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


Inhaltsverzeichnis

1 Einleitung .................................................. 1
  1.1 Motivation.............................................. 2
  1.2 Zielsetzung............................................ 3
  1.3 Gliederung............................................ 5

2 Agententechnologie ................................. 7
  2.1 Hintergrund.......................................... 8
  2.2 Eigenschaften von Agenten.......................... 8
  2.3 BDI-Modell.......................................... 9
  2.4 BDI-Plattformen.................................... 12
  2.5 Jadex.................................................. 13
    2.5.1 Agent Definition File.......................... 14
    2.5.2 Agentenmodell................................ 18
    2.5.3 Agenda.......................................... 19
    2.5.4 Integration über Adapter...................... 19
  2.6 Zusammenfassung.................................... 21

3 Middleware .............................................. 23
  3.1 Hintergrund.......................................... 24
  3.2 Anforderungen an Geschäftsanwendungen........... 24
  3.3 Komponentenbasierte Softwareentwicklung...... 25
  3.4 Middleware-Technologien............................ 27
    3.4.1 Remote Procedure Call........................... 27
    3.4.2 Object Request Broker........................... 27
    3.4.3 Message-oriented Middleware................... 29
    3.4.4 Transaktionsmonitore............................. 31
    3.4.5 Applikationsserver............................... 32
  3.5 Java Platform, Enterprise Edition................. 33
    3.5.1 3-Schichten Architektur........................ 33
    3.5.2 Bestandteile der Spezifikation................... 35
    3.5.3 Enterprise JavaBeans 2.1..................... 37
    3.5.4 Enterprise JavaBeans 3.0.................... 39
    3.5.5 Java Message Service........................... 41
  3.6 Spring Framework.................................. 43
  3.7 Zusammenfassung.................................... 44
4 Analyse der Integration ................................................. 45
  4.1 Anforderungen von Agenten .................................. 46
  4.2 Integrationsansätze .............................................. 47
  4.3 Containerbasierte Integration .................................. 48
    4.3.1 BlueJADE ................................................ 48
    4.3.2 Jadex ..................................................... 49
    4.3.3 Realisierung für Jadex .................................. 50
  4.4 Komponentenbasierte Integration ............................. 51
    4.4.1 Restriktionen für Enterprise JavaBeans .............. 51
    4.4.2 Living Systems Technology Suite ..................... 52
    4.4.3 Adaptive Enterprise Solution Suite ................. 54
    4.4.4 Realisierung für Jadex ................................ 56
  4.5 Zusammenfassung ............................................... 59

5 Implementierung eines Integrationsansatzes ............... 61
  5.1 Auswahl und Begründung ...................................... 62
  5.2 Architektur der Integration .................................... 63
  5.3 Bestandteile der Implementierung ............................ 65
    5.3.1 AdapterEntity ........................................... 65
    5.3.2 TimerDispatcher ........................................ 67
    5.3.3 MessageDispatcher ...................................... 68
    5.3.4 AMS, DF und Planbibliothek ............................ 69
    5.3.5 Webinterface ............................................ 70
  5.4 Aufgetretene Probleme ......................................... 70
    5.4.1 Abbildung auf ein Datenbankschema .................. 71
    5.4.2 Synchronisierter Agentenzugriff ...................... 74
  5.5 Evaluation anhand einer Beispielanwendung .............. 76
    5.5.1 Szenariobeschreibung .................................. 77
    5.5.2 Umsetzung ............................................... 78
  5.6 Zusammenfassung ............................................... 81

6 Schlussbetrachtung .................................................. 83
  6.1 Zusammenfassung ............................................... 84
  6.2 Ausblick ....................................................... 85

Abkürzungen ............................................................. 89
Abbildungsverzeichnis .................................................. 93
Literaturverzeichnis ..................................................... 95
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.

1.1 Motivation


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-
Zielsetzung


1.2 Zielsetzung


---

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


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


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.


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.
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.

2.1 Hintergrund


2.2 Eigenschaften von Agenten

Ausgehend von der bisherigen Definition eines Agenten⁴ 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 für menschlichen Agenten zurückgreifen und von ihrer Software kurzentschieden 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:


- **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.


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-

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 wahrgenommenen Informationen aus seiner Umwelt für seine Entscheidungen relevant sind. Darüber hinaus kann sein Wissen über die Umwelt veraltet und damit nicht mehr korrekt sein.


• **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.
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**


\(^5\) http://www.nuin.org/

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.

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 ermöglicht.

2.5.1 Agent Definition File

Jadex Agenten bestehen aus einem in einer XML-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]):

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.

Abb. 5: Aufbau des ADF für Jadex Agenten
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) zugreifen werden kann. Dazu kann man einfache Beliefs und Beliefssets 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.
- Beliefssets können mehrere Objekte eines Typs speichern. Entsprechend der Mengenlehre kann ein Objekt dabei immer nur einmal in einem Beliefsset vorkommen, wobei die Gleichheit zweier Objekte mittels equals() geprüft wird und ein bestehendes Objekt bei Gleichheit von dem Neuen aus dem Beliefsset 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.

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:


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-

**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:


- **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**
Properties


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.\textsuperscript{7}


Auch wenn Agentenmodelle und Klassen zur Laufzeit normalerweise statische Konstrukte darstellen, bieten doch beide die Möglichkeit, ihre Struktur zur Laufzeit zu ändern.\textsuperscript{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.

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

\textsuperscript{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


2.5.4 Integration über Adapter


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

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

- `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:
Zusammenfassung

- 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 ihm betreffenden von der Plattform verwendeten Ressourcen wieder freizugeben. Dieses geschieht nur im Fall, dass der Agent beendet werden möchte beziehungsweise beendet werden soll.


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.

3.1 Hintergrund


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 Anforderungen an Anwendungen, die in Geschäftsanwendungen 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 Programmiersprachen 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-
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.

- **Performance** (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.


### 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)


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


Ein Komponentenmodell legt den Rahmen für die Entwicklung und Ausführung von Komponenten fest, der strukturelle Anforderungen hinsichtlich Verknüpfungs- beziehungsweise Kompositions möglichkeiten 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 Entwicklung die Erweiterbarkeit der nichtfunktionalen Dienste nicht gewährleistet. Üblicherweise legt man sich mit der Wahl eines Komponentenmodells auf dieses und

die von ihm erbrachten, nichtfunktionalen Dienste fest. AOP ermöglicht da-
gegen 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-Technolo-
gien 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 syn-
chronen Aufruf von Programmfunktionen über Rechnergrenzen hinweg, die 
neben den TP-Monitoren (siehe Kapitel 3.4.4) eine der ältesten Middlewa-
re-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 Protokolle dar, das nach 
dem Client-Server Modell transparentes, verteiltes Rechnen ermöglicht. Um 
die Schnittstelle eines Dienstes unabhängig von der eingesetzten Program-
miersprache veröffentlichen zu können, musste diese mit einer deklarativen 
Schnittstellenbeschreibungssprache – Interface Description Language (IDL) 
genannt – beschrieben werden. Neben dem von Sun entwickelten ONC RPC 
tstanden noch einige andere zueinander inkompatible RPC Protokolle.

3.4.2 Object Request Broker

Object Request Broker (ORB) erweitern das RPC-Konzept von verteilten Pro-
zedurauffrufen um die Möglichkeit einer Kommunikation zwischen verteilten 
Objekten. Die OMG (Object Management Group) entwickelte 1991 den ers-
ten international anerkannten Standard für einen ORB: Die so genannte Com-
men Object Request Broker Architecture (CORBA) legt alle für den plattform-
und sprachunabhängigen Einsatz von verteilten Objekten erforderlichen De-
tails 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\textsuperscript{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]).


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

\textsuperscript{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.

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\(^\text{15}\) und die bisherige Zustandslosigkeit eines Web Services.\(^\text{16}\) So kommt es, dass trotz weitreichender 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 eben- falls 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.

\[\text{Abb. 8: Synchrone Kommunikation}\]

\(^{15}\) 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 zurück schickt, kann der Aufrufer die Zeit bis zum Eintreffen der Antwort für weitere Berechnungen nutzen, 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 Informationsaustausches 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).

### 3.4.4 Transaktionsmonitore


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 Zugriffsschutzverantwortung, 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\(^\text{19}\), 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.


3.4.5 Applikationsserver


- **Verwaltung der installierten Komponenten**: Der Applikationsserver muss die von den Benutzern angeforderten Komponenten instanzieren 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 Yuhana20, 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.


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.

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 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 Teilspekte der gesamten Laufzeitumgebung spezifizieren, APIs festlegen und teilweise auch Test-Suiten (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.

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.


- **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.


• **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 Adapters 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.


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.


\(^{21}\) XDDoclet vereinfacht die Entwicklung von EJB dahingehend, dass im einfachsten Fall nur noch die Beanklasse selbst erstellt werden muss. Alle anderen Bestandteile würde XDDoclet erzeugen (http://xdoclet.sourceforge.net/xdoclet/index.html).
• Das **Komponenten-Interface** deklariert die eigentlichen Geschäfts- 
thoden, 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 da-
gegen einen netzweiten Zugang zum Bean (über RMI-IIOP). Die Un-
terscheidung zwischen den beiden Interfaces ist notwendig, da die loka-
le 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 ge-
schieht über Session Beans, die als Fassade [GHJV97] für die Entity Be-
ans 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 Kon-
figurationsdatei, die dem Container alle nötigen Information zur Instal-
lation (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 wur-
de, ist die Entwicklung von Serverkomponenten nach J2EE 1.4 sehr aufwen-
dig. 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 Me-
thodennamen per Konvention mit denen der Beanklasse überein. Über den 
Deployment-Deskriptor werden alle Abhängigkeiten zwischen den EJBs fest-
gehalten. 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 Instanz-
variablen mit so genannten Annotations (dt. Anmerkungen, Notizen) zu ver-
sehen, wird in EJB 3.0 genutzt, um die vormals im Deployment-Deskriptor 
aufgeführten Konfigurationseinstellungen der Komponenten direkt im Quell-
code mitzuführen. So können zum Beispiel Transaktionsattribute für Session 
Bean-Methoden oder Datenbankabbildungen der persistenten Objekte direkt 
an den Variablen vorgenommen werden. Dies reduziert den Entwicklungsauf-
 wand, weil die Erstellung und Pflege von Deployment-Deskriptoren und ex-
ternen Mapping-Dateien überflüssig wird. Da die Einstellungen im Quelltext
festgeschrieben sind und ohne Neukompilierung nicht geändert werden kön-
nen, wurden die Deployment-Deskriptoren nicht ganz abgeschafft. Sie ermög-
llichen vielmehr Änderungen an den im Quelltext hart kodierten Einstellungen
vorzunehmen, um beispielsweise eine Anwendung an ihre Umgebung anzu-
passen

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 ausge-
führt wird. Sollte der Aufruf schon innerhalb einer Transaktion stattfinden,
will 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 Re-
ferenzen zum Beispiel auf andere EJBs oder den Entitymanager (s.u.) zu ver-
sehen, so dass das Bean die Referenzen direkt nutzen kann, ohne sich die Re-
ferenz 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 ge-
worden. 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-Spezifika-
tion ihre Komponenten-Interfaces implementieren wie man es von der objek-
torientierten Entwicklung gewohnt ist.

Das Persistenzmodell wurde komplett neu entwickelt. Geschäftsdaten können
nu 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 Ab-
bildung auf Datenbankfelder zu bestimmen. Der ehemals über Home-Interfa-
ces realisierte Zugriff auf Entity Beans wird nun über eine zentrale Instanz,
den Entitymanager, geregelt. Über ihn können Geschäftsdaten 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 einge-
führt. Sie ermöglichen es dem Entwickler, die schon vorhandenen nichtfunk-
tionalen Dienste des Containers um eigene Aspekte zu erweitern. So können
EJBs mit einer Annotation für Interceptors versehen werden. Der Methoden-
aufruf 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


- **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]](image)

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.


Eine JMS Nachricht besteht immer aus den folgenden drei Bestandteilen:

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.

3.6 Spring Framework


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.


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 Middlewaretechniken 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

4.1 Anforderungen von Agenten


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.


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:

- 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-
Analyse der Integration

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

4.3 Containerbasierte Integration


4.3.1 BlueJADE

Ein Vertreter dieser Art von Integration ist BlueJADE\textsuperscript{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\textsuperscript{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 Mikrokern erreicht, der über die Java Management Extensions (JMX) API\textsuperscript{24} verwaltet und dessen Funktionalität über zusätzliche, ansteckbare Dienste erweitert werden kann (vgl. [JBoss05]).


\textsuperscript{22} http://sourceforge.net/projects/bluejade/
\textsuperscript{23} http://labs.jboss.com/portal/jbossas/
\textsuperscript{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]).

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


---

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](http://jademx.sourceforge.net)
weiterten Zugriffs können somit auch JUnit-Tests\textsuperscript{27} für Jadex-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 Jadex 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.
- Innerhalb von J2SE 5.0 lässt sich Jadex ohne Applikationsserver betreiben.

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


\textsuperscript{27}JUnit ist ein Java Framework zum Testen von Einheiten (engl. Units), welches zumeist Java Klassen sind.
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 Synchronisierungsprimitive nur innerhalb einer JVM wirken und die Synchronisierung in einem Cluster damit nicht möglich wäre.
- 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.
- 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 *Development Suite* umfasst eine Sammlung von Werkzeugen, die Modellierung, Entwicklung und Testen von Agenten erleichtern. Ebenso gibt es
Komponentenbasierte Integration 53

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 Development 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]


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 Development 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.


4.4.3 Adaptive Enterprise Solution Suite


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:


- 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.

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


Kommunikation


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.


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:


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 neu. 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 aufrufen. 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-
Zusammenfassung

zuständigen 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 Dienstes 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 Operationen 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

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.

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) nahe liegt.

Beider 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, 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


Abb. 19: Architektur der gewählten komponentenbasierten Integration von Jadex


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 `IJadexAgent` 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.
- 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 TimerDispatchers.
- $isAgentThread(Thread)$ 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.
Bestandteile der Implementierung

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


![Sequenzdiagramm des wechselseitigen Zugriffs von Adapter und Agent](image)


### 5.3.2 TimerDispatcher

Aufgabe des Timer Dispatchers 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 implementiert) 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 eintreffenden 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.


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


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).


Alternativ dazu hat der Autor analog zu der AMS und DF Planbibliothek der Standalone-Plattform eine Reihe von \textit{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-

5.3.5 Webinterface


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-
Aufgetretene Probleme

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


5.4.1 Abbildung auf ein Datenbankschema


Fehlende Trennung von fachlichen Werten und Logik


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
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-Da-
tei abgelegt. EJB 3.0 Entities verwenden stattdessen Annotiations, um diese 
Konfigurationsinformationen direkt in den Klassen abzulegen.

Wie sich bei der Analyse von Jadex zeigte, gibt es keine Trennung von fach-
lichen Werten und Logik. Die erkennbaren Strukturen zeigen stattdessen eine 
saubere Trennung zwischen Agentenmodell (siehe Kapitel 2.5.2 ) und instan-
zierten Modell auf. Innerhalb des instanzierten 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 wer-
den müssen, wobei für jede Instanzvariable einer Klasse entschieden werden 
müsste, ob sie zustandsrelevante Informationen oder nur temporäre Laufzeit-
informationen enthält.

Zyklische Abhängigkeiten

Mit Hilfe des Software-Tomographen34 (Sotograph genannt) – einem Werk-
zug der Software-Tomography GmbH zur Analyse der Qualität von Archi-
tekturen und Quelltexten beliebiger Anwendungen – hat der Autor feststellen 
können, dass große Teile des Objektgraphen zyklisch abhängig sind. Abbil-
dung 23 kann diese Abhängigkeiten nur ansatzweise verdeutlichen, indem sie 
die Zyklen auf Package-Ebene für jadex.runtime darstellt. Schwarze Li-
nien 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. Ei-
ne 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 inso-
fem, als dass es im Umkehrschluss weniger unabhängigen Klassen in dem Ob-
jektgraphen gibt, die leichter zu verstehen sind, da eben diese Unabhängigkeit 
Wechselwirkungen mit anderen Klassen unterbindet.

Aufgetretene Probleme

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.


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

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 leistungsfähig 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 deterministischem 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 nebeneinanderliegender 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-

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 Agentenname 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 Annotationen @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 nebeneinläufiger 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


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 Streckeninformations für dieses Szenario zu verwenden, da die Daten der Betreiber – selbst wenn sie über eine offen gelegte Dienst Schnittstelle (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/](http://reiseauskunft.bahn.de/bin/query.exe/dn/)
37 [http://www.geofox.de](http://www.geofox.de)
Agentensystem anbietet, wurden die Betreiber als Agenten modelliert, deren Dienste ebenfalls einheitlich angesprochen werden können.

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 \([\text{Betreibernummer}, \text{Reisekosten}, \text{Reisedauer}]\) 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 \textit{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 \textit{TravelAssistant} umgesetzt. Er nimmt Anfragen vom Benutzer entgegen und startet bei jeder Anfrage mit Hilfe des AMS einen Arbeiter (\textit{Worker}), der die Anfrage abarbeitet. Dazu übergibt er ihm die Benutzervorgaben und den Namen des Anfragenden, damit auf die Anfrage geantwortet werden kann.

Ein \textit{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 \textit{TravelCompanies} und beginnt dann sukzessiv, eine Route zwischen Start- und Zielort unter Beachtung der Benutzervorgaben zu suchen. Dazu wird eine für mehrdimensionale Kanten gewichtungen 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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{diagram26.png}
\caption{Petrinetzdarstellung des Dijkstra-Algorithmus}
\end{figure}

Zu Beginn dieses iterativen Algorithmus befinden sich wie aus Abbildung 26 ersichtlich alle Orte in der Stelle \textit{locations}. Mit einer initialen Transition wird der Startpunkt der Anfrage in die Menge der bekannten Orte (\textit{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.


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


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 beein-
trächtigen aber nicht die Grundfunktionalität der implementierten Plattform,
wie Nachrichtenaustausch und Scheduling der Agenten.

Bei der Programmierung von Agenten müssen allerdings eine Reihe von Ein-
schränkungen hingenommen werden, um den Anforderungen eines Applikati-
onsservers an seine Anwendungskomponenten gerecht zu werden. Diese sol-
len an dieser Stelle zusammengefasst werden:

- Aufgrund der Restriktion von EJB bezüglich der Threadmanipulation
dürfen Jadex-Agenten nur mobile Pläne innerhalb eines Applikations-
  servers 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 Lauf-
  zeit von EJBs beeinflussende Mittel ist ebenfalls per Restriktion unter-
  sagt.
- Da Applikationsserver häufig nur in einer Konsolenumgebung laufen, ist
  die direkte Nutzung grafischer Benutzerschnittstellen im Plancode eben-
  falls verboten. Die Interaktion mit den Agenten kann nur über Nachrichten
  erfolgen.
- Die Verwendung statischer Variablen ist eingeschränkt möglich, aller-
  dings muss dann auf die Fähigkeit zur Clusterbildung des Applikations-
  servers verzichtet werden.

Die oben genannten Einschränkungen reduzieren die Einsatzgebiete der Agen-
ten 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 dispatch-
SubgoalAndWait() oder sendMessageAndWait() die Methode nach
Erfüllung der Wartebedingung von neuem ausgeführt, so dass größere und da-
mit unelegante Verzweigungsblöcke notwendig werden.
Schlussbetrachtung


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.


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


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


---

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


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.

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 Common 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
<table>
<thead>
<tr>
<th>Abkürzung</th>
<th>Bedeutung</th>
</tr>
</thead>
<tbody>
<tr>
<td>IIOP</td>
<td>Internet Inter-ORB Protocol</td>
</tr>
<tr>
<td>J2EE</td>
<td>Java 2 Platform, Enterprise Edition</td>
</tr>
<tr>
<td>J2ME</td>
<td>Java 2 Platform, Micro Edition</td>
</tr>
<tr>
<td>J2SE</td>
<td>Java 2 Platform, Standard Edition</td>
</tr>
<tr>
<td>JADE</td>
<td>Java Agent Development Framework</td>
</tr>
<tr>
<td>Jadex</td>
<td>JADE Extension</td>
</tr>
<tr>
<td>Java EE</td>
<td>Java Platform, Enterprise Edition</td>
</tr>
<tr>
<td>JAXP</td>
<td>Java API for XML Processing</td>
</tr>
<tr>
<td>JAX-RPC</td>
<td>Java API for XML-based RPC</td>
</tr>
<tr>
<td>JAX-WS</td>
<td>Java API for XML Webservices</td>
</tr>
<tr>
<td>JCA</td>
<td>J2EE Connector Architecture</td>
</tr>
<tr>
<td>JCP</td>
<td>Java Community Process</td>
</tr>
<tr>
<td>JDBC</td>
<td>Java Database Connectivity</td>
</tr>
<tr>
<td>JEMS</td>
<td>JBoss Enterprise Middleware Suite</td>
</tr>
<tr>
<td>JMS</td>
<td>Java Message Service</td>
</tr>
<tr>
<td>JMX</td>
<td>Java Management Extensions</td>
</tr>
<tr>
<td>JNDI</td>
<td>Java Naming and Directory Interface</td>
</tr>
<tr>
<td>JSF</td>
<td>JavaServer Faces</td>
</tr>
<tr>
<td>JSP</td>
<td>JavaServer Pages</td>
</tr>
<tr>
<td>JSTL</td>
<td>JavaServer Pages Standard Tag Library</td>
</tr>
<tr>
<td>JTA</td>
<td>Java Transaction API</td>
</tr>
<tr>
<td>JVM</td>
<td>Java Virtual Machine</td>
</tr>
<tr>
<td>LDAP</td>
<td>Lightweight Directory Access Protocol</td>
</tr>
<tr>
<td>LS/TS</td>
<td>Living Systems Technology Suite</td>
</tr>
<tr>
<td>MAS</td>
<td>Multi-Agenten System</td>
</tr>
<tr>
<td>MBean</td>
<td>Managed Bean</td>
</tr>
<tr>
<td>MOM</td>
<td>Message-oriented Middleware</td>
</tr>
<tr>
<td>MTS</td>
<td>Microsoft Transaction Server</td>
</tr>
<tr>
<td>MVC</td>
<td>Model-View-Controller</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
</tr>
<tr>
<td>O/R Mapper</td>
<td>Objekt-relationale Mapper</td>
</tr>
<tr>
<td>OMG</td>
<td>Object Management Group</td>
</tr>
<tr>
<td>ONC</td>
<td>Open Network Computing</td>
</tr>
<tr>
<td>OOP</td>
<td>Objektorientierte Programmierung</td>
</tr>
<tr>
<td>Abkürzung</td>
<td>Erklärung</td>
</tr>
<tr>
<td>-----------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>OQL</td>
<td>Object Query Language</td>
</tr>
<tr>
<td>ORB</td>
<td>Object Request Broker</td>
</tr>
<tr>
<td>OTM</td>
<td>Object Transaction Monitor</td>
</tr>
<tr>
<td>POJO</td>
<td>Plain Old Java Object</td>
</tr>
<tr>
<td>RMI</td>
<td>Remote Method Invocation</td>
</tr>
<tr>
<td>RMI-IIOP</td>
<td>RMI über IIOP</td>
</tr>
<tr>
<td>RPC</td>
<td>Remote Procedure Call</td>
</tr>
<tr>
<td>SAX</td>
<td>Simple API for XML</td>
</tr>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured Query Language</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol / Internet Protocol</td>
</tr>
<tr>
<td>TILAB</td>
<td>Telecom Italia Lab</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>WAM</td>
<td>Werkzeug-Automat-Material</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>XSL</td>
<td>eXtensible Stylesheet Language</td>
</tr>
</tbody>
</table>
Abbildungsverzeichnis

Abb. 1: Übersicht über die Java Editionen nach [Sun00] ......................... 4
Abb. 2: Modell der Handlungsregulationstheorie ................................. 10
Abb. 3: Zusammenhang der BDI-Architektur ...................................... 12
Abb. 4: Klassifikation von Agentenplattformen nach [PBL05a] ............. 13
Abb. 5: Aufbau des ADF für Jadex Agenten ......................................... 14
Abb. 6: Zusammenhang von Jadex-Agent, Adapter und Plattform .......... 20
Abb. 7: Kommunikation nach dem Proxy-Entwurfs muster .................... 28
Abb. 8: Synchronre Kommunikation .................................................... 29
Abb. 9: Asynchronre Kommunikation mittels Queue .............................. 30
Abb. 10: Java EE Architektübersicht nach [BCEHJ06] ........................... 34
Abb. 11: Kommunikationsmodelle von JMS nach [BaGa02] .................. 41
Abb. 12: Container- und komponentenbasierte Integration .................... 47
Abb. 13: Beispielarchitektur einer proprietären Integration von Jadex ... 51
Abb. 14: Architektur von LS/TS nach [RiCK05] ..................................... 53
Abb. 15: Architektur der Adaptive Enterprise Solution Suite nach [AgIn05] ............................................................... 55
Abb. 16: Realisierung des Nachrichtentransports ................................ 57
Abb. 17: Realisierung der Agentzustandsspeicherung ............................ 58
Abb. 18: Vor- und Nachteile der Integrationsarten ............................... 60
Abb. 19: Architektur der gewählten komponentenbasierten Integration von Jadex ............................................................... 63
Abb. 20: UML-Klassendiagramm der Integration .................................. 64
Abb. 21: Sequenzdiagramm des wechselseitigen Zugriffs von Adapter und Agent ................................................................. 67
Abb. 22: Webinterface für die Dienste des AMS ................................... 70
Abb. 23: Packetabhängigkeiten des Jadexframeworks ............................ 73
Abb. 24: Leistungsverhalten optimistischer Sperrprotokolle beim schreibenden Zugriff ......................................................... 75
Abb. 25: Abstrakte Darstellung eines Streckennetzes ........................... 78
Abb. 26: Petrinetzdarstellung des Dijkstra-Algorithmus ....................... 79
Abb. 27: Sequenzdiagramm einer Suchanfrage ................................... 80
Abb. 28: Webinterface der Beispielanwendung .................................... 81
Literaturverzeichnis


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