nicht in diesem, sondern in Kapitel 4.4.4 beschrieben.

4.4.3 Adaptive Enterprise Solution Suite

Bei der Adaptive Enterprise Solution Suite von Agentis International handelt es sich um ein weiteres kommerzielles Produkt, das BDI-Agenten in Applikationsservern realisiert (vgl. [AgIn03b]). Ähnlich der LS/TS stellt Agentis[28] alle für den Entwicklungsprozess benötigten Werkzeuge zur Verfügung: Mit dem *AdaptiveEnterprise Architect* kann man Modelle der Agenten entwerfen. Diese Modelle werden danach vom *AdaptiveEnterprise Builder* in lauffähige EJB Komponenten übersetzt. Als Laufzeitumgebung realisiert der so genannte *AdaptiveEnterprise Server* die nötige Vermittlung zwischen Agenten und

28 Im Folgenden wird der Kürze halber *Agentis* und die *Adaptive Enterprise Solution Suite* synonym verwendet.

Applikationsserver. Letztlich ermöglicht der *AdaptiveEnterprise Analyzer* die Überwachung der laufenden Agenten (vgl. [AgIn03a] und [AgIn05]).

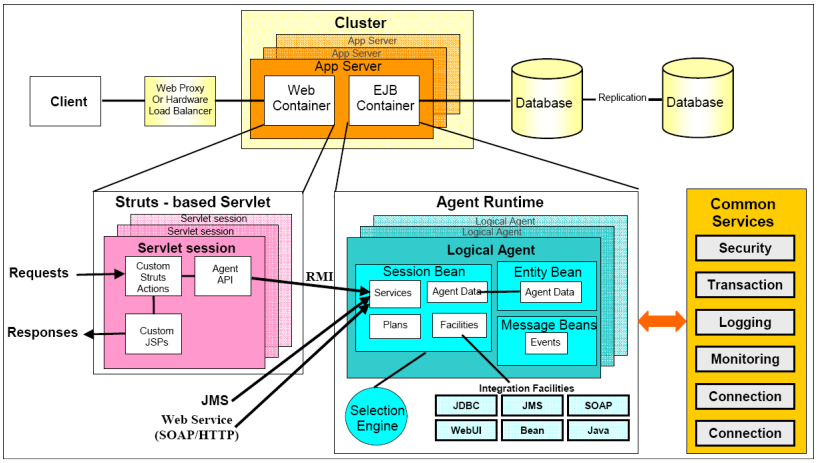

Abb. 15: Architektur der Adaptive Enterprise Solution Suite nach [AgIn05]

Abbildung 15 bietet einen Überblick über die Architektur von Agentis. Im Gegensatz zu LS/TS ermöglicht die Dokumentation der Agentis Architektur allerdings einen tieferen Einblick in die Realisierung der Integration. Nach der Dokumentation wird die Laufzeitumgebung zusammen mit den Agenten wie schon bei LS/TS in Form von unterschiedlichen EJBs in einem Applikationsserver installiert. Nachfolgend wird die Nutzung der verschiedenen EJB-Typen erklärt:

- Agentis verfolgt zwei verschiedene Ansätze, um den Agentenzustand zu verwalten: Für maximale Leistung werden zustandsbehaftete Session Beans eingesetzt. In der Regel befindet sich der Zustand der Beans im Arbeitsspeicher, deswegen ist der Zugriff auf den Agenten besonders schnell. Allerdings ist der Zustand nicht dauerhaft gespeichert, so dass er bei einem Systemfehlers verloren gehen würde. Die Alternative dazu besteht darin, den Agentenzustand in Entity Beans zu verwalten, auf die über ein zustandsloses Session Bean zugegriffen wird.
- Agentenpläne werden durch Session Beans realisiert. So genannte *Facilities* (dt. Möglichkeiten, Gelegenheiten) bieten den Agenten die Möglichkeit, auf vorgefertigte Dienste vom Applikationsserver zurückzugreifen.
- Ebenfalls als Session Beans werden die vom Agenten angebotenen Dienste realisiert. Eine besondere API erleichtert den Zugriff auf diese Dienste von konventionellen Komponenten wie etwa anderen EJBs oder einer Weboberfläche.
- Message-driven Beans ermöglichen letztlich Agenten asynchron anzusprechen. Eingehende Nachrichten erzeugen dabei ein Ereignis, das von dem entsprechenden Agenten verarbeitet wird.

Agentis macht somit konkrete Aussagen darüber, wie Agentenzustände verwaltet werden können. Offen bleibt die Frage, wie die Proaktivität der Agenten realisiert wird. Da die Agenten gemäß den EJB Restriktionen (siehe Kapitel 4.4.1) nicht ihre eigenen Threads verwalten dürfen, muss eine externe Instanz die Agenten auf deren Verlangen hin wieder aktivieren. Da der Zeitpunkt der Aktivierung zwar zeitlich von den Agenten vorgegeben wird, die Instanz aber die Ausführung der Agenten steuert, handelt es sich bei ihr um einen Scheduler. Im folgenden Abschnitt wird erläutert wie ein solcher Scheduler für EJBs realisiert werden kann.

4.4.4 Realisierung für Jadex

Wie sich in den Analysen der vorangegangenen Kapitel zeigt, ist der EJB Container als Plattform für proaktive Agenten durchaus nutzbar. Nachfolgend wird deshalb als erstes allgemein aufgezeigt wie ein EJB Container die in Kapitel 4.1 angeführten Anforderungen von BDI-Agenten theoretisch realisieren könnte. Anschließend wird der Bezug zu Jadex hergestellt, indem erklärt wird wie Jadex anhand dieser Realisierungsmöglichkeiten komponentenbasiert integriert werden könnte.

Kommunikation

Sowohl LS/TS (in der Enterprise Edition) als auch Agentis setzen auf JMS als Nachrichtentransportsystem. Da JMS über den reinen Nachrichtentransport hinaus mit persistenten Warteschlangen und der verzögerter Nachrichtenzustellung noch einige unternehmensrelevante Eigenschaften zusichert, scheint diese Wahl naheliegend. Bei eingehenden Nachrichten werden automatisch festgelegte Message-driven Beans benachrichtigt, die die Nachricht weiter bearbeiten. Zu unterscheiden sind dabei drei verschiedene Ansätze des Nachrichtentransports, die in Abbildung 16 dargestellt werden:

1. Der erste Ansatz sieht vor, dass jedem Agent eine Nachrichtenwarteschlange zusammen mit einem Message-driven Bean zugeordnet wird, das in dieser Queue beziehungsweise in diesem Topic eingehende Nachrichten an genau diesen einen Agenten weiterleitet.
2. Alternativ dazu könnte man nur einen Topic einsetzen, über den alle zwischen den Agenten ausgetauschten Nachrichten transportiert würden. Allerdings müsste dann anhand der JMS-Nachrichten erkennbar sein, welchem Agenten die jeweilige Nachricht zuzuordnen ist. Weiterhin existiert für jeden Agenten wiederum ein Message-driven Bean, welches das Topic abonniert hat und selektiv, diesen einen Agenten betreffende Nachrichten aus dem Topic filtert und ihm zustellt.
3. Der letzte Ansatz geht ebenfalls von einem Topic oder einer Queue aus. Allerdings nimmt bei diesem Ansatz ein einziges Message-driven Bean alle Nachrichten entgegen und entscheidet selbständig anhand der Nachricht, wem diese zuzustellen ist. Aufgrund seiner Funktion kann man dieses eine Bean auch als *Message Dispatcher* (dt. Nachrichtenverteiler) bezeichnen.

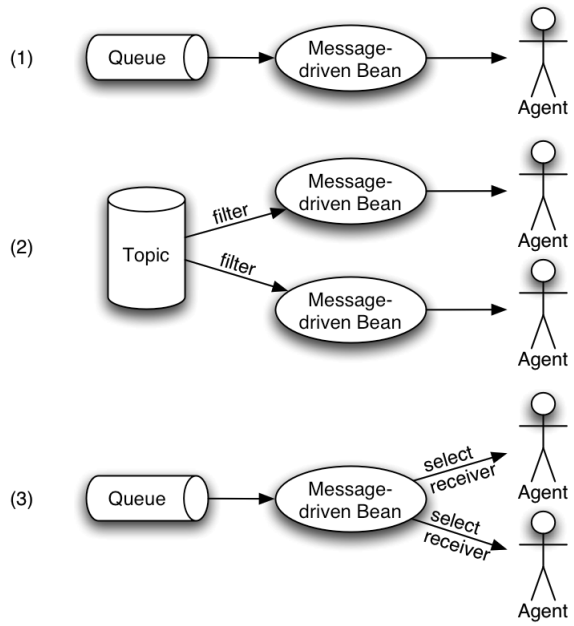

Abb. 16: Realisierung des Nachrichtentransports

Agentenzustand

Da zustandslose Beans nicht genutzt werden können, um den Agentenzustand zu verwalten, können sie die Agentennachrichten nur vom Message Broker entgegen nehmen und sie an die eigentliche den Agentenzustand speichernde Instanz weiterreichen. Bei der Untersuchung von Agentis haben sich zwei gute Ansätze finden lassen, die in Abbildung 17 dargestellt werden:

1. In der einfacheren Variante wird ein zustandsbehaftetes Session Bean für die Speicherung des Agentenzustands eingesetzt. Da der Applikationsserver diese Art von Beans unter hoher Last auslagern kann, gilt als einzige Anforderung an den Agentenzustand, dass er serialisierbar sein muss. Ausgelagert wird üblicherweise auf die Festplatte, da Session Beans keine persistenten Objekte sind. Die Referenz auf das Session Bean muss von den Dispatchern erreichbar abgelegt werden, damit diese äußere Ereignisse an den Agenten weiterleiten können. Dieses könnte zum Beispiel über JNDI erreicht werden, indem die Referenz über den Agentennamen im Verzeichnis abgelegt wird.
2. Die schwerer zu realisierende Variante sieht eine Abbildung des Agentenzustands auf die Datenbank vor. Dafür müsste allerdings der komplette, den Agentenzustand ausmachende Objektgraph auf ein Datenbankschema abgebildet werden. Der Zugriff auf diesen Zustand wäre dann über einen eindeutigen Schlüssel wie etwa dem Agentennamen möglich.

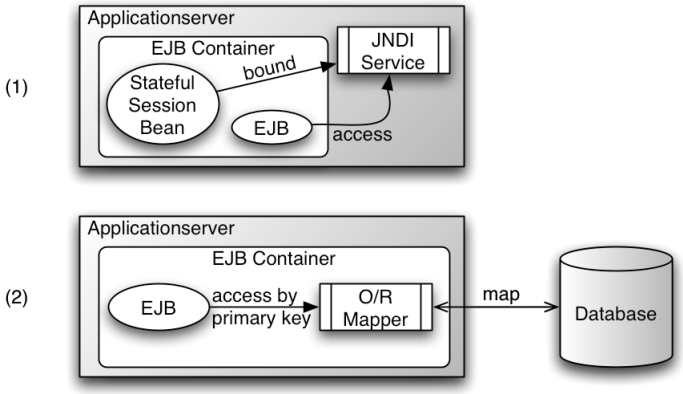

Abb. 17: Realisierung der Agentzustandsspeicherung

In der letzten Variante wurde absichtlich keine Aussagen über die Abbildung des Agentenzustands gemacht, da es unerheblich ist, welches objekt-relationale Mapping-Framework dafür eingesetzt wird. Eine Abbildung mit Hilfe von EJB 2.1 Entity Beans oder EJB 3.0 Entities ist ebenso denkbar wie der Einsatz eines externen Mapping-Frameworks wie zum Beispiel Hibernate.[29]

Scheduling

Für die Realisierung der letzten Anforderung proaktiver BDI-Agenten ließen sich bei den beiden untersuchten Produkten keine Informationen entnehmen. Bei der Beschreibung der Anforderungen (siehe Kapitel 4.1) wurden kurz zwei Möglichkeiten aufgezeigt wie Ausführung von Agenten zeitlich geplant werden kann. Gemäß den in Kapitel 4.4.1 aufgezählten Restriktionen für EJB scheidet eine eigenverantwortliche, interne Threadverwaltung für Agenten aus. Als einzige Möglichkeit verbleibt also ein externer Scheduler, um die Ausführung der Agenten zeitlich zu steuern.

Allerdings ist diese Anforderung im Bereich von Geschäftsanwendungen nicht neues. Auch aktuelle Anwendungen müssen einmalige oder regelmäßige zeitlich gesteuerte Arbeiten verrichten, um beispielsweise die monatlichen Kundenrechnungen zu erstellen. Vor der EJB 2.1 Spezifikation wurde dieses Problem mit Hilfe von externen Anwendungen gelöst, die zu vordefinierten Zeitpunkten bestimmte Methoden an den EJBs aufruft. Auch wenn diese Lösung funktionieren mag, bringt sie doch einige Nachteile mit sich:

- Die Lösung ist proprietär, da die Anwendung nicht unbedingt mehr auf unterschiedlichen Applikationsservern installiert werden kann.
- Der Scheduler wird nicht vom Applikationsserver verwaltet. Die Wiederherstellung eines konsistenten Zustands nach einem Systemfehler ist somit nicht notwendigerweise möglich.
- Die Erstellung neuer und das Löschen bestehender zeitlich gesteuerter Abläufe ist schwierig zu realisieren und kann zu inkonsistenten System-

29 `http://www.hibernate.org`

zuständen führen, da entsprechende Operationen nicht transaktional ausgeführt werden.

Da das Scheduling von EJBs ein signifikantes Problem darstellte, wurde es in der EJB 2.1 Spezifikation (siehe [DeKe05c]) mit der Einführung eines neuen Diensts behoben. Der so genannte *TimerService* (dt. Zeitgeberdienst) erlaubt es zustandslosen EJBs sich über eine API für zeitlich gesteuerte Ereignisse (*Timer* genannt) zu registrieren beziehungsweise diese Registrierung wieder zu löschen. Als Timer kommen dabei sowohl periodische als auch einmalige Ereignisse in Frage, die zum Zeitpunkt ihrer Auslösung eine in der Spezifikation vorgegebene EJB-Methode aufrufen. Der TimerService selbst ist eine transaktionale Ressource. Das bedeutet alle Operation laufen innerhalb der Transaktion ab, in der das EJB ausgeführt wird. Wird eine Transaktion zurückgesetzt, so werden möglicherweise erstellte Timer ebenfalls annulliert beziehungsweise gelöschte Timer wiederhergestellt. Die Spezifikation sichert darüber hinaus zu, dass bei einem Wiederanlauf des Systems alle während der Unterbrechung ausgelösten Timer ausgeführt werden.

Mit Blick auf die Anforderungen BDI-Agenten lässt sich festhalten, dass sich ein proaktives Verhalten mit dem TimerService ohne weiteres realisieren lässt. Falls ein Agent in naher Zukunft wieder handeln möchte, müsste er nur am Ende seiner aktuellen Handlung einen Timer registrieren. Da dies nach der Spezifikation allerdings nur zustandslose EJBs dürfen, muss ein zustandsloses Session Bean die Aufgabe der Registrierung übernehmen. Da dieses Bean bei allen ausgelösten Timern jedes Agenten aktiviert wird und dann entscheiden muss, welchem Agent der aktuell ausgelöste Timer gilt, kann man dieses Bean als *Timer Dispatcher* (dt. Rechenzeitverteiler) bezeichnen.

Bezug zu Jadex

Die eben angeführten Realisierungsmöglichkeiten sind ausreichend, um auf ihrer Basis proaktive, soziale Agenten ohne ein kognitives Modell zu verwirklichen. Da sich die Anforderungen von BDI-Agenten nahezu vollständig in den in Kapitel 2.5.4 beschriebenen Schnittestellen wieder finden lassen (z.B. `IJadexAgent.messageArrived()` und `IAgentAdapter.notifyIn()`), besteht eine komponentenbasierte Integration von Jadex aus der Realisierung der Basisdienste (Timer Dispatcher und eventuell Message Dispatcher) und der optionalen Abbildung des Agentenzustands auf ein Datenbankschema.

Wie sich die beschriebenen Komponenten zu einer Integration komponieren lassen, wird in Kapitel 5.2 erläutert.

4.5 Zusammenfassung

Nach der Feststellung der Anforderungen von BDI-Agenten an ihre Plattform (Nachrichtentransport, Scheduling und Agentenzustandsverwaltung) und der anschließenden Untersuchung wie sich diese Anforderungen im Kontext eines Java EE Applikationsservers umsetzen lassen, hat sich gezeigt, dass es zwei

grundlegende Integrationsarten gibt: Entweder integriert man eine bestehende Plattform über eine proprietäre Verwaltungsschnittstelle in den Applikationsserver oder man bildet das Agentenkonzept und die von der Plattform erbrachten Dienste auf das EJB Komponentenmodell von Java EE ab. Die zweite Variante ist ungleich komplexer zu realisieren, da der Programmierung von EJBs bestimmte Restriktionen auferlegt sind, um die Sicherheit des Servers und die Portabilität der Anwendung zu gewährleisten.

	Komponente	Art	Administrierbarkeit	Persistenter Agentenzustand	Feingranularer Zugriff auf Agentenzustand	Dynamische Plattform	Portabilität	Transaktionales Messaging	Transaktionales Scheduling
Containerbasierte Integration	Art der Anbindung	JMX	+			+	-	-	-
		Servlet	-			+	+	-	-
	Agentenzustand	Arbeitsspeicher		-		+		-	-
		Persistenz via Addon		+		+		-	-
Komponentenbasierte Integration	Kommunikation	Queue und MDB je Agent	-			-	+	+	
		Gemeinsamer Topic; MDB je Agent	-			-	+	+	
		Gemeinsame Queue und MDB	-			+	+	+	
	Agentenzustand	Session Bean	-	-					
		Serialierung	-	+	-				
		O/R Mapper	-	+	+				
	Scheduling	Externer Scheduler	-				-		-
		TimerService	-				+		+

Abb. 18: Vor- und Nachteile der Integrationsarten

Abbildung 18 bietet eine Übersicht über die Vor- und Nachteile der aufgeführten Integrationstechniken. Da sich eine Integration aus verschiedenen Techniken zusammensetzen lässt werden die Techniken in Kategorien aufgeteilt dargestellt. Fehlt die Beurteilung für eine Kategorie (in Form eines *Plus* oder *Minus*), so hat die Wahl der Technik keinen Einfluss auf die dazugehörige Eigenschaft.

Es lässt sich festhalten, dass eine proprietäre Integration nur das Management der Agenten und ihrer Plattform unterstützt, auch wenn Erweiterungen wie persistente Agentenzustände realisierbar sind. Eine komponentenbasierte Integration bietet dagegen alle Vorteile, die der EJB Container den Komponenten als nutzbarer Dienst zur Verfügung stellt. Die Verwaltung der Agenten muss dagegen über eine eigene Schnittstelle erfolgen, da die herstellerabhängigen Verwaltungsschnittstellen innerhalb des EJB Containers keine Anwendung finden können. Die Eigenschaften der Techniken wurden bis auf *dynamische Plattform* und *feingranularer Zugriff auf den Agentenzustand* schon erklärt. Auf die beiden Eigenschaften wird in Kapitel 5.1 und Kapitel 6.2 detaillierter eingegangen. Kurz gesagt bezeichnet *dynamische Plattform* die Eigenschaft der dynamischen Erstellung weiterer Agenten auf der Plattform, während der *feingranulare Zugriff auf den Agentenzustand* eine Folge der Serialisierung des Agentenzustands ist.

Kapitel 5

Implementierung eines Integrationsansatzes

Nach der Analyse der Integrationsmöglichkeiten wird in diesem Kapitel die Implementierung eines der genannten Integrationsansätze erläutert. Dazu wird zuerst die Auswahl des gewählten Ansatzes und der für die Integration verwendeten Anwendungen begründet. Anschließend werden die Architektur der Integration sowie die implementierten Komponenten und ihre Notwendigkeit für Jadex erläutert. Bei der Implementierung aufgetretene Probleme werden ebenso aufgeführt wie ihre Lösung erklärt.

Abschließend wird die Implementierung anhand einer Beispielanwendung evaluiert. Nach der Beschreibung des dafür gewählten Szenarios wird näher auf die technische Umsetzung sowie die benutzten Algorithmen eingegangen.

5.1 Auswahl und Begründung

Bei der Auswahl eines Integrationsansatzes für eine Implementierung hat sich der Autor an den in Kapitel 4.5 genannten Vor- und Nachteilen der Ansätze orientiert. Die Auswahlkriterien setzen sich dabei wie folgt zusammen:

- Ziel der Implementierung soll es sein, möglichst viele der in Kapitel 3.2 genannten Anforderungen von Geschäftsanwendungen zu erfüllen.
- Zur Realisierung dieser nichtfunktionalen Anforderungen soll möglichst auf die von Applikationsservern angebotenen Dienste zurückgegriffen werden, um den Entwicklungsaufwand so gering wie möglich zu halten.
- Die Integration soll so implementiert werden, dass bestehende Jadex-Agenten (mit Einschränkung, siehe Kapitel 5.6) ohne weitere Anpassungen auf dem Applikationsserver laufen können.
- Letztlich hat sich der Autor das Ziel gesetzt, möglichst wenig Quellcode des Jadex Frameworks zu modifizieren, damit sich neue Jadex Versionen ohne größere Anpassungen ebenfalls im Applikationsserver nutzen lassen.

Aufgrund dieser Kriterien fiel die Wahl auf eine komponentenbasierte Integration, da das Komponentenmodell der EJB-Spezifikation den Anforderungen von Geschäftsanwendungen genügt. Darüber hinaus sind EJBs inhärent verteilte Komponenten, so dass der Bezug zu einem verteilten Multi-Agenten System (MAS)[30] nahe liegt.

Bei der Wahl des Applikationsservers hat sich der Autor entschieden, den aktuell in Entwicklung befindlichen JBoss Applikationsserver zu nutzen. Dieser quelloffene Server hat einerseits den Vorteil, dass er die neue EJB 3.0 Spezifikation umsetzt, obwohl diese noch im Entwurf und damit als noch nicht ganz vollständig zu betrachten ist. Die gegenwärtig genutzten EJB 2.1 erhöhen den Entwicklungsaufwand deutlich, da selbst kleine Projekte alle in Kapitel 3.5.3 beschriebenen Artefakte erfordern. Die neue Spezifikation verspricht einen verringerten Entwicklungsaufwand, indem sie die Anzahl der notwendigen Artefakte drastisch reduziert. EJB 2.1 Entity Beans haben im Gegensatz zu EJB 3.0 Entities darüber hinaus den Nachteil, dass die Abbildung komplexer Strukturen, wie etwa Vererbung, nur unzureichend unterstützt wird. Aufgrund des Fehlens eines integralen, transaktionsorientierten Schedulers schied Spring als Middleware-Plattform aus.

Andererseits enthält der JBoss Applikationsserver in seiner Standardinstallation eine ebenfalls in Java geschriebene Datenbank namens Hypersonic[31], die innerhalb des Applikationsservers läuft und für die Persistierung von Entities vorkonfiguriert ist. Da Hypersonic zum Testen der Beispielanwendung ausreichend ist, wurde auf die Anbindung einer externen Datenbank verzichtet.

---

30 Bei Multi-Agenten Systemen handelt es sich um Systeme, bei denen unterschiedlich spezialisierte Agenten gemeinsam ein Problem lösen.

31 `http://www.hsqldb.org`

Für den Fall, dass Modifikationen des Jadex Quellcodes notwendig werden sollten, wurde für diese Arbeit ein eigener Entwicklungszweig angelegt. Dieser Zweig beruht auf der aktuell stabilen Jadex Version 0.941.

5.2 Architektur der Integration

Da eine komponentenbasierte Integration viele Variationen anbietet, wurde die Wahl auf eine Architektur eingeschränkt, die in Abbildung 19 dargestellt wird. Der Zustand der Jadex-Agenten wird dabei mit Hilfe eines O/R Mappers auf ein Datenbankschema abgebildet. Der Mapper lädt den Agentenzustand auf Anfrage und sichert den möglicherweise modifizierten Zustand nachdem die Ausführung des Agenten vorerst abgeschlossen ist.

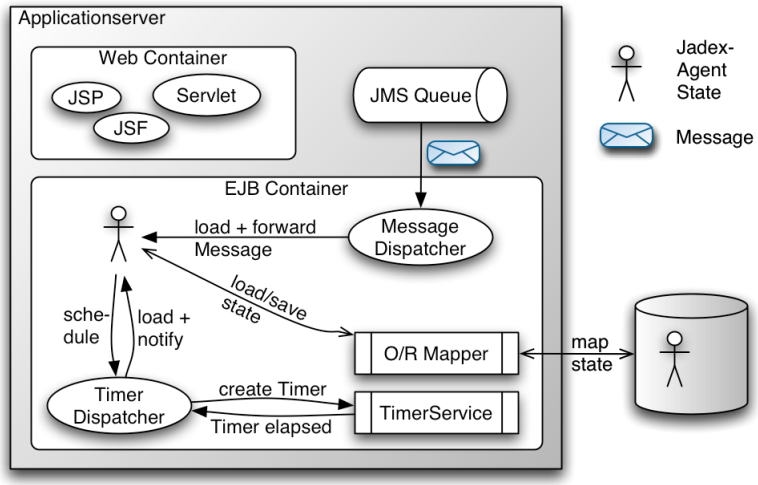

Abb. 19: Architektur der gewählten komponentenbasierten Integration von Jadex

Die Kommunikation mit anderen Agenten geschieht über eine JMS Queue, deren Nachrichten von einem Message Dispatcher entgegengenommen werden. Der Message Dispatcher ermittelt abhängig vom Empfänger jeder Nachricht den zuständigen Agenten, lädt dessen Zustand aus der Datenbank und übergibt ihm die Nachricht zur weiteren Verarbeitung.

Damit Agenten proaktiv handeln können ermöglicht ein weiterer Dispatcher (Timer Dispatcher) den Agenten, sie auf ihre Anfrage hin zu einem beliebigen, zukünftigen Zeitpunkt wieder auszuführen. Der Timer Dispatcher leitet diese Anfragen an den TimerService weiter und vermerkt dabei, vom wem die Anfrage stammt. Ist der festgelegte Zeitpunkt erreicht, wird der Timer ausgelöst und der Dispatcher vom TimerService aufgerufen. Der Dispatcher ermittelt daraufhin den für den Timer vermerkten Agenten, lädt und benachrichtigt ihn anschließend über den ausgelösten Timer.

Die Entscheidung, nur eine JMS Queue als Nachrichtenbus für alle Agenten zu verwenden, begründet sich damit, dass es sich bei JMS Destinations um

administrierte Objekte handelt. Diese werden normalerweise von einem Administrator bei der Installation einer Anwendung eingerichtet und können zur Laufzeit nicht mehr modifiziert werden. Da sich aber auf einer Agentenplattform die Anzahl der Agenten durchaus verändern kann (durch mobile Agenten oder die Architektur der Anwendung bedingt; siehe Kapitel 5.5), könnte für neue Agenten keine Destination erstellt werden. Sie könnten also keine Nachrichten empfangen. Die Wahl eines Message-driven Bean und/oder einer Destination pro Agent ist also nur in statischen Szenarien sinnvoll, in denen permanent dieselben Agenten innerhalb einer Plattform laufen.

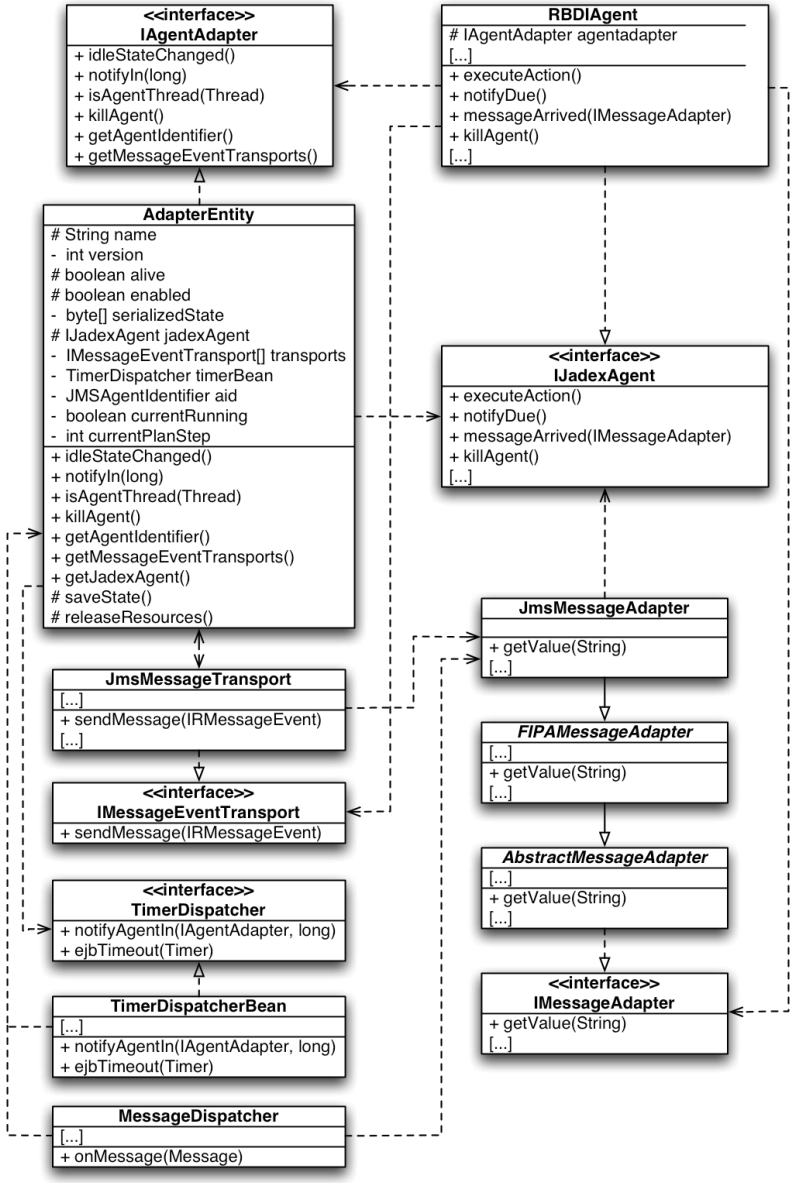

Abb. 20: UML-Klassendiagramm der Integration

5.3 Bestandteile der Implementierung

Für die Implementierung wurde analog zu den anderen Jadex-Adaptern das Package `jadex.adapter.j2ee` gewählt. Alle nachfolgend genannten Komponenten, die Beispielanwendung sowie einige kleinere, hier nicht weiter aufgeführte Testanwendungen befinden sich unterhalb dieser Package-Struktur.

Bevor auf die einzelnen Komponenten näher eingegangen wird, soll mit Hilfe des UML-Diagramms (Unified Modeling Language) in Abbildung 20 ein Überblick über die Architektur und die Abhängigkeiten der Komponenten gegeben werden. Das Klassendiagramm ist an Abbildung 19 angelehnt und stellt die Architektur der Integration konkreter dar.

5.3.1 AdapterEntity

Den Kern der Implementierung bildet ein Adapter, der die Ausführung der BDI-Agentenlogik von Jadex in einem EJB Container ermöglicht. Da der Adapter den Zustand der Jadex-Agenten beinhaltet, wurde er als EJB 3.0 Entity modelliert, um den Zustand der Agenten persistent halten zu können. Folgende Attribute machen demnach den elementaren Zustand eines Agenten aus und wurden über entsprechende Annotations als solche gekennzeichnet:

- Als Primärschlüsselattribut wurde der Agentenname gewählt, da dieser innerhalb der Plattform eindeutig ist.
- Weitere Attribute sind zwei Boolean, die den Agenten als lebendig/tot beziehungsweise aktiv/inaktiv bezeichnen. Das erste Attribut wurde notwendig, da sich Entities nicht selbst löschen können. Sollte ein Agent also sich selbst beenden wollen, wird nur das Attribut gesetzt und er wird kurz darauf von einer außenstehenden Instanz entfernt.
- Letztlich sichert ein Byte Array den in `IJadexAgent` enthaltenen, internen Zustand der BDI-Logik des Agenten, indem das entsprechende Objekt mit `saveState()` in das Array geschrieben, beziehungsweise mit `getJadexAgent()` aus dem Array wieder eingelesen wird. Das Aktualisieren des Byte Arrays wird vom EJB Container durch die Annotations `@PrePersist` und `@PreUpdate` automatisch ausgeführt, bevor das Objekt in die Datenbank geschrieben wird. Warum das Problem der Speicherung des Agentenzustands auf diesem Weg gelöst wurde, wird in Kapitel 5.4.1 näher erklärt.

Alle Attribute werden wie üblich im Konstruktor initialisiert. Das `IJadex-Agent` implementierende Objekt eines BDI-Agenten wird von einer Factory des Jadex-Frameworks erzeugt, der die für die Erzeugung notwendigen Parameter übergeben werden. Abschließend wird mit `idleStateChanged()` dafür gesorgt, dass der Agent initial ausgeführt wird.

Darüber hinaus gibt es einer Reihe transienter (d.h. nicht persistenter) Attribute, die bei Bedarf genutzt werden, für den Zustand des Agenten aber nicht weiter relevant sind. Dies sind im Folgenden:

- Eine Referenz auf `IJadexAgent` enthält den deserialisierten Zustand des Agenten. Dieses Feld wird erst dann initialisiert, wenn auf die Referenz über `getJadexAgent()` zugegriffen wird.
- Alle der Plattform bekannten `IMessageEventTransports` werden auf Anfrage des Jadex Agenten über `getMessageEventTransports()` initialisiert und aus Gründen der Leistung in einem Array für weitere Anfragen zwischengespeichert.
- Ebenfalls zwischengespeichert wird eine Referenz auf den TimerDispatcher, der in Kapitel 5.3.2 näher beschrieben wird und die AID (AgentIdentifier) des Agenten.
- Ein Boolean gewährleistet, dass die Methode `idleStateChanged()` nicht rekursiv aufgerufen wird.
- Schließlich sichert ein Integer, dass ein Agent immer nur eine maximale Anzahl von Planschritten durchführt, bevor er persistiert wird. Dadurch wird verhindert, dass ein Agent durchgängig Aktionen ausführen kann. Durch die transaktionale Sperrung des Agentenzustands bedingt, würde er in dieser Zeit nicht auf eingehende Nachrichten reagieren können, da für die Zustellung der Nachrichten eine neue Transaktion erstellt wird.

Wie sich aus Kapitel 2.5.4 ersehen lässt, muss der Adapter `IAgentAdapter` implementieren, damit die BDI-Logik auf die durch den Adapter von der Plattform vermittelten Dienste zugreifen kann. Diese Methoden wurden wie folgt umgesetzt:

- `idleStateChanged()` wird vom Agenten aufgerufen, wenn ein Ereignis seinen Zustand verändert hat. In der Standalone-Plattform wird der dem Agenten zugehörige Thread wieder aufgeweckt, welcher das Ereignis verarbeit. Da EJBs entsprechend den in Kapitel 4.4.1 genannten EJB Restriktionen keine Threads manipulieren dürfen, wird die Ausführung der Agenten nur durch die Threads realisiert, die dem TimerDispatcher und dem MessageDispatcher vom Applikationsserver zugewiesen werden. Dazu ruft diese die Methode `executeAction()` an `IJadexAgent` auf und stellt darüber hinaus sicher, dass der Agent nur eine maximale Anzahl von Planschritten ausführt.
- `notifyIn(long)` realisiert das spätere Aufwecken des Agenten mit Hilfe des TimerDispachers.
- `isAgentThread(Thread)` dient in der Standalone-Plattform der Synchronisierung der auf den Agenten zugreifenden Threads. Da ein Entity gemäß der Java EE Spezifikation immer nur von einem Thread zur Zeit bearbeitet werden kann und sich der Applikationsserver um die Synchronisierung kümmert, ist diese Methode nicht weiter relevant.
- `killAgent()` vermerkt den Agenten als gelöscht, indem das entsprechende Attribut gesetzt wird.

- `getAgentIdentifier()` und `getMessageEventTransports()` initialisieren nur die entsprechenden Objekte und übergeben die dann dem Jadex Agenten.

Der wechselseitige Zugriff von `AdapterEntity` und `IJadexAgent` sowie die daraus verursachten, rekursiven Aufrufe von `idleStateChanged()` werden in Abbildung 21 dargestellt. Der Agent wird mittels `messageArrived()` über eine eingehende Nachricht benachrichtigt. Da sich aufgrund dieser Information der Agentenzustand geändert hat, teilt der Agent dem Adapter über `idleStateChanged()` mit, dass er wieder ausgeführt werden möchte. Der Adapter ruft daraufhin solange `executeAction()` am Agenten auf, bis dieser alle momentanen Aktionen ausgeführt hat oder eine gegebene Anzahl von Aktionen erreicht wurde. Letzteres verhindert die schon beschriebene durchgängige Ausführung des Agenten.

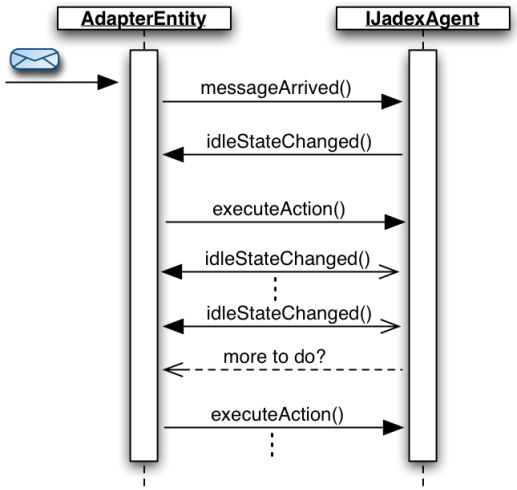

Abb. 21: Sequenzdiagramm des wechselseitigen Zugriffs von Adapter und Agent

Da sich bei der Ausführung von Aktionen der Zustand des Agenten normalerweise ändert (weil etwa bei der Verarbeitung der Nachricht ein neues Ziel übernommen wurde), ruft der Agent wiederholt `idleStateChanged()` auf, um dies dem Adapter mitzuteilen. Dieser Aufruf ist rekursiv, da sich der Agent schon innerhalb der Methode befindet. Der Adapter verhindert den rekursiven Aufruf, indem er beim Betreten von `idleStateChanged()` ein Boolean setzt, und dieses beim Verlassen der Methode löscht. Ist das Boolean beim Betreten der Methode schon gesetzt, so wird `executeAction()` nicht aufgerufen.

5.3.2 TimerDispatcher

Aufgabe des TimerDispatchers ist es, das Scheduling der Agenten durchzuführen. Dies ist nötig, da Agenten ihre eigenen Threads nicht verwalten dürfen,

sondern diese nur bei Bedarf zur Verfügung gestellt bekommen. Implementiert wurde der Dispatcher als zustandsloses Session Bean, da nur zustandslose EJBs den TimerService des EJB Container nutzen können.

Die nach außen sichtbare Schnittstelle umfasst genau eine Methode: Über `notifyAgentIn(IAgentAdapter, long)` kann sich ein Adapter (der `IAgentAdapter` imlementiert) für eine zukünftige Ausführung registrieren. Die Methode erzeugt einen Timer, der nach der gewünschten Zeit abläuft und zusätzlich den Namen des Agenten enthält, der den Timer registriert hat.

Die zweite Methode wird vom Container aufgerufen, wenn ein Timer abläuft. EJB 2.1 mussten dafür das Interface `javax.ejb.TimedObject` implementieren, mit EJB 3.0 reicht es, die entsprechende Methode mit der Annotation `@Timeout` zu kennzeichnen. Die Methode lädt mit Hilfe des zuvor beim Erstellen des Timers gespeicherten Agentennamens den entsprechenden Agenten aus der Datenbank und benachrichtigt ihn über `IJadexAgent.notifyDue()`.

5.3.3 MessageDispatcher

Bei dem MessageDispatcher handelt es sich um eine Message-driven Bean, dessen Aufgabe es ist, in einer bestimmten JMS Destination ankommende Nachrichten zu verarbeiten. Der EJB Container sorgt aufgrund der per `@ActivationConfigProperty` Annotation definierten Konfiguration dafür, dass das EJB für eine bestimmte Queue registriert wird, so dass bei einer eingehenden Nachricht die `onMessage(Message)` Methode des EJB aufgerufen wird.

In dieser Methode wird als erstes der Empfänger der Nachricht ermittelt. Da das EJB keine Informationen über die Kodierung der Nachricht hat, wird der Empfänger der Nachricht zusätzlich in einem JMS Feld gespeichert, auf das über einen Schlüssel (`JmsMessageAdapter.JMS_SENDER_KEY`) vom EJB zugegriffen werden kann. Das Message-driven Bean lädt den für die Nachricht zuständigen Jadex Agenten und übergibt ihm die in einem Adapter (`JmsMessageAdapter` implementiert `IMessageAdapter`) verpackte Nachricht.

Da die von Jadex verwendeten Agentennachrichten aus einer Reihe von Variablen bestehen, auf die über Schlüssel zugegriffen wird, hat sich der Autor entschieden, eben diese Variablen in einer JMS MapMessage zu verschicken. Als Variablen kommen beliebige primitive Werte und serialisierbare Objekte in Frage. Der Adapter ermöglicht den Agenten dabei einen transportunabhängigen Zugriff auf die Variablen.

Für den Versand von Nachrichten gibt es bisher nur den `JmsMessageTransport` für die Plattform-interne Kommunikation. Die einzige Methode `sendMessage(IRMessageEvent)` erzeugt eine in einem `JmsMessageAdapter` verpackte `MapMessage`, schreibt alle im `IRMessageEvent`

enthaltenen Variablen über den Adapter in die `MapMessage` und verschickt diese schließlich.

5.3.4 AMS, DF und Planbibliothek

Von der FIPA-Spezifikation übernommen wurden der AMS- und der DF-Agent.[32] Während ersterer den Lebenszyklus der Agenten verwaltet, ermöglicht der DF das Anbieten und Aufsuchen von Agentendiensten.

Beide Aufgabenbereiche wurden jeweils mit einem zustandslosen Session Bean realisiert. Mit Hilfe des AMS lassen sich etwa alle existierenden Agenten auf der Plattform anzeigen, neue Agenten erzeugen, bestehende Agenten löschen und Agenten aktivieren beziehungsweise deaktivieren. Das AMS arbeitet demnach nur auf dem Adapter der Agenten. Der DF verwaltet die von den Agenten angebotenen Dienste. Um die Informationen des DF persistent zu halten, wurde ein kleines Entity (`DFRegisteredAgents`) geschrieben, dass als Primärattribut den Agentennamen und als weiteres Attribut die vom Agenten angebotenen Dienste beinhaltet. Diese Dienstinformationen werden in `jadex.adapter.fipa.AgentDescription` gekapselt und in serialisierter Form in der Datenbank gespeichert. Die Aufgaben des DF umfassen das Erzeugen, Modifizieren und Löschen von DF Einträgen.

Zustandslose Session Beans zur Realisierung von AMS und DF wurden deswegen gewählt, da sich EJBs ohne großen Implementierungsaufwand von externen Anwendungen und internen Komponenten nutzen lassen. Dieser Punkt ist wichtig, da AMS und DF nicht nur von den Agenten genutzt werden, sondern auch von dem Benutzer. Schließlich muss dieser eine Möglichkeit haben, beispielsweise neue Agenten zu starten (siehe Kapitel 5.3.5).

Für die Nutzung von AMS und DF durch die Agenten wurde jeweils eine Capability erstellt, die im ADF eingebunden und dann genutzt werden kann. Beide Capabilities sind prinzipiell gleich aufgebaut. Durch eine Reihe von Zielen können die jeweiligen Dienste vom AMS beziehungsweise DF im ADF angesprochen werden. Zum Erreichen des Ziels wird bei beiden Capabilities zuerst versucht das jeweilige Ziel mit einem *lokalen Plan* zu erreichen. Der Plan ist lokal, weil er direkt auf das entsprechende Session Bean von AMS beziehungsweise DF zuzugreifen versucht. Da auf dieser Plattform laufende Agenten selbst Entities sind, kann der Plan sich über JNDI eine Referenz zu einem der Session Beans holen und die gewünschte Methode aufrufen. Für Agenten entfernter Plattformen gilt dies normalerweise nicht. Sie müssten das Session Bean unter der expliziten Nutzung eines ORBs ansprechen.

Alternativ dazu hat der Autor analog zu der AMS und DF Planbibliothek der Standalone-Plattform eine Reihe von *Remote-Plänen* geschaffen, die sich über einen beliebigen Nachrichtentransport an die lokalen Agenten für AMS und DF wenden, welche wiederum das entsprechende Session Bean aufrufen. Auf-

32 Der Kürze wegen im Folgenden AMS und DF genannt.

grund der in der Capability angegeben Reihenfolge, wird zuerst immer versucht, den lokalen Plan auszuführen. Sollte dieser fehlschlagen, etwa weil der anfragende Agent auf einer entfernten Plattform verweilt und auf dieser kein ORB verfügbar ist, wird der Remote-Plan aufgerufen. Dieser sollte in der Regel funktionieren, da der Plan über einen beliebigen Transport mit dem lokalen AMS beziehungsweise DF der Zielplattform kommuniziert.

5.3.5 Webinterface

Da sich bei der Installation einer Anwendung auf den Applikationsserver nicht automatisch vorbestimmte Entities erzeugen lassen, muss ein anderer Weg gefunden werden, Agenten zu erzeugen. Die Lösung dieses Problems ist eine einfache Benutzerschnittstelle in Form eines Webinterface, das mit JSF realisiert wurde. Dieses ermöglicht den Benutzern, alle vom AMS und DF angebotenen Dienste zu nutzen, wozu zum Beispiel das Erzeugen und Beenden von Agenten gehört. Das Webinterface greift dabei direkt auf die von Session Beans realisierten Dienste des AMS beziehungsweise DF zu.

Abb. 22: Webinterface für die Dienste des AMS

Abbildung 22 zeigt einen Teil des Webinterface. Die dargestellte Form ermöglicht die Nutzung der AMS Dienste: Alle existierenden Agenten werden angezeigt und über entsprechende Schaltflächen können einzelne Agenten aktiviert/deaktiviert und gelöscht werden. Ein weiterer Link ermöglicht die Erzeugung neuer Agenten.

5.4 Aufgetretene Probleme

Wie bei der Realisierung von Softwareprojekten üblich, traten bei der Implementierung des im Rahmen dieser Arbeit gewählten Integrationsansatzes ebenfalls Probleme auf, die es zu lösen beziehungsweise zumindest zu umge-

hen galt. Die aufgetretenen Probleme sind dabei selten die Folge einer mangelhaften Planung, vielmehr sind sie auf fehlerbehaftete oder unzureichend dokumentierte Bibliotheken und Frameworks zurückzuführen, die für die Implementierung genutzt wurden.

Folgend wird deswegen nacheinander auf die bei der Implementierung des Integrationsansatzes aufgetretenen Probleme eingegangen. Es wird erklärt, worin die Probleme bestanden, wodurch sie verursacht und wie sie gelöst wurden.

5.4.1 Abbildung auf ein Datenbankschema

Zu Beginn dieser Arbeit galt es sich einen Überblick über die Funktionsweise von Jadex und dem JBoss Applikationsserver zu verschaffen. Während von letzterem nur die Dienste in Anspruch genommen werden, die in der Java EE Spezifikation und dem Benutzerhandbuch[33] ausreichend dokumentiert sind, musste für Jadex eine detailliertere Analyse durchgeführt werden. Dies war notwendig, da das Framework nicht nur aufgrund der implementierten BDI-Logik genutzt wird, sondern auch der Zustand dieser Logik auf ein Datenbankschema abgebildet werden muss. Das Durcharbeiten des Jadex Quelltextes zur Identifikation zustandsbehafteter Klassen samt ihrer Attribute wurde durch die nachfolgend beschriebenen Gründe soweit erschwert, dass der Autor die Idee einer Datenbankabbildung verwarf und den Zustand der Agenten stattdessen wie in Kapitel 5.3.1 einfach serialisiert ablegt.

Fehlende Trennung von fachlichen Werten und Logik

Üblicherweise geschieht die Abbildung auf ein Datenbankschema durch das Nachvollziehen des abzubildenden Objektgraphen, der verwendeten fachlichen Gegenstände und Konzepte. Viele Entwicklungsmethodiken wie etwa WAM (Werkzeug-Automat-Material) und MVC sehen eine Trennung der fachlichen Werte (Terminus WAM: *Material*; MVC: *Modell*) von der fachspezifischen Logik (Terminus WAM: *Werkzeug*; MVC: *Controller*) vor. Diese Trennung reduziert die Komplexität der Anwendung und erhöht die Wiederverwendbarkeit der einzelnen Komponenten (vgl. [GHJV97]). Ein Objektgraph entsteht durch Abhängigkeiten zwischen einzelnen Materialien, die in der Regel als einzelne Klassen implementiert werden. Hat zum Beispiel ein Material *Kunde* neben Kundennummer, Vornamen und Nachnamen noch eine *Anschrift*, die wiederum als eigenständiges Material modelliert wurde, so ist das Material *Kunde* von *Anschrift* abhängig. Man spricht in dem Fall von einer gerichteten Assoziation. Idealerweise ergeben diese Abhängigkeiten einen Baum, dessen Blätter unabhängige Materialien darstellen.

Die Trennung von Datenhaltung und Datenverarbeitung vereinfacht den Abbildungsprozess, da nur die Materialien abgebildet werden müssen. So besteht das Erstellen einer Abbildungskonfiguration für einen O/R Mapper neben der

33 `http://docs.jboss.org/jbossas/jboss4guide/r4/html/`

Abbildung von Materialklassen auf Relationen und der Abbildung primitiver Variablen der Klassen auf Attributnamen, ebenfalls aus der Abbildung von Abhängigkeiten zwischen den Klassen auf Primär- und Fremdschlüssel. Die Konfiguration wird bei den meisten O/R Mappern in einer externen XML-Datei abgelegt. EJB 3.0 Entities verwenden stattdessen Annotations, um diese Konfigurationsinformationen direkt in den Klassen abzulegen.

Wie sich bei der Analyse von Jadex zeigte, gibt es keine Trennung von fachlichen Werten und Logik. Die erkennbaren Strukturen zeigen stattdessen eine saubere Trennung zwischen Agentenmodell (siehe Kapitel 2.5.2) und instanziierten Modell auf. Innerhalb des instanziierten Modells, dessen Klassen sich überwiegend im Package `jadex.runtime` befinden, lassen sich allerdings keine weiteren Strukturen erkennen, die eine Abbildung vereinfacht hätten. Letztlich hätte für eine Abbildung der gesamte Quelltext durchgearbeitet werden müssen, wobei für jede Instanzvariable einer Klasse entschieden werden müsste, ob sie zustandsrelevante Informationen oder nur temporäre Laufzeitinformationen enthält.

Zyklische Abhängigkeiten

Mit Hilfe des Software-Tomographen[34] (*Sotograph* genannt) – einem Werkzeug der Software-Tomography GmbH zur Analyse der Qualität von Architektur und Quelltexten beliebiger Anwendungen – hat der Autor feststellen können, dass große Teile des Objektgraphen zyklisch abhängig sind. Abbildung 23 kann diese Abhängigkeiten nur ansatzweise verdeutlichen, indem sie die Zyklen auf Package-Ebene für `jadex.runtime` darstellt. Schwarze Linien stellen Abhängigkeiten zwischen den Packages dar, während die helleren paketübergreifende Vererbung symbolisieren. Die Strichstärke gibt den Grad der Abhängigkeiten beziehungsweise die Anzahl der Vererbungen wieder. Eine Darstellung mit der Granularität von Klassen wäre zwar ebenso möglich, allerdings würde man in diesem Graphen nichts mehr erkennen können.

Zyklische Abhängigkeiten erschweren das Verständnis des Quelltexts insofern, als dass es im Umkehrschluss weniger unabhängigen Klassen in dem Objektgraphen gibt, die leichter zu verstehen sind, da eben diese Unabhängigkeit Wechselwirkungen mit anderen Klassen unterbindet.

34 `http://www.software-tomography.com/html/sotograph.htm`

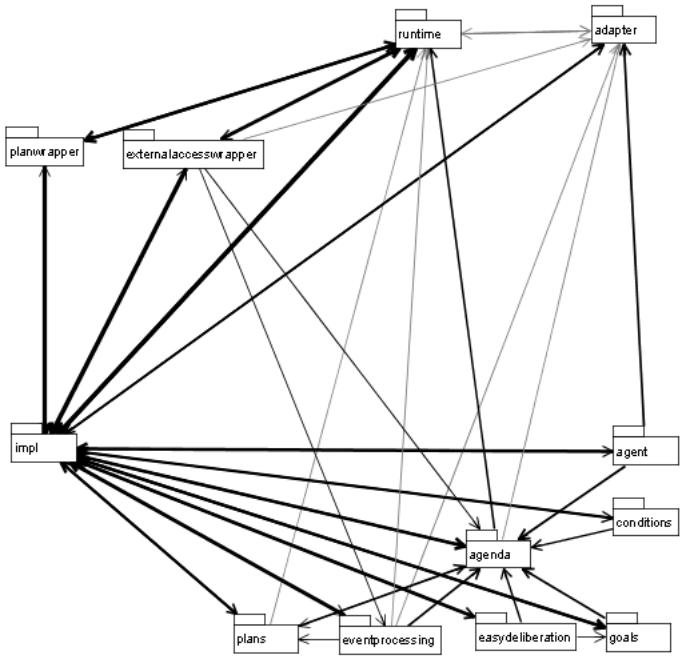

Abb. 23: Packetabhängigkeiten des Jadexframeworks

Abstraktion durch Interfaces

Ein weiteres Problem für die Abbildung entstand dadurch, dass die Entwickler konsequent Interfaces als Abstraktionsmittel genutzt haben. Dies ist eigentlich löblich und im Fall von Jadex notwendig, um zum Beispiel den kontextunabhängigen Zugriff auf die Komponenten der Jadex-Agenten zu gewährleisten. Abhängig davon, ob der Zugriff innerhalb eines Plans oder von außerhalb (etwa durch eine GUI) geschieht, werden andere Mechanismen für den Zugriff gewählt, um ihn zu synchronisieren.

Wie der Autor allerdings feststellen musste, scheint es mit der polymorphen Nutzung von Beziehungen in der genutzten JBoss Version Probleme zu geben. Sobald eine der Referenzen auf ihren konkreten Typ gecastet werden sollte, gab der Applikationsserver eine Fehlermeldung über eine angeblich fehlerhafte Typkonvertierung zurück. Da die EJB 3.0 Persistenz-Spezifikation ([DeKe05b]) allerdings Polymorphie vorsieht, geht der Autor von einem Fehler in der Implementierung der Spezifikation aus. Entsprechende Anfragen im JBoss Forum konnten allerdings bis zum Zeitpunkt der Abgabe dieser Arbeit nicht zufriedenstellend beantwortet werden.

Fazit

Sowohl die technischen Probleme bei der Realisierung von Polymorphie, als auch das erschwerte Verständnis des Quelltextes, bedingt durch zyklische Abhängigkeiten und fehlende Separation zwischen Daten und Anwendungslogik, haben den Autor dazu bewogen, den Objektgraph stattdessen serialisiert in der Datenbank zu speichern. Es soll an dieser Stelle erwähnt werden, dass eine

objekt-relationale Abbildung auf jeden Fall möglich ist. Allerdings setzt diese entweder ein intimes Verständnis der Funktionsweise des Jadex Frameworks oder ein Refactoring des Quelltextes mit dem Ziel einer besseren Strukturierung voraus. Beides würde den Zeitrahmen dieser Arbeit bei weitem überschreiten, so dass eine einfachere Lösung angebrachter erschien.

5.4.2 Synchronisierter Agentenzugriff

Ein weiteres Problem machte sich erst zur Laufzeit der Agenten bemerkbar, als diese unter einer gewissen *Last* getestet wurden. Der Begriff *Last* bezeichnet die von einem System verarbeitete Arbeitsmenge [HaRa01] und ist in diesem Fall wie folgt zu verstehen: Bei jeder eingehenden Nachricht und bei jedem abgelaufenen Timer eines Agenten stellt der EJB Container einen Thread bereit, der das Ereignis abarbeiten soll. Zu diesem Zweck wird in beiden Fällen zuerst der betreffende Agent ermittelt, sein Zustand über den O/R Mapper aus der Datenbank geladen und anschließend über das jeweilige Ereignis in Kenntnis gesetzt. *Last* tritt dann auf, wenn mehrere Ereignisse gleichzeitig oder zumindest zeitnah eintreffen, so dass mehrere, jeweils in einer eigenen Transaktion laufende Threads um den Zugriff auf den Agentenzustand konkurrieren.

Die Synchronisierung dieser Zugriffe wird per Spezifikation durch den Container geregelt, das heißt es ist Aufgabe der Container- und damit Applikationsserverhersteller, den Zugriff auf die EJBs beziehungsweise EJB 3.0 Entities zu synchronisieren. Da der Container die Synchronisierung erledigt, braucht sich der Anwendungsentwickler nicht um diesen Aspekt des nebenläufigen Zugriffs zu kümmern. Tatsächlich wird ihm sogar per Restriktion (siehe Kapitel 4.4.1) die manuelle Synchronisierung verboten. Nach [DeKe05c] gibt es zwei verschiedene Varianten wie Containerhersteller die Synchronisierung realisieren können:

- Bei einem **optimistischen Sperrprotokoll** aktiviert der Container mehrere Instanzen eines Entity beziehungsweise Entity Bean, die in verschiedenen Transaktionen benutzt werden können. Bei Abschluss der Transaktion wird geprüft, ob zwischenzeitlich eine andere Transaktion den Zustand des Entity verändert hat. Sollte das der Fall sein, wird die gerade abschließende Transaktion zurückgesetzt. Um die Änderungen durch Transaktionen nachvollziehen zu können, wird in der Regel eine Versionsnummer zusammen mit dem Entity abgelegt. Diese Art der Synchronisation ist dann vorteilhaft, wenn überwiegend lesender Zugriff auf die Entities geschieht.
- In einem **pessimistisches Sperrprotokoll** erwirbt die Transaktion eine exklusive Sperre auf das Entity. Alle anderen Transaktionen müssen beim Zugriff auf dasselbe Entity warten, bis die aktuelle Transaktion abgeschlossen ist. Der Zugriff geschieht also rein seriell. Aufgrund der exklusiven Sperrung von Entities sind pessimistische Sperren bei überwiegend lesendem Zugriff weniger leitungsfähig als optimistische.

Da der JBoss Applikationsserver in der Standardkonfiguration optimistische Sperren verwendet und eine Änderung dieser Einstellung nicht durchgeführt werden konnte, kam es bei wiederholten Versuchen, in denen Agenten unter Last Arbeit zu verrichten hatten, zu einem Einbruch der Leistung und gar zu nicht deterministischen Verhalten des Systems. Dazu muss zuerst bemerkt werden, dass sich der Zustand eines Agenten bei einem Zugriff auf denselben in der Regel verändert, das heißt rein lesender Zugriff kommt so gut wie nie vor. Treten mehrere Ereignisse für einen Agenten zeitnah ein, konkurrieren wie oben beschrieben mehrere Transaktionen um den Zugriff. Bei einem optimistischen Sperrprotokoll werden alle Transaktionen parallel ausgeführt, allerdings kann nur die schnellste Transaktion erfolgreich abschließen. Bei allen anderen muss der Container aufgrund der zwischenzeitlichen Modifikation des Agentenzustands die Transaktionen zurücksetzen. Je mehr Transaktionen also um den Zugriff konkurrieren, desto mehr Transaktionen werden wahrscheinlicher zurückgesetzt. Der beschriebene Effekt ist unter dem Begriff *Trashing* (dt. wegwerfend) bekannt und ist ein Nachteil optimistischer Sperrprotokolle (vgl. [HaRa01]).

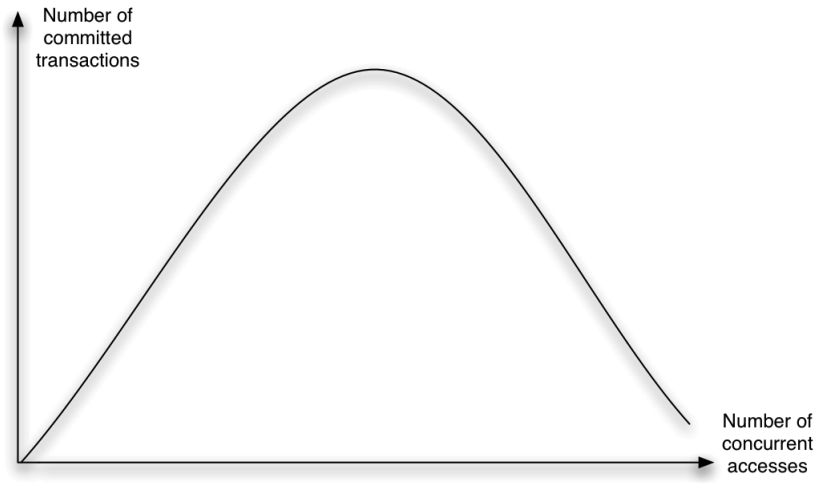

Abb. 24: Leistungsverhalten optimistischer Sperrprotokolle beim schreibenden Zugriff

Abbildung 24 verdeutlicht dieses Verhalten, indem sie die Anzahl nebenläufiger Zugriffe auf einen Agenten über die Anzahl erfolgreich abschließender Transaktionen aufträgt. Bis zu einem gewissen Grad der Nebenläufigkeit steigt die Gesamtleistung des Systems. Wird aber ein diesem Fall unbestimmter Punkt überschritten, überwiegen die durch optimistische Sperren verursachte Rollbacks (dt. Zurücksetzen) der Transaktionen und die Gesamtleistung, also der Anteil der erfolgreich abschließenden Transaktionen, bricht ein.

Da es sich bei JMS Broker und TimerService um transaktionale Ressourcen handelt, wäre dieses Verhalten zwar suboptimal, aber im Rahmen dieser Arbeit nicht weiter gravierend, da es sich bei der Implementierung ja um kein aus-

gereiftes Softwareprodukt handeln muss. Zurückgesetzte Transaktionen würden bedeuten, dass zugestellte Nachrichten und ausgelöste Timer ebenfalls zurückgesetzt werden würden. Nach einer bestimmten Frist würde der Container versuchen, die entsprechenden Nachrichten nochmals zuzustellen beziehungsweise die abgebrochen Timer erneut zu aktivieren. Allerdings musste der Autor bei der Nutzung des JBoss Applikationsserver feststellen, dass die JBoss Implementierung des TimerService nicht immer so transaktional ist wie sie theoretisch sein sollte. So werden beispielsweise Timer ausgelöst, obwohl die den Timer erstellende Transaktion immer noch nicht abgeschlossen ist und die definitive Auslösung des Timers damit noch gar nicht garantiert ist. Dieses Verhalten sorgt bei den Agenten für ein nicht nachvollziehbares Verhalten, da der Agent zum Teil von ausgelösten Timern aufgeweckt wird, obwohl diese Timer gar nicht existieren sollten. Nach einem Bericht der entsprechenden Fehler im JBoss Forum sollen sie im nächsten Release (dt. Veröffentlichung) des Applikationsservers behoben sein.

Als vorläufige Lösung wurde stattdessen eine Komponente entwickelt, die den Zugriff auf die Agenten über ein Semaphor manuell serialisiert.[35] Dazu wird in der Klasse `LockAgents` eine statische Liste von Agentennamen geführt, die momentan gesperrt sind und auf die über die statischen Methoden `lockAgent()` und `releaseAgent()` (jeweils mit dem Agentennamen als Parameter) zugegriffen werden kann. Der Zustand eines Agenten wird mittels der statischen Methode `AdapterEntity.loadAndLock()` realisiert, der unter anderem der Agentenname übergeben wird. Innerhalb der Methode wird als erstes der Agentennamen gesperrt, anschließend der Agentenzustand geladen und zurückgegeben. Falls der Agent momentan schon in einer anderen Transaktion genutzt wird, wird die Transaktion beim Sperren des Agenten solange blockiert bis der Agent wieder freigegeben wurde. Die Freigabe erfolgt dabei nach Abschluss der Transaktion in der Methode `AdapterEntity.releaseResources()`, die aufgrund der Annotations `@PostUpdate`, `@PostPersist` und `@PostRemove` nach dem Persistieren von Veränderungen, neu angelegten und gelöschten Agenten automatisch vom Container aufgerufen wird. Diese Lösung ist als Workaround (dt. provisorische Lösung) gedacht. Sie verhindert zwar einen Einsatz der Agentenplattform in einem Cluster, sorgt aber gleichzeitig für ein ausgewogenes Laufzeitverhalten des Systems.

5.5 Evaluation anhand einer Beispielanwendung

Zur Demonstration und Validierung der Integrationslösung wird im Rahmen dieser Arbeit eine innerhalb eines bestimmten Szenarios angesiedelte Beispielanwendung realisiert. In Frage kommende Anwendungsdomänen des Szenarios sind jene, in denen üblicherweise Agententechnologie und Applikationsserver zum Einsatz kommen. Da der Fokus dieser Arbeit in der Analyse der

35 Mit *Serialisierung* ist hier nicht der gleichnamige Mechanismus zur Speicherung von Objekten gemeint, sondern die Nacheinanderausführung nebenläufiger Prozesse.

Integrationsmöglichkeiten und der Implementierung einer solchen liegt, wird das Szenario nur einen Teil der Domäne abdecken.

Die Anwendung selbst sollte neben den von der Plattform erbrachten Diensten wie Nachrichtentransport, Scheduling, AMS und DF, ebenso die durch die Nutzung eines Applikationsservers bedingten Vorteile aufzeigen. Dazu zählen etwa die Anbindung beliebiger weiterer Komponenten und die inhärenten Eigenschaften der EJBs wie Persistenz und Transaktionalität.

Folgend wird zuerst das vom Autor gewählte Szenario dargelegt, bevor auf die Architektur der Anwendung und die technischen Details der Realisierung näher eingegangen wird.

5.5.1 Szenariobeschreibung

Mittlerweile stellt jeder größere Verkehrsnetzbetreiber eigene Werkzeuge zur Routenfindung zur Verfügung, mit denen der Kunde sich unter Angabe von Start-, Zielort und optional weiteren Parametern eine Reiseroute innerhalb des jeweiligen Betreibernetzes zusammenstellen kann. Bekannte Beispiele sind die bundesweit agierende Deutsche Bahn AG[36] und der hamburgische Verkehrsnetzbetreiber HVV[37]. Lässt sich aber eine Reise nicht innerhalb des Streckennetzes eines Betreibers durchführen, müssen mehrere Verkehrsunternehmen in Anspruch genommen werden. In Ermangelung entsprechender Hilfsmittel bleibt die betreiberübergreifende Planung häufig dem Kunden überlassen, so dass dieser die Auswahl der geeigneten Verkehrsmittel koordinieren muss. Zwar ermöglichen einige Verkehrsnetzbetreiber aufgrund von Kooperationsverträgen mit anderen Betreibern eine Suche in einem erweiterten Streckennetz (siehe HVV), diese Lösungen bieten aber trotzdem keine betreiberübergreifende Planung.

Für das zur Evaluation gewählte Szenario wurde deswegen ein Reiseplanungssystem ausgesucht, das wie folgt arbeitet: Mit Hilfe eines als Agent modellierten Reiseplanungsassistenten soll eine vom Benutzer gewünschte Reiseroute gesucht werden. Dazu sucht der Assistent zuerst nach verfügbaren Reiseunternehmen und kontaktiert diese anschließend, um eine den Vorgaben des Benutzers entsprechende Reiseroute zusammenzustellen.

Es würde den Rahmen dieser Arbeit bei weitem überschreiten, reale Streckennetzinformationen für dieses Szenario zu verwenden, da die Daten der Betreiber – selbst wenn sie über eine offen gelegte Dienstschnittstelle (etwa als Web Service) angesprochen werden könnten – in einem proprietären Format zurückgeliefert werden würden, das jeweils in ein bekanntes Format zu konvertieren wäre. Stattdessen hat sich der Autor entschieden, einige fiktive Verkehrsnetzbetreiber zu modellieren, deren Daten in einem einheitlichen Format, respektive einer gemeinsamen Ontologie vorliegen. Da es sich in einem

36 `http://reiseauskunft.bahn.de/bin/query.exe/dn/`
37 `http://www.geofox.de`

Agentensystem anbietet, wurden die Betreiber als Agenten modelliert, deren Dienste ebenfalls einheitlich angesprochen werden können.

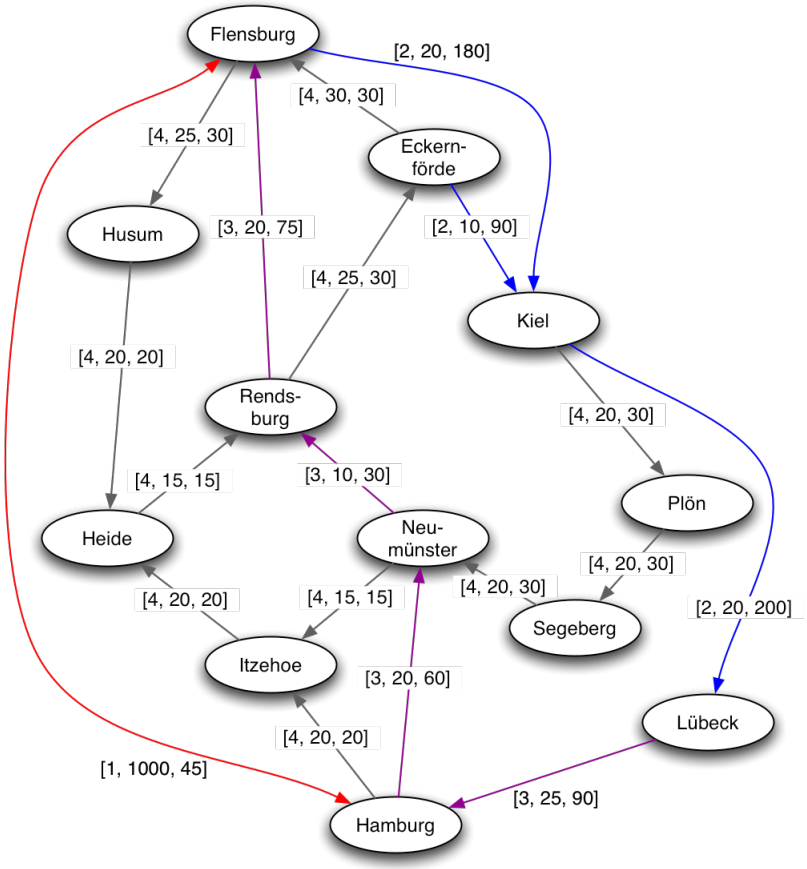

Abb. 25: Abstrakte Darstellung eines Streckennetzes

Zu diesem Zweck wurde ein Streckennetz erstellt, dessen abstrakte Darstellung in Abbildung 25 zu sehen ist. In Form eines Graphen mit Orten als Knoten und gerichteten, gewichteten Kanten als Routen soll ein einfaches, verschiedene Betreiber verknüpfendes Streckennetz modelliert werden. An den Kanten befinden sich deswegen mehrdimensionale Gewichtungen, die in der Form [Betreibernummer, Reisekosten, Reisedauer] abgelegt werden. Über eine entsprechende GUI kann ein Benutzer dann Start, Ziel und optional die maximalen Reisekosten sowie -dauer angeben. Der Assistent versucht aufgrund dieser Vorgaben daraufhin eine Strecke zusammenstellen und übergibt das Ergebnis dann der GUI.

5.5.2 Umsetzung

Die Implementierung des Szenarios befindet sich im Package `jadex.adapter.j2ee.examples.travel` und enthält neben den Agenten das Objektmodell des Graphen. Die Klasse `Route` stellt eine nicht

weiter teilbare Etappe einer Route dar, die über Etappenstart, -ziel sowie Reisedauer und Kosten der Etappe verfügt.

Die Verkehrsnetzbetreiber werden durch einen Agententypen namens `TravelCompany` repräsentiert, der über unterschiedliche initiale Zustände als einer von vier Betreibern gestartet werden kann. Entsprechend des initialen Zustands unterscheidet sich das Streckennetz von dem der anderen Betreiber. Nach ihrem Start registrieren sich die Agenten der Betreiber am DF und publizieren dort ihren Suchdienst, so dass er vom Assistenten gefunden werden kann.

Der Reiseassistent ist im Typ `TravelAssistant` umgesetzt. Er nimmt Anfragen vom Benutzer entgegen und startet bei jeder Anfrage mit Hilfe des AMS einen Arbeiter (`Worker`), der die Anfrage abarbeitet. Dazu übergibt er ihm die Benutzervorgaben und den Namen des Anfragenden, damit auf die Anfrage geantwortet werden kann.

Ein `Worker` existiert immer nur für die Dauer einer Anfrage und beendet sich selbst, nachdem er das Suchergebnis zurückgeschickt hat. Er holt sich als erstes eine Liste aller am DF registrierten `TravelCompanies` und beginnt danach sukzessiv, eine Route zwischen Start- und Zielort unter Beachtung der Benutzervorgaben zu suchen. Dazu wird eine für mehrdimensionale Kantengewichtungen angepasste Variante des Dijkstra-Algorithmus (vgl. [Dijk59]) benutzt. Dieser Algorithmus arbeitet auf zwei Mengen von Orten, die in der Beispielanwendung die Knoten darstellen: Die Menge aller besuchten Orte und die Menge der bekannten, aber noch nicht besuchten Orte, in der der Startort zu Beginn der Suche liegt.

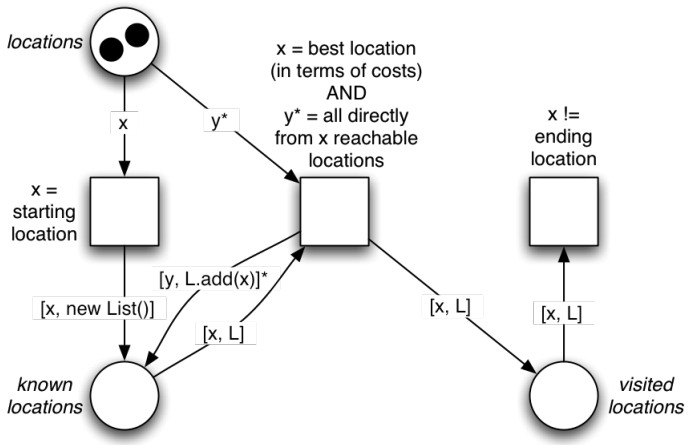

Abb. 26: Petrinetzdarstellung des Dijkstra-Algorithmus

Zu Beginn dieses iterativen Algorithmus befinden sich wie aus Abbildung 26 ersichtlich alle Orte in der Stelle *locations*. Mit einer initialen Transition wird der Startpunkt der Anfrage in die Menge der bekannten Orte (*known locations*) geschoben. Aus der Menge der bekannten Orte wird dann derjeni-

ge Ort ausgewählt, der am schnellsten beziehungsweise günstigsten erreichbar ist und die Benutzeranforderungen an die Route erfüllt. Wie Reisedauer und Preis gegeneinander aufgewogen werden wird durch den Ausdruck `choose_next_nodes` im ADF von `Worker` bestimmt und ist leicht anpassbar. Der ausgewählte Ort wird aus der Menge der bekannten entfernt und zu der Menge der besuchten (*visited locations*) hinzugefügt. Anschließend wird überprüft, welche neuen Orte direkt von dem gerade besuchten Ort erreichbar sind, indem entsprechende Anfragen an die `TravelCompanies` geschickt werden. Die Ergebnisse werden jeweils mit der Information der bisher zurückgelegten Strecke zu der Liste der bekannten Orte hinzugefügt, falls sie noch nicht besucht wurden. Der `Worker` wartet dabei entweder auf den Erhalt aller Antworten oder eine bestimmte Zeitspanne, bevor er mit der Suche fortfährt. Zu langsame Antworten werden verworfen, so dass die Suchzeit des `Workers` nicht von der Erreichbarkeit der `TravelCompanies` abhängt. Danach beginnt die nächste Iteration. Wieder wird durch den oben genannten Ausdruck ein Ort ausgewählt und verschoben und die Menge der bekannten Knoten aktualisiert.

Der Algorithmus endet, wenn der Zielort in der Menge der besuchten Orte liegt oder wenn der Ausdruck zur Wahl des nächsten Knoten keinen liefert, zum Beispiel weil die Menge der bekannten Knoten nur solche enthält, die nach den Benutzervorgaben nicht erreichbar sind. Nach Abschluss der Suche wird der Benutzer über das Ergebnis informiert. In Abbildung 26 verbleibt aufgrund einer Transition nur der Zielort in *visited locations*. Da für jeden Ort in einer Liste die Zwischenstationen gespeichert werden, mit denen der Ort ereicht wurde, ist die Reiseroute damit ermittelt.

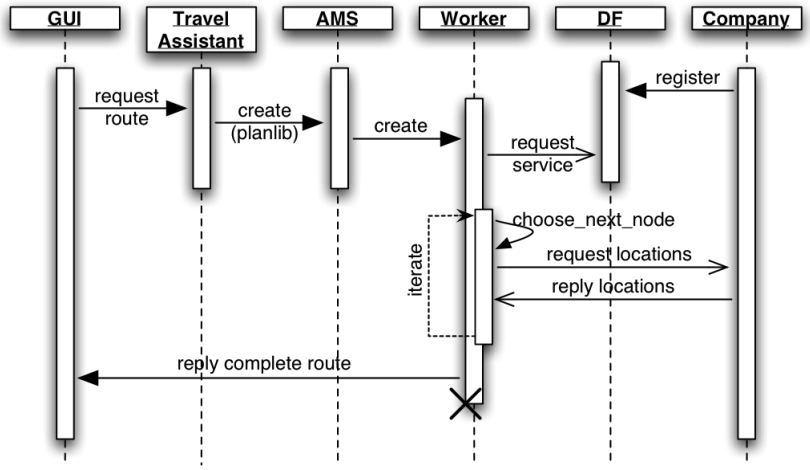

Abb. 27: Sequenzdiagramm einer Suchanfrage

Abbildung 27 verdeutlicht den Programmfluss bei einer Suchanfrage und den Zusammenhang der eben genannten Komponenten, indem die Aufrufhierarchie zwischen GUI, `TravelAssistant`, `Worker` und `TravelCompanies` in einem Sequenzdiagramm dargestellt wird.

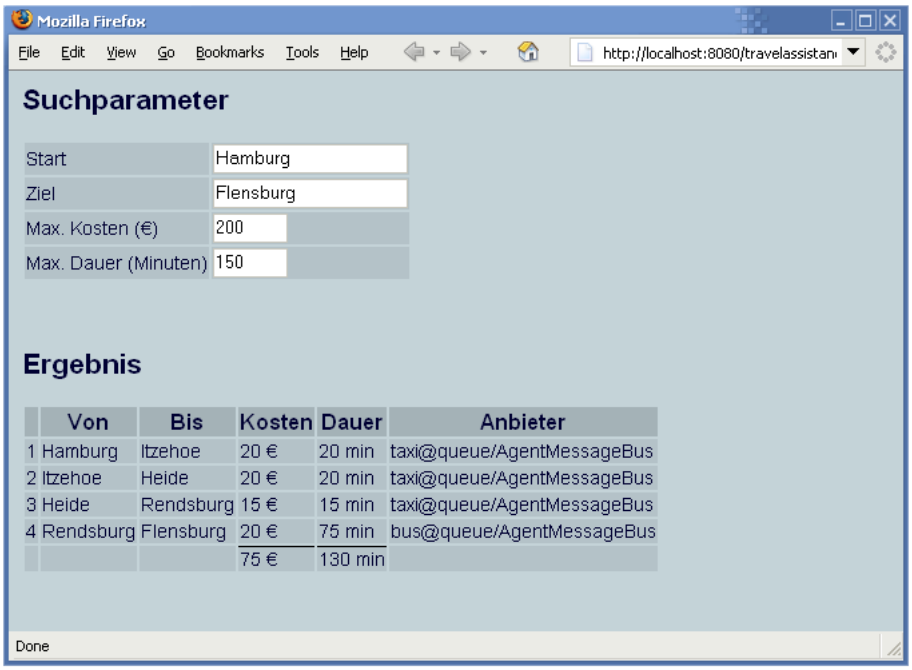

Abb. 28: Webinterface der Beispielanwendung

Die in Abbildung 28 dargestellte GUI ist ein einfaches, mit JSF gestaltetes Webinterface. Da JMS zum Nachrichtentransport zwischen Agenten verwendet wird, kann die GUI ebenfalls mit den Agenten kommunizieren, indem sie sich einfach als ein weiterer Agent ausgibt. Die Klasse `CommBusConnection` kapselt diesbezüglich das Kodieren und Dekodieren der Nachrichten, so dass der Zugriff auf den Assistenten aus Sicht der GUI nahezu transparent geschieht.

5.6 Zusammenfassung

Die zuletzt beschriebene Beispielanwendung hat gezeigt, dass eine Integration von proaktiven Agenten und bewährter Middleware trotz paradigmatischer Unterschiede, wie der Kontrolle der Ausführung, machbar ist. Während die Komponenten eines Applikationsservers vom Container kontrolliert ausgeführt werden, sieht das Agentenkonzept eine selbstkontrollierte und damit autonome Ausführung vor. Durch die Standardisierung der Schnittstellen ist die Erweiterbarkeit der Agentenplattform relativ unproblematisch und vor allem portabel. Die Persistierung der Agenten und die Clusterfähigkeit vieler Applikationsserver ermöglicht darüber hinaus große Agentensysteme, da nur aktive Agenten Ressourcen wie Arbeitsspeicher und Rechenzeit in Form von Threads belegen. Der Zustand inaktiver Agenten wird in der Datenbank abgelegt und bei Bedarf (z.B. bei einer eingehenden Nachricht) wieder geladen.

Sowohl die Serialisierung des Agentenzustands als auch die manuelle Synchronisierung des Zugriffs auf die Agentenzustände sind als Workarounds zu

verstehen und haben ihre Nachteile. Die Lösungen dieser Probleme beeinträchtigen aber nicht die Grundfunktionalität der implementierten Plattform, wie Nachrichtenaustausch und Scheduling der Agenten.

Bei der Programmierung von Agenten müssen allerdings eine Reihe von Einschränkungen hingenommen werden, um den Anforderungen eines Applikationsservers an seine Anwendungskomponenten gerecht zu werden. Diese sollen an dieser Stelle zusammengefasst werden:

- Aufgrund der Restriktion von EJB bezüglich der Threadmanipulation dürfen Jadex-Agenten nur *mobile Pläne* innerhalb eines Applikationsservers verwenden.
- Da untätige Agenten persistiert werden, müssen *Beliefs und Pläne serialisierbar* sein, das heißt entsprechend benutzte Klassen müssen `java.io.Serializable` implementieren.
- Eigenhändiger Zugriff auf *Threads*, *ServerSockets* und andere die Laufzeit von EJBs beeinflussende Mittel ist ebenfalls per Restriktion untersagt.
- Da Applikationsserver häufig nur in einer Konsolenumgebung laufen, ist die direkt Nutzung *grafischer Benutzerschnittstellen* im Plancode ebenfalls verboten. Die Interaktion mit den Agenten kann nur über Nachrichten erfolgen.
- Die Verwendung *statischer Variablen* ist eingeschränkt möglich, allerdings muss dann auf die Fähigkeit zur Clusterbildung des Applikationsservers verzichtet werden.

Die oben genannten Einschränkungen reduzieren die Einsatzgebiete der Agenten keineswegs, sie verkomplizieren aber die Entwicklung der Agenten in manchen Fällen, insbesondere durch die erzwungene Nutzung mobiler Pläne. Bei diesen wird bei jeder ansonsten blockierenden Aktion wie `dispatchSubgoalAndWait()` oder `sendMessageAndWait()` die Methode nach Erfüllung der Wartebedingung von neuem ausgeführt, so dass größere und damit unelegante Verzweigungsblöcke notwendig werden.

Kapitel 6

Schlussbetrachtung

Diese Arbeit schließt mit einem Resümee über die erreichten Ergebnisse. Die in der Einleitung vermittelte Zielsetzung wird nochmals aufgegriffen und auf ihre Erfüllung hin überprüft.

In dem anschließenden Ausblick wird kurz auf Entwicklungsmöglichkeiten eingegangen, die im Rahmen dieser Arbeit nicht mehr umgesetzt werden konnten. Dazu gehören etwa andere Lösungen der in Kapitel 5.4 angesprochenen Probleme, aber auch gänzlich neue Komponenten, welche die Plattform erweitern und beispielsweise FIPA-konform gestalten würden.

6.1 Zusammenfassung

Agententechnologie bietet eine ideale Grundlage zur Entwicklung von Anwendungen in dynamischen Umgebungen, da Agenten im Gegensatz zu konventioneller Software auf unvorhergesehene Systemzustände adäquat reagieren können. Agentenplattformen ermöglichen mittels geeigneter ACLs die plattformübergreifende Kommunikation – Agentensysteme sind demnach inhärent verteilte Systeme.

Akzeptierte Middleware wie etwa die im JCP spezifizierte Java EE Plattform realisieren ebenfalls eine Umgebung für verteilte Anwendungen. Während die Agententechnologie aber Agenten als Einheit der Verteilung vorsieht, sind dies in Java EE beliebige, objektorientiert entwickelte Komponenten. Durch ihre Laufzeitumgebung wird den Komponenten häufig geforderte Funktionalität zur Verfügung gestellt, so dass man sich bei der Entwicklung von Komponenten auf die Implementierung der Geschäftslogik konzentrieren kann. Technische Details bleiben außen vor und werden von der Laufzeitumgebung realisiert.

Bei der Analyse bestehender Integrationslösungen hat sich gezeigt, dass alle komponentenbasierten Ansätze kommerzielle Produkte sind. Frei erhältliche Integrationen realisieren dagegen einen containerbasierten Ansatz, der nur die Verwaltungsfunktionen eines Applikationsservers nutzen kann. Der Vorteil einer komponentenbasierten Integration liegt wie in Kapitel 4.5 dargestellt in der Unterstützung zahlreicher nichtfunktionalen, vom EJB Container realisierten Eigenschaften. Wie in Kapitel 5.6 erklärt wurde, ermöglichen insbesondere die persistente Speicherung der Agenten und die vom Applikationsserver realisierte Clusterbildung deutliche größere Agentensysteme als arbeitsspeicherbasierte Plattformen wie beispielsweise JADE leisten könnten.

Durch die Nutzung von JMS für den Transport von Agentennachrichten ist es des Weiteren möglich, andere Komponenten an die Plattform anzubinden, so dass sowohl Jadex-Agenten als auch andere Komponenten auf die Dienste der jeweils anderen Seite zugreifen können. Es muss allerdings erwähnt werden, dass die Implementierung in ihrem momentanen Zustand keine plattformübergreifende Kommunikation zulässt. Das liegt daran, dass die Nutzung von JMS die Bibliotheken des verwendeten JMS Providers voraussetzt, die unter anderem das herstellerabhängige Übertragungsprotokoll implementieren. Da sich der JMS Provider einer entfernten Plattform von der lokalen unterscheiden kann, müssten dessen Bibliotheken auf der lokalen Plattform vorhanden sein, damit diese Nachrichten an entfernte Agenten schicken kann. Dieses Defizit lässt sich aber, wie im nächsten Abschnitt erklärt wird, mit Hilfe der JCA ausgleichen. Insofern wurde die Zielsetzung dieser Arbeit – Interoperabilität mit bestehender Software und Laufzeit der Agenten innerhalb des Applikationsservers – erfüllt.

Mit der Implementierung eines auf EJBs basierenden Adapters wurde für Java EE Applikationsserver der Einsatz eines kognitiv mächtigen BDI-Kon-

zepts zur Entwicklung von asynchron kommunizierenden Softwaresystemen ermöglicht. Zwar lässt sich dieses auch mit JMS und TimerService realisieren, sie bieten aber keine weiteren Hilfsmittel zur Strukturierung eines dynamischen, nebenläufigen Systems wie es das Agentenparadigma im Allgemeinen und BDI im Speziellen vermögen. Mit AMS und DF können Agenten dagegen dynamisch erzeugt und von anderen Agenten aufgefunden werden.

Solange die in Kapitel 5.6 genannten Einschränkungen eingehalten werden, können Jadex-Agenten mit Hilfe der Standalone-Plattform entwickelt und erst anschließend auf einem Applikationsserver installiert werden. Da die Standalone-Plattform Werkzeuge zur Fehlersuche beinhaltet, würde sich die Entwicklung von Agenten auf dieser Plattform anbieten.

Die Beispielanwendung demonstriert zwar den Einsatz von Agenten innerhalb Applikationsservern, allerdings ließe sich diese Anwendung ebenso mit klassischen EJBs lösen. Die Frage, wann Jadex-Agenten klassischen EJBs vorzuziehen sind, hängt von der Umgebung und den Aufgaben ab, die das System erfüllen soll. Sie ist hauptsächlich in Szenarien sinnvoll, in denen längerfristige, personalisierte[38] Aufgaben in hochdynamischen Umgebungen ausgeführt werden. Beispiele für solche Szenarien sind von Anwendern gestellte Suchanfragen (etwa nach dem günstigsten Warenpreis), die von Rechercheagenten im Internet gelöst werden sollen. Die Aufgaben erfordern eine ausgedehnte Suche in verschiedenen Internetquellen. Neben der Anfrage von bekannten Quellen sollen nach Möglichkeit ebenso neue Quellen gefunden werden, um eine akzeptable Qualität der Suchergebnisse zu gewährleisten.

Da Message-driven Beans zustandslos sind und der TimerService ebenfalls nur von zustandslosen EJBs genutzt werden kann, müsste eine Anwendung die Zuordnung von Nachrichten und abgelaufenen Timern selbst implementieren, um eine personalisierte Aufgabe zu erfüllen.

6.2 Ausblick

Obwohl der Jadex-Adapter für Java EE Applikationsserver lauffähig ist, gibt es Erweiterungspotential, das im Rahmen dieser Arbeit nicht mehr ausgeschöpft werden konnte. Dies gilt beispielsweise für das in Kapitel 5.4.1 beschriebene Problem der Abbildung des Agentenzustands auf ein Datenbankschema. Die gegenwärtige Architektur von Jadex erschwert eine solche Abbildung, daher wurde für diese Arbeit mit der Objektserialisierung von Java ein einfacherer Mechanismus zur Speicherung der Agenten gewählt. Während durch die Serialisierung bedingt immer der gesamte, binär abgespeicherte Agentenzustand geladen und deserialisiert werden muss, bevor auf ihn zugegriffen werden kann, können bei einer Abbildung auf ein Datenbankschema auch nur Teile des Agentenzustands geladen und genutzt werden.

38 *Personalisierte Aufgaben* sind solche, die an eine Identität geknüpft sind, sich also identifizieren und zuordnen (z.B. zu einem Benutzer) lassen.

Neben dem offensichtlichen Vorteil eines optimierten Zugriffs auf die nur benötigten Teile des Zustands, hat eine Abbildung noch weitere Vorteile: Ein Schema kann weiterentwickelt werden. Das bedeutet, dass zukünftige Versionen des Adapters existierende Agentenzustände nutzen können, da das bestehende Datenbankschema bei Bedarf an die neue Version angepasst werden kann. Serialisierte Objekte können dagegen immer nur mit einer bestimmten Version einer Klasse genutzt werden.

Ein weiterer Vorteil einer Datenbankabbildung besteht darin, dass sie einen feiner abgestuften Zugriff auf den Agentenzustand ermöglicht. Durch die Serialisierung bedingt muss immer der gesamte Zustand geladen und transaktional gesperrt werden, auch wenn eventuell nur ein kleiner Teil des Zustands benötigt wird. Eine Abbildung würde die Sperrung nur der Teile des Agentenzustands bedeuten, die wirklich verwendet werden. Die daraus resultierende potentiell erhöhte Nebenläufigkeit der Agenten erlaubt die längerfristige Sperrung von Teilen des Zustands. So sind beispielsweise Agentenpläne möglich, die die genutzten Agentenressourcen (etwa Einträge in der Wissensbasis) transaktional sperren und die Änderungen bei erfolgreichem Planabschluss wirksam beziehungsweise bei einem Fehlschlagen des Plans zurückgesetzt werden. Bei dem Standard- und mobilen Plan obliegt es dem Entwickler, etwaig benutzte Agentenressourcen beim Abschluss eines Plans manuell zurückzusetzen.

Wie im letzten Abschnitt erklärt, kann über JMS nur der lokale Nachrichtenaustausch erfolgen. Die Java EE Spezifikation hat allerdings mit der JCA eine Schnittstelle geschaffen, über die Fremdprotokolle von EIS in einen Applikationsserver genutzt werden können. Nach [Sun03] ermöglicht die JCA EJBs und Servlets ausgehende Anfragen entweder unter Verwendung einer generischen Schnittstelle, Common Client Interface (CCI) genannt, oder über eine eigene, herstellerspezifische Schnittstelle. Welche der beiden Schnittstellen genutzt werden können, hängt vom Hersteller des *Resource Adapters* ab. Das CCI hat den Vorteil, dass sich ein Adapter dank der generischen Schnittstelle nachträglich durch einen anderen ersetzen lässt.

Die JCA spezifiziert über ausgehende Anfragen hinaus noch Schnittstellen für die Bearbeitung von eingehenden Anfragen. Dies erlaubt es über Fremdprotokolle zugängliche Dienste zu veröffentlichen. Entsprechende Anfragen werden ähnlich wie bei JMS an Message-driven Beans weitergeleitet, welche die Anfrage bearbeiten.

Die beiden genannten Mechanismen sind ausreichend für die Implementierung eines FIPA-konformen Resource Adapters. Eingehende Agentennachrichten würden dabei wie bei JMS an ein Message-driven Bean geleitet werden, dass den Empfänger der Nachricht ermittelt, den entsprechenden Agenten lädt und ihm die Nachricht zustellt. Ausgehende Nachrichten würden mit Hilfe des CCI verschickt. Der Resource Adapter würde eine Verbindung zur Zielplattform aufbauen und über diese die Nachricht übermitteln. Die Nutzung des CCI hätte darüber hinaus den Vorteil, dass es als generische Schnittstelle beliebig aus-

tauschbar ist. Es könnte also zur Laufzeit der Agentenplattform ein weiterer Resource Adapter installiert werden. Dessen Existenz müsste der Plattform allerdings über eine Verwaltungsschnittstelle bekannt gemacht werden, da die Plattform dies selbst nicht feststellen kann. Die Agentenplattform ließe sich demnach mit beliebigen Protokollen erweitern, um die Interoperabilität mit anderen Agentensystemen sicher zu stellen.

Wie sich in diesem Abschnitt gezeigt hat, bietet die in dieser Arbeit realisierte Integration noch Potential für Erweiterungen, die aufgrund der Komplexität der Themen (Architektur von Jadex und Sprechakt-basierte Agentenkommunikation) umfangreich genug für weitere Diplomarbeiten sind.

Offen bleiben auch weiterführende Untersuchungen, inwieweit Agententechnologie für andere unternehmensrelevante Softwarebereiche von Nutzen sein kann. Paul Taylor, Peter Evans-Greenwood und James Odell führen diesbezüglich das Beispiel des *Business Process Management (BPM)* an, bei dem es unter anderem um die Modellierung von Geschäftsprozessen geht [TaEO05]. Mit Hilfe formaler Beschreibungssprachen wie etwa BPEL (Business Process Execution Language) lassen sich reale Geschäftsprozesse zur Effizienzsteigerung auf Softwaresysteme abbilden [Zapl05]. Diese Abbildungen sind statisch. Zur Entwicklungszeit müssen deshalb alle möglichen Verzweigungen im Arbeitsablauf (engl. Workflow) berücksichtigen werden, da nicht berücksichtigte Abläufe einen manuellen Eingriff von einem Operator notwendig werden lassen. Nach [TaEO05] könnte Agententechnologie diesen Nachteil kompensieren, ermöglicht sie doch eine dynamische Anpassung an Geschäftsfälle, die bei der Modellierung nicht berücksichtigt wurden.

Abkürzungen

ACC	Application Client Container
ACL	Agent Communication Language
ADF	Agent Definition File
AMS	Agent Management System
AOP	Aspect-oriented Programming
API	Application Programming Interface
BDI	Belief, Desire, Intention
BMP	Bean-managed Persistence
BPEL	Business Process Execution Language
BPM	Business Process Management
CAL	Core Agent Layer
CCI	Common Client Interface
CMP	Container-managed Persistence
CORBA	Commen Object Request Broker Architecture
CTM	Component Transaction Monitor
DBMS	Datenbankmanagementsystemen
DCOM	Distributed Component Object Model
DF	Directory Facilitator
DNS	Domain Name Service
DOM	Document Object Model
DTP	Distributed Transaction Processing
EAI	Enterprise Application Integration
EIS	Enterprise Information Systems
EJB	Enterprise JavaBeans
EJB-QL	EJB Query Language
ERP	Enterprise Resource Planing-System
FIPA	Foundation for Intelligent Physical Agents
GIOP	General Inter-ORB Protocol
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IDL	Interface Description Language
IETF	Internet Engineering Task Force

IIOP	Internet Inter-ORB Protocol
J2EE	Java 2 Platform, Enterprise Edition
J2ME	Java 2 Platform, Micro Edition
J2SE	Java 2 Platform, Standard Edition
JADE	Java Agent Development Framework
Jadex	JADE Extension
Java EE	Java Platform, Enterprise Edition
JAXP	Java API for XML Processing
JAX-RPC	Java API for XML-based RPC
JAX-WS	Java API for XML Webservices
JCA	J2EE Connector Architecture
JCP	Java Community Process
JDBC	Java Database Connectivity
JEMS	JBoss Enterprise Middleware Suite
JMS	Java Message Service
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JSF	JavaServer Faces
JSP	JavaServer Pages
JSTL	JavaServer Pages Standard Tag Library
JTA	Java Transaction API
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
LS/TS	Living Systems Technology Suite
MAS	Multi-Agenten System
MBean	Managed Bean
MOM	Message-oriented Middleware
MTS	Microsoft Transaction Server
MVC	Model-View-Controller
NFS	Network File System
O/R Mapper	Objekt-relationale Mapper
OMG	Object Management Group
ONC	Open Network Computing
OOP	Objektorientierte Programmierung

OQL	Object Query Language
ORB	Object Request Broker
OTM	Object Transaction Monitor
POJO	Plain Old Java Object
RMI	Remote Method Invocation
RMI-IIOP	RMI über IIOP
RPC	Remote Procedure Call
SAX	Simple API for XML
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol / Internet Protocol
TILAB	Telecom Italia Lab
UML	Unified Modeling Language
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WAM	Werkzeug-Automat-Material
WSDL	Web Services Description Language
XSL	eXtensible Stylesheet Language

Abbildungsverzeichnis

Literaturverzeichnis

[ACK03] Gustavo Alonso, Fabio Casati und Harumi Kuno: *Web Services: Concepts, Architectures and Applications.* Springer, ISBN 3-540-44008-9, 2003.

[AgIn03a] Agentis International, Inc.: *Simplifying the Complexity of Application Development.* http://www.agentissoftware.com/en/resources/downloads/whitepapers/AgentisWhitePaper.pdf, 2003.

[AgIn03b] Agentis International, Inc.: *Goal-Directed Agent Technology.* http://www.agentissoftware.com/en/resources/downloads/whitepapers/GoalDirectedAgentTechnologyWP.pdf, 2003.

[AgIn05] Agentis International, Inc.: *Simplifying Process Integration across Enterprise Applications and Data.* http://www.agentissoftware.com/en/resources/downloads/whitepapers/CompositeApplicationsWP.pdf, 2005.

[BaGa02] Martin Backschat und Otto Gardon: *Enterprise JavaBeans: Grundlagen - Konzepte - Praxis.* Spektrum, Akademischer Verlag, ISBN 3-8274-1322-2, 2002.

[BCEHJ06] Jennifer Ball, Debbie Bode Carson, Ian Evans, Kim Haase und Eric Jendrock: *The Java EE 5 Tutorial.* http://java.sun.com/javaee/5/docs/tutorial/doc/JavaEETutorial.pdf, Stand: 18.02.2006.

[BCPR03] F. Bellifemine, G. Caire, A. Poggi, G. Rimassa: *JADE – A White Paper.* http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf, September 2003.

[BHKPB03] Sara Bouchenak, Daniel Hagimont, Sacha Krakowiak, Noel De Palma und Fabienne Boyer: *Experiences implementing efficient Java thread serialization, mobility and persistence.* In: *Software: Practice and Experience, Band 33.* http://sardes.inrialpes.fr/papers/files/04-Bouchenak-SPE.pdf, September 2003.

[BrHa02] Stefan Brantschen und Thomas Haas: *Agents in a J2EE World.* In: *AgentLink News (9)*, S. 15-19. http://www.agentlink.org/newsletter/9/AL-9.pdf, 2002.

[CaLy03] Michelle Casagni und Margaret Lyell: *Comparison of Two Component Frameworks: The FIPA-Compliant Multi-Agent System and The Web-Centric J2EE Platform.* In: *Proceedings of the 25th International Conference on Software Engineering*, S. 341-351. IEEE Computer Society, ISBN 0-7695-1877-X, 2003.

[CGBKR02] Dick Cowan, Martin Griss, Bernard Burg, Robert Kessler und Brian Remick: *A Robust Environment for Agent Deployment.* http://www.agentcities.org/Challenge02/Proc/Papers/ch02_44_cowan.pdf, 2002.

[CoGB02] Dick Cowan, Martin Griss und Bernard Burg: *BlueJADE – A service for managing software agents.* http://www.hpl.hp.com/techreports/2001/HPL-2001-296R1.pdf, 2002.

[CoGr02] Dick Cowan und Martin Griss: *Making Software Agent Technology available to Enterprise Applications.* http://www.hpl.hp.com/techreports/2002/HPL-2002-211.pdf, 2002.

[DeKe05a] Linda DeMichiel und Michael Keith: *JSR 220: EJB 3.0 Simplified API.* http://jcp.org/en/jsr/detail?id=220, Stand: 18.12.2005.

[DeKe05b] Linda DeMichiel und Michael Keith: *JSR 220: Java Persistence API.* http://jcp.org/en/jsr/detail?id=220, Stand: 18.12.2005.

[DeKe05c] Linda DeMichiel und Michael Keith: *JSR 220: EJB Core Contracts and Requirements.* http://jcp.org/en/jsr/detail?id=220, Stand: 18.12.2005.

[Dijk59] Edsger. W. Dijkstra: *A note on two problems in connexion with graphs.* In: *Numerische Mathematik (Band 1)*, S. 269-271. Springer, 1959.

[EiMe05] Michael Eichhorst und Mira Mezini: *Alice: Modularization of MiddlewareUsing Aspect-Oriented Programming.* In: *Thomas Gschwind und Cecilia Mascolo: Software Engineering and Middleware; 4th International Workshop 2004*, S. 47-63. Springer, ISBN 3-540-25328-9, 2005.

[FrGr96] Stan Franklin, Art Graesser: *Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents.* In: *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*. Springer, 1996.

[GaHJ97] Erich Gamma, Richard Helm und Ralph E. Johnson: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, ISBN 0201633612, 1997.

[GHJV97] Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, ISBN 0-201-63361-2, 1997.

[GJSB05] James Gosling, Bill Joy, Guy Steele, Gilad Bracha: *The Java Language Specification, Third Edition.* Addison-Wesley, ISBN 0-321246-78-0, Juni 2005.

[GPPTW99] M. Georgeff, B. Pell, M. Pollack, M. Tambe und M. Wooldridge: *The Belief-Desire-Intention Model of Agency.* In: *J. P. Muller, M. Singh, and A. Rao, Hrsg. Intelligent Agents V*. Springer, `http://www.csc.liv.ac.uk/~mjw/pubs/atal98b.pdf`, März 1999.

[GrTh00] Volker Gruhn und Andreas Thiel: *Komponentenmodelle: DCOM, Javabeans, Enterprise Java Beans, CORBA.* Addison-Wesley, ISBN 3-827-31724, 2000.

[Grub93] Thomas R. Gruber: *A translation approach to portable ontologies.* In: *Knowledge Acquisition (Band 5, Nummer 2)*, S. 199-220. `http://tomgruber.org/writing/ontolingua-kaj-1993.pdf`, 1993.

[Hack98] Winfried Hacker: *Allgemeine Arbeitspsychologie: Psychische Regulation von Arbeitstätigkeiten (1. Auflage).* Huber, ISBN 3-456-82917-5, 1998.

[HaRa01] Theo Härder und Erhard Rahm: *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer, ISBN 3-540-42133-5, 2001.

[Harb04] Mathias Harbeck: *Diplomarbeit zum Thema: BDI-Agentensysteme auf mobilen Geräten.* Universität Hamburg, Fachbereich Informatik, Verteilte Systeme und Informationssysteme, `http://vsis-www.informatik.uni-hamburg.de/getDoc.php/thesis/162/harbeck_da.pdf`, 2004.

[HBSFS02] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli und Kate Stout: *Java Message Service Specification (Version 1.1).* `http://java.sun.com/products/jms/docs.html`, Stand: 12.04.2002.

[JBoss05] The JBoss Group: *JBoss 4.0 - The Official Guide.* Sams, ISBN 0672326485, 2005.

[John02] Rod Johnson: *Expert One-on-one J2EE Design and Development.* Wrox Press, ISBN 0-7645-4385-7, 2002.

[Kühn01] Ralf Kühnel: *Agentenbasierte Software: Methode und Anwendungen.* Addison-Wesley, ISBN 3-8273-1739-8, 2001.

[KuWö02] Michael Kuschke und Ludger Wölfel: *Web Services kompakt.* Spektrum Akademischer Verlag, ISBN 3-8274-1375-3, 2002.

[Mos03] Marco Mosconi: *Diplomarbeit zum Thema: Modularisierung und Adaptierung von Komponenteninteraktionen mit Object Teams und dem CORBA Komponentenmodell.* Technische Universität Berlin, `http://www.objectteams.org/publications/Diplom_Marco_Mosconi.pdf`, 16. Mai 2003.

[MuJo00] Richard Murch und Tony Johnson: *Agententechnologie: Die Einführung – Intelligente Software-Agenten auf Informationssuche im Internet.* Addison-Wesley, ISBN 3-8273-1652-9, 2000.

[NiNy05] Jaran Nilsen und Anders Nygaard: *Bachelor Thesis: Managing Agents in Application Servers.* `http://agents.jaranweb.com/project/forprosjekt.pdf`, 04.03.2005.

[Pan04] Debu Panda: *Simplifying EJB Development with EJB 3.0.* `http://www.theserverside.com/articles/article.tss?l=SimplifyingEJB3`, Oktober 2004.

[PBL03] Alexander Pokahr, Lars Braubach und Winfried Lamersdorf: *Jadex: Implementing a BDI-Infrastructure for JADE Agents.* In: *EXP - In Search of Innovation, Band 3, Nr. 3*, S. 76-85. Telecom Italia Lab, Turin, Italien, September 2003.

[PBL05a] Alexander Pokahr, Lars Braubach und Winfried Lamersdorf: *Jadex: A BDI Agent System Combining Middleware and Reasoning.* In: *R. Unland, M. Calisti, M. Klusch (Hrsg.), Software Agent-Based Applications, Platforms and Development Kits*, S. 143-168. Birkhäuser, ISBN 3-7643-7347-4, 2005.

[PBL05b] Alexander Pokahr, Lars Braubach und Winfried Lamersdorf: *Jadex: A BDI Reasoning Engine.* In: *R. Bordini, M. Dastani, J. Dix and A. Seghrouchni (Hrsg.), Multi-Agent Programming*. Kluwer Book, 2005.

[PBL05c] Alexander Pokahr, Lars Braubach und Winfried Lamersdorf: *A Goal Deliberation Strategy for BDI Agent Systems.* In: *Third German conference on Multi-Agent System TEchnologieS (MATES-2005)*. Springer, 2005.

[PuRP06] Arno Puder, Kay Römer und Frank Pilhofer: *Distributed Systems Architecture.* Morgan Kaufmann, ISBN 1-55860-648-3, 2006.

[RiCK05] Giovanni Rimassa, Monique Calisti und Martin E. Kernland: *Living Systems Technology Suite.* In: *R. Unland, M. Calisti, M. Klusch (Hrsg.), Software Agent-Based Applications, Platforms and Development Kits*, S. 73-93. Birkhäuser, ISBN 3-7643-7347-4, 2005.

[Rim04] Giovanni Rimassa: *JADE Persistence Add-on.* `http://jade.tilab.com/dl.php?file=JADEPersistenceAddOn-3.3.zip`, Stand: 28.02.2004.

[Scha04] Bernhard Schäfers: *Sozialstruktur und sozialer Wandel in Deutschland.* UTB, ISBN 3-8252-2186-5, 2004.

[Star04] Thomas Stark: *J2EE – Einstieg für Anspruchsvolle.* Addison-Wesley, ISBN 3-8273-2184-0, 2004.

[Sun00] Sun Microsystems: *J2ME Building Blocks for Mobile Devices; White Paper on KVM and the Connected, Limited Device Configuration (CLDC).* `http://java.sun.com/products/cldc/wp/KVMwp.pdf`, 2000.

[Sun03] Sun Microsystems: *J2EE Connector Architecture Specification (Version 1.5).* `http://java.sun.com/j2ee/connector/download.html`, 2003.

[Szyp98] Clemens Szyperski: *Component Software – Beyond Object-Oriented Programming.* Addison-Wesley, ISBN 0-201-17888-5, 1998.

[TaEO05] Paul Taylor, Peter Evans-Greenwood und James Odell: *Agents in the Enterprise.* `http://evans-greenwood.com/peter/papers/aswec-2005-agents-in-the-enterprise.pdf`, 2005.

[Volp94] Walter Volpert: *Wider die Maschinenmodelle des Handelns; Aufsätze zur Handlungsregulationstheorie.* Wolfgang Pabst Verlag, ISBN 3-928057-58-8, 1994.

[WikiBDI06] Wikipedia: *BDI-Agenten-Architekturen.* `http://de.wikipedia.org/wiki/BDI-Agenten-Architekturen`, Stand: 10.02.2006.

[WITG04] Whitestein Information Technology Group AG: *LS/TS: Agent Modeling Language Specification (Version 0.9).* 2004.

[WITG05a] Whitestein Information Technology Group AG: *LS/TS: Core Agent Layer (Version 1.1).* 2005.

[WITG05b] Whitestein Information Technology Group AG: *LS/TS: Run-Time Enviroment Overview (Version 1.1).* 2005.

[WoJe95] Michael Wooldridge, Nicholas R. Jennings: *Intelligent Agents: Theory and Practise.* In: *The Knowledge Engineering Review, Band 10 (2), revidierte Version*, S. 115-152. http://www.agent.ai/doc/upload/200302/wool95.pdf, 1995.

[Wool02] Michael J. Wooldridge: *An Introduction to MultiAgent Systems.* John Wiley and Sons Ltd, ISBN 0-471-49691-X, 2002.

[Wool96] M. Wooldridge: *Practical Reasoning with Procedural Knowledge: A Logic of BDI Agents with Know-How.* In: *D. M. Gabbay and H.-J. Ohlbach (Hrsg.), Proceedings of the International Conference on Formal and Applied Practical Reasoning*. Springer, http://www.csc.liv.ac.uk/~mjw/pubs/fapr96-bdi.pdf, Juni 1996.

[Yuan05] Michael Yuan: *POJO Application Frameworks: Spring vs. EJB 3.0.* http://www.onjava.com/pub/a/onjava/2005/06/29/spring-ejb3.html, 29.06.2005.

[Zapl05] Sonja Zaplata: *Diplomarbeit zum Thema: Prozessintegration in Middleware für mobile Systeme.* Universität Hamburg, http://vsis-www.informatik.uni-hamburg.de/getDoc.php/thesis/305/Diplomarbeit_ZA_041005.pdf, 04.10.2005.

Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit der Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den 25.04.2006