Diplomarbeit

Konzeption, Architektur und Integration eines Regel- und Aktionenprozessors zur regelbasierten Manipulation von Objekten in E-Commerce-Applikationen

Matthias Kallfass

September 2002

Forschungszentrum Informatik
an der Universität Karlsruhe

Diplomarbeit

Konzeption, Architektur und Integration eines Regel- und Aktionenprozessors zur regelbasierten Manipulation von Objekten in E-Commerce-Applikationen

Matthias Kallfass

September 2002

Hauptreferent:	Prof. Dr. rer. nat. habil. Paul Levi
Korreferent:	Prof. Dr.-Ing. Peter C. Lockemann
Betreuer:	Dr. Dipl.-Inform. Hans Breckle
	Dipl.-Math. Armin Bantle

Hiermit erkläre ich, die vorliegende Arbeit selbständig erstellt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Karlsruhe, den 30. September 2002 .....

(Matthias Kallfass)

In Erinnerung an meine
Mutter
†21.12.2000

Vorwort

Die vorliegende Diplomarbeit wurde zum größten Teil bei der INTERSHOP Communications GmbH, Niederlassung Stuttgart, erstellt.

Mein besonderer Dank gilt den beiden Niederlassungsleitern Herrn Dipl.-Inform. Frank Schneider und Herrn Dipl.-Kfm. Edgar Haller für die Möglichkeit, diese Diplomarbeit bei INTERSHOP zu erstellen. Für die organistorischen Hilfen und stets fruchtbaren Gespräche danke ich meinem Teamleiter Herrn Dipl.-Ing. Benjamin Rost und meinem technischen Betreuer, Herrn Dipl.-Ing. Gerd Weckenmann, sowie allen Mitarbeitern der Niederlassung Stuttgart für das hervorragende Arbeitsklima.

Besonderen Dank gebührt Herrn Prof. Dr. rer. nat. habil. Paul Levi für die Übernahme des Referats und Herrn Prof. Dr.-Ing. Peter C. Lockemann für die Bereitschaft, das Korreferat zu übernehmen. Die wissenschaftliche Betreuung übernahmen Herr Dr. Dipl.-Inform. Hans Breckle und Herr Dipl.-Math. Armin Bantle. Durch ihre Bereitschaft zu fachlichen und organisatorischen Diskussionen ermöglichten sie einen reibungslosen Ablauf der Diplomarbeit, dafür herzlichen Dank!

Meine Schwester Beate und Oliver Balle haben die Ausarbeitung sorgfältig Korrektur gelesen, vielen Dank für die Bereitschaft dies zu tun.

Diese Diplomarbeit bildet den Schlusspunkt meines Studiums der Informatik an der Universität Karlsruhe. An dieser Stelle möchte ich mich vor allem bei meiner Familie, aber auch bei all meinen Verwandten, Freunden und Bekannten für die jahrelange Unterstützung während meiner Studienzeit bedanken.

Inhaltsverzeichnis

Kapitel 1

Einführung

Die Entwicklung von Informationssystemen hatte und hat große Auswirkungen auf die Art und Weise wie Geschäfte abgewickelt werden. In den letzten 20 bis 30 Jahren wurden Geschäftsprozesse in zunehmendem Maße mit Hilfe von Informationssystemen umgesetzt. In den letzten zehn Jahren sind die Anforderungen, denen betriebliche Informationssysteme genügen müssen, stark angewachsen. Neue Konzepte wurden erforscht um effektive und zuverlässige Informationssysteme zu entwickeln, die diesen wachsenden Anforderungen gerecht werden.

Durch die zunehmende Verbreitung des Internets entstanden neue Geschäftsprozesse und -arten, wie der Austausch von Geschäftsdaten zwischen Firmen über das Internet (EDI - Electronic Data Interchange) oder der Vertrieb von Produkten und Dienstleistungen über das Internet (E-Commerce - Electronic Commerce).

Parallel zu dieser Entwicklung erforschte die Informatik die Möglichkeit, menschliches Wissen innerhalb von Rechnersystemen zu speichern und mit deren Hilfe Probleme selbständig zu lösen („Künstliche Intelligenz“). Ein Ergebnis dieser Forschungen stellen wissensbasierte Systeme dar. Ein Hauptanwendungsgebiet von wissensbasierten Systemen sind die so genannten Expertensysteme, deren Hauptzweck es ist, einen oder mehrere Experten zu simulieren und dessen Wissen jederzeit einem größeren Anwenderkreis zur Verfügung zu stellen.

1.1 Motivation

Die initiale Idee zu dieser Diplomarbeit stammt aus einem E-Commerce-Projekt, das einen e-Marketplace realisiert. Das Projekt wurde mit dem Problem konfrontiert, dass die implementierten Geschäftsprozesse ausschließlich in der Obhut des Marktplatzbetreibers liegen und die über den Marktplatz vertreibenden Verkäufer keine Möglichkeit haben, die Geschäftsprozesse an ihre Bedürfnisse anzupassen. Kurzfristige Marketingaktionen wie zum Beispiel: „Kaufen Sie 2

Produkte zum Preis von einem“ oder „Wenn sie Waren im Wert von X € kaufen, bekommen sie dieses Geschenk kostenlos dazu“ sind nicht ohne erheblichen Konfigurations- und Programmieraufwand von Seiten des Marktplatzbetreibers und somit auch Kosten zu realisieren.

Eine plausible Idee, diesen Nachteil zu beheben, ist die Integration eines Regel- und Aktionenprozessors, der aufgrund von Regeln, Objekte innerhalb eines E-Commerce-Systems modifizieren kann. Mit Hilfe dieser Regeln sollen zum Beispiel die oben genannten Marketingaktionen auf einfache Art und Weise durch die über den Marktplatz vertreibenden Verkäufer realisiert werden können. Innerhalb eines Marktplatzes kann jeder Verkäufer seine Regel selbstständig ändern und somit die Geschäftsprozesse kostengünstig an seine Bedürfnisse anpassen.

Als Grundlage des Regel- und Aktionenprozessors soll ein wissensbasiertes System dienen, das die aktuelle Situation anhand der Zustände der Objekte innerhalb des E-Commerce-Systems bestimmt und wiederum Aktionen auf diesen Objekten ausführt. Die Regeln („das Wissen“) des wissensbasierten Systems sollten direkt von einem entsprechenden Domänenexperten (Marketing-, Controlling-Experte, ...) erzeugt und verändert werden können, ohne dass dieser Wissen über die Systemarchitektur des E-Commerce-Systems oder Programmierkenntnisse besitzen muss.

1.2 Aufgabenstellung

Mittels eines Regel- und Aktionenprozessors sollen Objekte innerhalb eines E-Commerce-Systems durch Regeln der Form „**WENN** <*Vorbedingung*>, **DANN** <*Aktion*>“ manipuliert werden können. Mit Hilfe dieser Regeln können Geschäftsprozesse innerhalb des E-Commerce-Systems erstellt und manipuliert werden. Durch den Einsatz des Regel- und Aktionenprozessors erhofft man sich, dass eine Teilmenge der Geschäftsprozesse schneller, kostengünstiger und auf einem höheren Abstraktionsniveau geändert und erstellt werden können.

Regeln der Form „**WENN** <*Vorbedingung*>, **DANN** <*Aktion*>“ sind in der Informatik im Bereich der wissensbasierten Systeme bekannt und erforscht. Die Regeln stellen das Wissen des regelbasierten Systems dar, das in bestimmten Situation (repräsentiert durch die aktuellen Fakten) angewandt wird.

Im Rahmen der Diplomarbeit soll evaluiert werden, ob es möglich und sinnvoll ist, einen solchen Regel- und Aktionenprozessor in ein **bestehendes** E-Commerce-System zu integrieren. Es soll eine geeignete Systemarchitektur für die Integration gefunden und prototypisch implementiert werden. Als E-Commerce-System soll enfinity von der INTERSHOP Communications AG eingesetzt werden.

Zur Wissensakquisition soll eine benutzerfreundliche Symbolsprache definiert werden, mit deren Hilfe ein Geschäftsexperte die Regeln des Regel- und Aktionenprozessors modifizieren kann.

Abschließend soll aufgezeigt werden, innerhalb welcher Grenzen ein solches System unter realen Bedingungen sinnvoll einsetzbar ist.

1.3 Gliederung

In Kapitel 2 werden die Grundlagen der Diplomarbeit vorgestellt. Die Grundlagen repräsentieren den aktuellen Stand der Technik und der Forschung. Es wird auf die Gebiete „Betriebliche Informationssysteme“ und „E-Commerce-Systeme“ eingegangen. Die aktuellen Forschungsergebnisse zum Thema „Geschäftsregeln“ werden in Abschnitt 2.3 dargestellt. Wissensbasierte Systeme werden am Ende des Kapitels dargestellt, dabei wird auf die Wissensakquisition, die Wissensrepräsentation, den Vorgang der Inferenz sowie den zur Inferenz eingesetzten Rete-Algorithmus eingegangen.

Kapitel 3 beschreibt die Konzeption der in der Diplomarbeit erarbeiteten Integration eines Regel- und Aktionenprozessors in ein bestehendes E-Commerce-System. Dabei wird auf die Migration im Vorgehensmodell und auf die Systemarchitektur eingegangen. Am Ende dieses Kapitels wird die entworfene benutzerfreundliche Regeldarstellung beschrieben.

Die konkrete Realisierung der Integration wird in Kapitel 4 dargestellt. Dabei wird auf die Anbindung der Regelmaschine Jess, die bei der Implementierung der Diplomarbeit zum Einsatz kommt, eingegangen.

Die Leistung der vorgestellten Integration wird in Kapitel 5 diskutiert. Die Leistungsanalyse bezieht sich zum einen auf die Performance sowie auf die konkreten Anwendungsmöglichkeit der vorgestellten Lösung.

Eine Zusammenfassung der Arbeit wird in Kapitel 6 gegeben. Dieses Kapitel endet mit dem Ausblick auf Verbesserungs- und Weiterentwicklungsmöglichkeiten.

In Anhang A wird das Ergebnis der im Rahmen dieser Diplomarbeit getesteten Regelmaschinen dargestellt.

Kapitel 2

Grundlagen

Im Folgenden werden die Grundlagen, auf denen die Diplomarbeit beruht, beschrieben. Es wird der aktuelle Stand der Technik, Entwicklung und Forschung dargestellt.

In Abschnitt 2.1 werden betriebliche Informationssysteme und deren Entwicklungsprozesse dargestellt. In Abschnitt 2.2 wird auf die Sonderform eines betrieblichen Informationssystems, das E-Commerce-System, eingegangen. Abschnitt 2.3 beinhaltet die Beschreibung des Konzeptes der Geschäftsregeln (*engl.: business rules*) sowie deren Möglichkeiten und Grenzen. Abschließend werden in Abschnitt 2.4 wissensbasierte Systeme vorgestellt und deren zugrunde liegende Technik beschrieben.

2.1 Betriebliche Informationssysteme

Unser Wirtschaftssystem würde in dieser Form nicht bestehen, wenn nicht eine Vielzahl der betrieblichen Aktivitäten und Aufgaben mit Hilfe von Rechnersystemen bearbeitet werden könnten. Das Finanzsystem, die Börse, aber auch kleinere und mittlere Unternehmen würden ohne elektronische Datenverarbeitung unter der anfallenden Datenflut zusammenbrechen.

Die Wirtschaftsinformatik beschäftigt sich seit geraumer Zeit mit der Erstellung von Informationssystemen zur Unterstützung und Umsetzung solcher Geschäftsprozesse. Nach [SH99] lässt sich ein Geschäftsprozess folgendermaßen definieren:

> **Definition 2.1.1 (Geschäftsprozess)**
> *Ein Geschäftsprozess ist eine Folge oder eine Menge von logisch zusammengehörenden Aktivitäten oder Geschäftsvorgängen, die für das Unternehmen einen Beitrag zur Wertschöpfung leisten und sich in der Regel am Kunden orientiert.*

In den meisten heute verfügbaren Geschäftsmodellen, werden auch betriebsintern Dienstnehmer als Kunden angesehen. In diesem Fall ist ein Geschäftsprozess immer am Kunden orientiert. Ein Geschäftsprozess besitzt immer einen definierten Start- und Endpunkt. Die Mehrzahl der Geschäftsprozessmodelle ist ereignisorientiert, das heißt, wenn ein bestimmtes Ereignis (Auslöser) eintritt, wird der Geschäftsprozess abgearbeitet. Die Aktivitäten, die innerhalb eines Geschäftsprozesses abgearbeitet werden, müssen nicht zwingenderweise sequentiell abgearbeitet werden. Parallel ablaufende Aktivitäten innerhalb eines Geschäftsprozesses haben immer einen Synchronisationspunkt, der letztmögliche Synchronisationspunkt ist der Endpunkt eines Geschäftsprozesses.

Den Teil eines Informationssystems, der den eigentlichen Geschäftsprozess implementiert, bezeichnet man als die Geschäftslogik der betrieblichen Anwendung.

2.1.1 Anforderungen an Geschäftsprozesse

Die Anforderungen, denen Geschäftsprozesse genügen müssen, haben sich in den letzten Jahrzehnten sehr stark geändert. Diesen neuen Anforderungen müssen auch die Informationssysteme, die diese Geschäftsprozesse implementieren oder unterstützen, gerecht werden. Folgende Ursachen lassen sich für die wachsenden Anforderungen identifizieren:

- Zunahme des Anteils der Geschäftsprozesse, die mit Hilfe von Informationssystemen abgearbeitet werden, zum Beispiel

  CRM: customer relationship management - Anwendung für das Management von Kundenbeziehungen.

  ERP: enterprise resource planning - Anwendung für Ressourceplanung.

- Moderne Technik ermöglicht es Firmen, neue Formen von Geschäften zu tätigen, neue Geschäftsprozesse entstehen (z.B. E-Commerce - electronic commerce, M-Commerce - mobile commerce).

- Flexibilisierung und Anpassbarkeit der Geschäftsprozesse - aufgrund der zunehmenden Globalisierung und Öffnung der Märkte werden Geschäftsprozesse immer häufiger von der aktuellen Marktsituation bestimmt [Wes99].

Als logische Konsequenz daraus sehen moderne Systemarchitekturen eine zunehmende Kapselung der Geschäftslogik vor. Diese Trennung ermöglicht es, die Geschäftslogik zu ändern, ohne dass dies eine Änderung der anderen Systemkomponenten (etwa der Datenhaltungslogik oder der Darstellungslogik) zur Folge hat. Dieses Vorgehen kann es auch ermöglichen, dass man die Geschäftslogik eines Systems während des Betriebs verändern kann.

2.1.2 Vorgehensmodelle

Die Art und Weise wie Informationssysteme für den betrieblichen Einsatz entwickelt, gewartet und erweitert werden, unterlag in den letzten Jahren starken Änderungen. Diese Änderungen wurden vor allem durch die zunehmende Objektorientierung und die wachsende Möglichkeit, Software zu modellieren, vorangetrieben [Wes99]. Ein weiterer Grund dafür ist, dass die ersten betrieblichen Anwendungen reine Datenhaltungssysteme waren. Diese wurden zum großen Teil aus einer sehr datengetriebenen Sicht entwickelt. Weitere Geschäftsprozesse wurden erst in der Folgezeit mit Hilfe von Informationssystemen umgesetzt. Im Folgenden werden die wichtigsten Vorgehensweisen beschrieben.

2.1.2.1 Herkömmliches Vorgehensmodell

Bis Mitte der neunziger Jahre setzte sich der Modellierungsprozess von betrieblichen Informationssystemen aus zwei sehr stark abgegrenzten Teilen zusammen. Abbildung 2.1 skizziert diese Vorgehensweise.

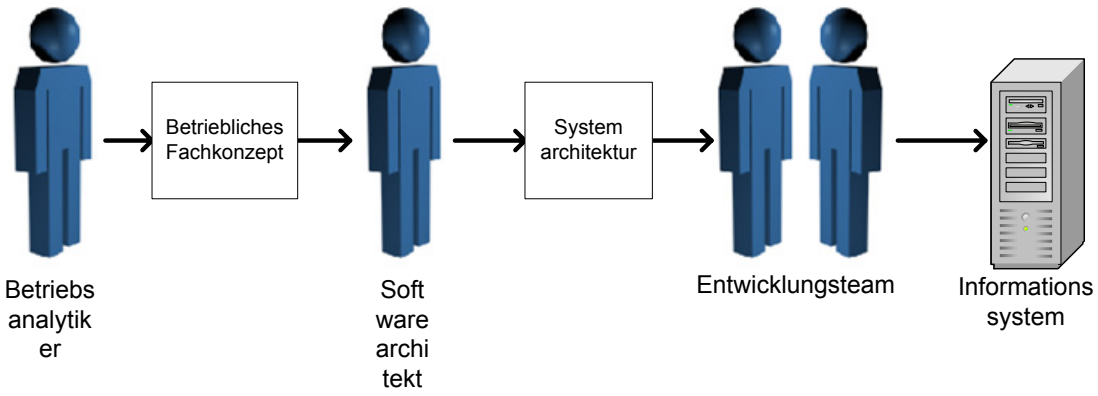

Abbildung 2.1: Herkömmliches Vorgehensmodell

Der erste Teil besteht aus der Definition der Geschäftsprozesse, die durch das System implementiert werden soll. Diese Geschäftsprozessmodellierung wurde von einem Betriebsanalytiker (*engl.: business analyst*) vorgenommen. Ein Artefakt, das in diesem Teilprozess entsteht, ist das betriebliche Fachkonzept.

Dieses betriebliche Fachkonzept wird von dem Softwarearchitekt aufgegriffen und in ein Datenverarbeitungskonzept bzw. in eine Softwarearchitektur überführt. Aufgrund dieser Softwarearchitektur implementiert das Entwicklungsteam das Informationssystem.

Ein Hauptproblem dieser Vorgehensweise ist, dass innerhalb dieser beiden Modellierungstätigkeiten sehr unterschiedliche Entitäten behandelt werden. Im betrieb-

lichen Fachkonzept erscheinen Entitäten wie Lieferant, Käufer oder Rechnung, während im Datenverarbeitungskonzept Entitäten wie Listen, Datenbanktabellen oder Datenstrukturen erarbeitet werden [Wes99].

Diese Vorgehensweise ist ein Grund dafür, dass das Endprodukt oftmals ein sehr monolithisches System, das wenig Möglichkeiten zur Erweiterung oder zur Änderung der implementierten Geschäftsprozesse bietet, war [Wes99, HMW98].

2.1.2.2 Vorgehensmodell mit Business Objects

Das Konzept der Business Objects (Geschäftsobjekte) zielt darauf ab, Entitäten zu definieren, die sowohl in der Geschäftsprozessmodellierung als auch in der Systemarchitektur und der Implementierung des betrieblichen Informationssystems verwendet werden können. Nach [Bus97] kann eine Business Object folgendermaßen definiert werden:

> **Definition 2.1.2 (Business Object)**
> *Ein Business Object umfasst die Informationen eines realen Geschäftskonzepts, Operationen und deren Grenzen, die innerhalb dieses Konzepts ausgeführt werden und die Verknüpfungen innerhalb des Konzepts sowie deren Beziehungen zu anderen Business Objects.*

Business Objects sollen hauptsächlich die Kommunikation zwischen dem Betriebsanalytiker und dem Softwarearchitekten verbessern. Sie bilden die gemeinsame Grundlage zur Definition der Anforderungen, die das Informationssystem erfüllen muss (siehe Abbildung 2.2). Der Betriebsanalytiker benutzt Business Objects zur Definition der Geschäftsprozesse, der Softwarearchitekt benutzt die gleichen Business Objects zur Modellierung des Softwaresystems.

Es ist nicht zwingend, dass ein Business Object, das im Geschäftsprozessmodell erscheint auch im Systemmodell bzw. in der Softwarearchitektur erscheinen muss. Aus informationstechnischer Sicht, kann es durchaus sinnvoll sein, ein Business Object nicht in dieser Form zu implementieren, durch ein anderes zu ersetzen oder durch eine Komposition von mehreren Objekten zu realisieren. Genauso wenig ist jedes Objekt, das in einem betrieblichen Informationssystem vorhanden ist, ein Business Object. Es gibt in jedem Informationssystem Objekte, deren Existenz nicht durch den implementierten Geschäftsprozess motiviert ist, sondern durch die Tatsache, dass sie aus informationstechnischen Gründen notwendig sind.

Ein weiterer Vorteil der Business Objects gegenüber der herkömmlichen Vorgehensweise ist die Möglichkeit, Frameworks zu bilden. Nach [GHJV96] definiert sich ein Framework folgendermaßen:

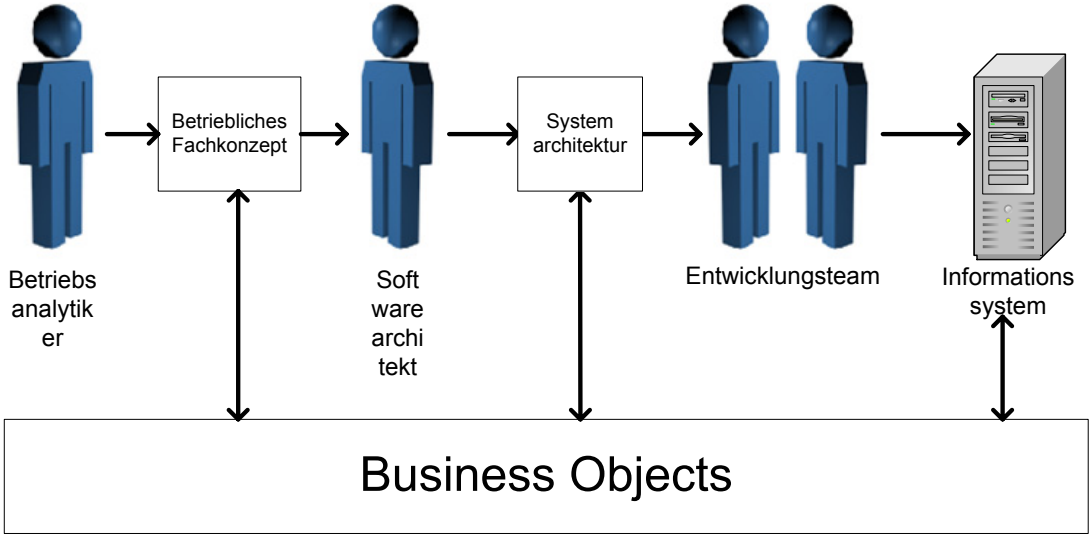

Abbildung 2.2: Vorgehensmodell mit Business Objects

Definition 2.1.3 (Framework)
Ein Framework besteht aus einer Menge von zusammenarbeitenden Klassen, die einen wieder verwendbaren Entwurf für eine bestimmte Klasse von Software darstellen.

Im Kontext der betrieblichen Anwendungen besteht ein Framework aus einer Menge von Klassen, die die Business Objects und deren Beziehungen untereinander (teilweise) implementieren.

Eines der ersten in Java implementierten Frameworks für betriebliche Informationssysteme war das SanFransisco-Framework von IBM [Int99].

Der Kern der Lösungen, die mit einem Framework gelöst werden, werden häufig mittels Entwurfsmustern dargestellt. Folgende Definition wurde aus [GHJV96] zusammengefasst:

Definition 2.1.4 (Entwurfsmuster)
Jedes Entwurfsmuster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass man diese Lösung beliebig oft anwenden kann, ohne sie jemals ein zweites Mal gleich auszuführen.

Ein Entwurfsmuster besitzt folgende grundlegende Elemente:

- *Mustername*
- *Problemabschnitt*

- *Lösungsabschnitt*
- *Konsequenzenabschnitt*

Ein Entwurfsmuster bietet also eine schablonenhafte Lösung für eine bestimmte Art von Problemen. Im Problemabschnitt wird beschrieben, welches Problem mit diesem Entwurfsmuster adressiert wird, in welchem Kontext es sich befindet und welche Bedingungen erfüllt sein müssen, um das Entwurfsmuster sinnvoll anwenden zu können. Der Lösungsabschnitt enthält die Elemente, sowie deren Beziehungen, Zuständigkeiten und Interaktionen. Die Lösung durch das Entwurfsmuster hat bestimmte Vor- und Nachteile, diese werden im Konsequenzenabschnitt beschrieben.

Das bekannteste Entwurfsmuster ist sicherlich das Singleton Muster. Es stellt sicher, dass von einer Klasse nur ein Objekt innerhalb des Systems vorhanden ist.

2.2 E-Commerce-Systeme

E-Commerce-Systeme ermöglichen Firmen, ihre Produkte und Dienstleistungen über das Internet zu vertreiben. Der Vorteil dieser Systeme ist, dass der komplette Geschäftsprozess auf Informationssystemen stattfindet. Zum Beispiel muss gegenüber einer Bestellung per Brief, Fax oder Telefon, diese nicht per Hand oder mittels Scannertechnologie in das Bestellabwicklungssystem eingegeben werden, dies senkt die Kosten pro Bestellprozess erheblich.

Im Folgenden sei die wohl allgemeinste Form eines E-Commerce-Systems, der verkäuferseitige Marktplatz (*engl.: seller-side marketplace*) beschrieben. Als Kunden des Marktplatzbetreibers treten zum einen die über den Marktplatz verkaufenden Firmen, sowie die Kunden, die bei diesen Firmen Produkte oder Dienstleitungen kaufen, auf. Der Marktplatzbetreiber verpflichtet sich gegenüber den verkaufenden Firmen, dass die (Teil-) Bestellungen eines Kunden in dessen Bestellabwicklungssystem eingepflegt werden.

Folgende Rollen und Akteure lassen sich innerhalb eines solchen Markplatzes identifizieren (1 = einmal pro System, n = mehrfach pro System möglich):

Betreiber (*engl.: organizer*) (1): Vertreter des Marktplatzbetreibers, sorgt für den reibungslosen Betrieb des Marktplatzes und administriert den verkäuferübergreifenden Teil des Marktplatzes.

Verkäufer (*engl.: seller*) (n): Vertreter der Firma, die über den Marktplatz Produkte vertreibt. Der Verantwortungsbereich des Sellers besteht aus der Datenpflege (zum Beispiel Katalogdaten) und der Bestellungsabarbeitung.

Käufer (*engl.: buyer*) (n): Kauft Produkte mittels des Marktplatzes bei mindestens einem auf dem Marktplatz vorhandenen Seller.

Alle anderen Formen von E-Commerce-Systemen lassen sich aus dieser Rollendefinition ableiten. So besteht zum Beispiel ein klassisches E-Commerce-System einer Firma aus nur einem Seller und der Organizer gehört der Firma des Sellers an, da die Firma gleichzeitig Betreiber und einziger Verkäufer ist.

2.2.1 Java 2 Enterprise Edition - basierte Systeme

Eine Vielzahl der heute verfügbaren E-Commerce-Systeme basieren auf der von Sun Microsystems entwickelten Plattform Java 2 Enterprise Edition (J2EE) [Sun99b, Sun00b, Sun00a].

Die J2EE-Plattform erweitert die Java 2 Standard Edition (J2SE) [Sun99c] um Komponenten, die hauptsächlich für den betrieblichen Einsatz entwickelt wurden. Die wichtigste Komponente des J2EE ist ein Mechanismus zur peristenten Datenhaltung von Java-Objekten (Enterprise JavaBeans) [Sun99a, Sun00b] in verteilten Systemen. Weitere Komponenten der Plattform sind die JavaServer Pages [Sun99g, Sun00b] (Zugriff auf Java-Objekte bei der Programmierung von HTML-Seiten) und Servlets [Sun99e, Sun00b] (ermöglicht die Ausführung von Objektmethoden innerhalb eines Web-Servers).

2.2.1.1 Die Systemarchitektur

Die Java 2 Enterprise Edition sieht dreistufige (*engl.: three-tier*) Systemarchitektur vor (siehe Abbildung 2.3).

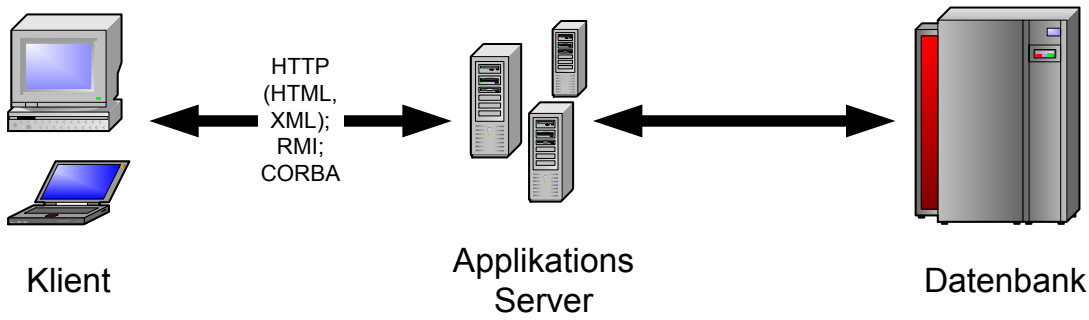

Abbildung 2.3: Die dreistufige Architektur der Java Enterprise Plattform

Die erste Stufe besteht aus den Klienten, die mittels HTTP, Remote Method Invocation (RMI) [Sun99d] oder CORBA auf die Applikations-Server zugreifen.

Die Klienten selbst besitzen keine Geschäftslogik, der Klient ist ausschließlich für die Darstellung der Benutzeroberfläche verantwortlich. Die Geschäftslogik wird innerhalb der Applikations-Server implementiert. Zur persistenten Datenhaltung greifen die Applikations-Server auf ein Datenbanksystem zurück (3. Stufe).

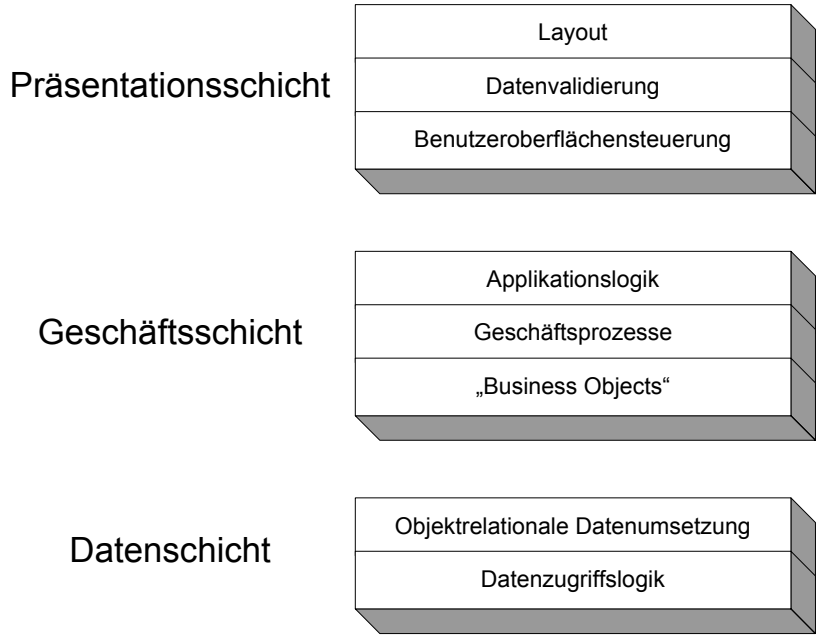

Abbildung 2.4: Die drei Schichten der Java Enterprise Architektur

Abbildung 2.4 zeigt die daraus resultierenden Schichten. Das ist zum einem die Präsentationsschicht (*engl.: presentation layer*), bestehend aus dem Layout der Benutzeroberfläche, der Validierung der Benutzereingaben und der Steuerung der Benutzeroberfläche. Die Geschäftsschicht (*engl.: business layer*) besteht aus der eigentlichen Applikationslogik, die die Geschäftsprozesse mit Hilfe der Business Objects implementiert. Die unterste Schicht besteht aus der objektrelationalen Datenumsetzung (*engl.: O/R-Mapping*) und aus der Datenzugriffslogik, die es ermöglicht, Daten persistent zu halten.

2.2.1.2 Enterprise JavaBeans

Enterprise JavaBeans [Sun98, Sun99a, Sun00b, DP00, Rom99] gibt es in zwei unterschiedlichen Ausprägungen: Session- und EntityBeans. SessionBeans erlauben es, bestimmte Dienste dem Klienten zur Verfügung zu stellen, sie modellieren für gewöhnlich bestimmte Vorgänge bzw. Geschäftsprozesse. EntityBeans repräsentieren Geschäftsobjekte (Business Objects), die in einem persistenten Speicher gehalten werden.

EJBs sind Remoteobjekte, das sind Objekte, auf die auch von entfernten Computern aus zugegriffen werden kann. Dieser entfernte Methodenaufruf wird mittels des Remote Method Invocation-Mechanismus' (RMI) [Sun99d] gemacht.

Enterprise JavaBeans sind Komponenten, die in eine Laufzeitumgebung, dem sogenannten EJB-Container, eingebettet werden. Der EJB-Container stellt einer EntityBean unter anderem folgende Dienste bzw. Schnittstellen zur Verfügung:

Java Transaction API (JTA): Schnittstelle zur Transaktionsabwicklung, diese wird durch einen Transaktionsmanager bereitgestellt.

Java DataBase Connectivity (JDBC): Schnittstelle zum Lesen und Schreiben von Daten in Datenbanken

Container Managed Persistence: Mechanismus zur automatischen Speicherung des Datensatzes einer EJB in einer Datenbank. Der Container Managed Persistence-Service erspart es dem Programmierer selbst die Datenhaltungslogik zu schreiben.

Eine Enterprise JavaBean-Komponente besteht aus einer Reihe von Klassen, Tabelle 2.1 gibt eine Übersicht über die wichtigsten Klassen einer EJB.

Klasse	**Funktion**	**Äquivalent im Relationenmodell**
EJBNAME	Ändern von Attributen und Löschen einer EJB	Tupel bzw. Zeile
*EJBNAME***Home**	Erzeugen, Löschen und Finden von EJBs	Relation bzw. Tabelle
*EJBNAME***Key**	Schlüssel einer EJB, dieser identifiziert eine EJB eindeutig	Schlüssel
*EJBNAME***Bean**	Implementierung der in *EJBNAME* und *EJBNAME***Home** deklarierten Methoden	—

Tabelle 2.1: Übersicht über die Klassen einer EJB

Das JavaBeans-Konzept [Sun97, S. 55] der Getter- und Setter-Methoden wurde bei den Enterprise JavaBeans übernommen. Dieses sieht vor, dass für ein Attribut eine Methode **get**<*ATTRIBUTNAME*>*()* zum Lesen dieses Attributes vorhanden ist. Kann ein Attribut gesetzt werden, so ist innerhalb dieser Klasse eine Methode **set**<*ATTRIBUTNAME*>*(neuerWert : Typ)* deklariert. Der Rückgabewert der get-Methode und der Parameter der set-Methode müssen mit dem Typ

des Attributes übereinstimmen.

Jedes Enterprise JavaBean-Objekt enthält neben den Methoden zum Lesen und Ändern auch eine **remove()**-Methode zum Löschen des Tupels aus der Datenbank und zum Zerstören des EJB-Objektes.

2.2.1.3 Transaktionen

Die J2EE-Plattform stellt einen Transaktionsdienst in Form eines so genannten Transaction Managers bereit. Die Steuerung der Transaktionen geschieht mittels der Java Transaction API [Sun99f].

Nach [SW01] lässt sich eine Transaktion (*engl.: transaction*) folgendermaßen definieren:

> **Definition 2.2.1 (Transaktion)**
> *Mit Hilfe einer Transaktion können mehrere Aktionen zu einer unteilbaren Einheit zusammengefasst werden. Man spricht auch von einer atomaren (Trans-)Aktion.*

Eine Transaktion muss folgenden Forderungen – den Transaktionsparadigmen – genügen [LKK93, S. 415] [SW01]:

Eigenschaft	Beschreibung
Atomizität (*engl.: atomicity*)	Eine Transaktion hat nur als Einheit eine Wirkung nach außen. Sie wird entweder erfolgreich abgeschlossen (**commit**) oder abgebrochen (**abort, rollback**). Wird eine Transaktion abgebrochen, so ist ihre Wirkung so, als wäre sie nie gestartet worden.
Konsistenz (*engl.: consistency*)	Transaktionen bewahren die Systeminvarianten, sie bewirkt einen konsistenten Datenbasiszustand sofern sie auf einen konsistenten Datenbasiszustand aufsetzte.
Isolation (*engl.: isolation*)	Transaktionen sind wechselwirkungsfrei – Parallel ablaufende Transaktionen laufen so ab, als würde jede für sich alleine ablaufen
Dauerhaftigkeit (*engl.: durability*)	Wird eine Transaktion erfolgreich beendet, so ist ihre Wirkung dauerhaft vorhanden.

Tabelle 2.2: Transaktionsparadigmen

Während ihrer Lebensdauer durchläuft eine Transaktion eine Reihe von Zuständ-

en und Zustandsübergängen. Zu Beginn wird sie von dem Zustand *inaktiv* in den Zustand *aktiv* versetzt. Im *aktiv*-Zustand bleibt die Transaktion entweder bis alle Einzelaktionen abgeschlossen sind (Nachfolgezustand: *abgeschlossen*) oder eine Einzelaktion gescheitert ist (Nachfolgezustand: *gescheitert*). Durch Festschreiben (*engl.: commit*) wechselt eine abgeschlossene Transaktion in den Endzustand *persistent*. Durch Zurücksetzen (*engl.: rollback*) wechselt eine gescheiterte Transaktion in den Endzustand *aufgegeben*.

2.2.2 INTERSHOP enfinity

In den folgenden Abschnitten wird die E-Commerce-Applikation der INTERSHOP Communications AG beschrieben. Dieses J2EE-basierte System wird in dieser Diplomarbeit als zugrunde liegende Plattform verwendet. INTERSHOP enfinity baut auf dem objekt-/prozessorientierten Paradigma auf. Die Objektorientierung ist durch die Verwendung der Java 2 Enterprise Edition Plattform gegeben. Die Implementierung der Geschäftsprozesse[1] erfolgt ebenfalls in Form von Prozessen.

2.2.2.1 Systemarchitektur

Eine enfinity Plattform besteht aus einem Transaktionsserver (enfinity Transactivity Server, eTS) und einem Katalogserver (enfinity Catalog Server, eCS). Die beiden Server unterscheiden sich in der darin implementierten Geschäftsframework sowie in den von ihnen beherbergten Geschäftsobjekten. Der eCS beinhaltet die Logik zur Verwaltung der Katalogdaten, zur Suche von Produkten, zur Verwaltung von Warenkörben usw., während der eTS die Logik zur Verwaltung der Kunden und deren Adressen, die Bestellabwicklung mit Preisberechnung usw., implementiert.

Der enfinity Plattform ist ein Webserver mit einem Webadapter vorangestellt. Der Webadapter leitet eine Anfrage an den dafür zuständigen enfinity Server weiter (siehe Abbildung 2.5). Zur persistenten Datenhaltung wird ein externes Datenbanksystem genutzt.

Zur Steuerung, Konfiguration und Entwicklung der Geschäftsprozesse der enfinity Plattform gibt es einen speziellen Klienten, das enfinity Management Center (eMC). Das eMC besteht aus mehreren Komponenten, diese sind:

Editoren für die wichtigsten Geschäftsobjekte: Mit Hilfe dieser Editoren können unter anderem Produkte, Produkttypen, Käufer und deren Adressen, Bestellungen und deren Status angelegt, editiert, gelöscht, importiert und exportiert werden.

[1] siehe Definition 2.1.1, Seite 5

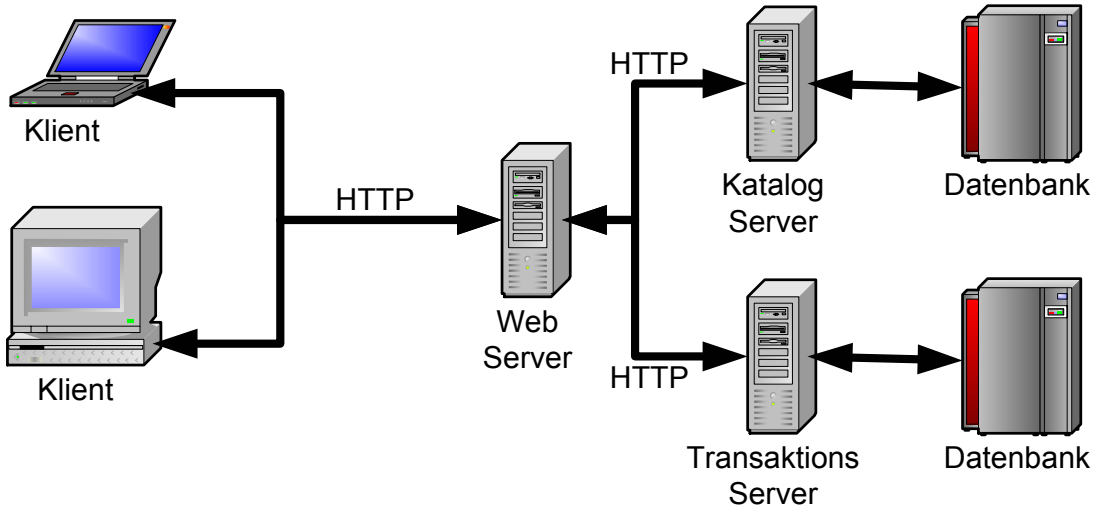

Abbildung 2.5: Die funktionalen Komponenten einer enfinity-Plattform

Administrationskomponente: Mithilfe dieser Komponente können die einzelnen Server einer enfinity-Plattform verwaltet und überwacht werden.

Visual Pipeline Manager: Editier- und Verwaltungseinheit für die Geschäftsprozesse, diese Komponente wird in Abschnitt 2.2.2.2 detailliert beschrieben. Abbildung 2.7 zeigt den Visual Pipeline Manager mit einer Pipeline.

Template Designer: Der Template Designer ermöglicht es dem eMC Benutzer, den Inhalt und das Layout von Templates zu erstellen. Templates werden in der INTERSHOP Meta Language (ISML) implementiert.

Analyse und Report-Komponente: Zeigt die wichtigsten Kennzahlen einer E-Commerce-Site in einer übersichtlichen sowohl textuellen als auch graphischen Darstellung. Das sind zum Beispiel: „Was ist das meistgekaufte Produkt?“, „Wie groß ist der Tagesumsatz innerhalb eines bestimmten Zeitraumes?“, „Wie groß sind die Antwortzeiten für eine Anfrage innerhalb eines bestimmten Zeitraumes?“

Abbildung 2.6 zeigt die verschiedenen Schichten der enfinity Architektur. Der Webadapter leitet eine Anfrage an einen enfinity Server weiter. Diese Anfrage löst in dem entsprechenden enfinity Server eine Pipelineabarbeitung aus. Innerhalb der Pipelineabarbeitung wird mittels des Geschäftsframeworks auf die entsprechenden Geschäftsobjekte zugegriffen. Die Ergebnisse der Pipelineabarbeitung werden in dem PipelineDictionary gespeichert. Der Templateprozessor, der für das letztendliche Layout der HTML-Seiten verantwortlich ist, ersetzt die im Template enthaltenen Platzhalter mit den im PipelineDictionary enthaltenen Werten.

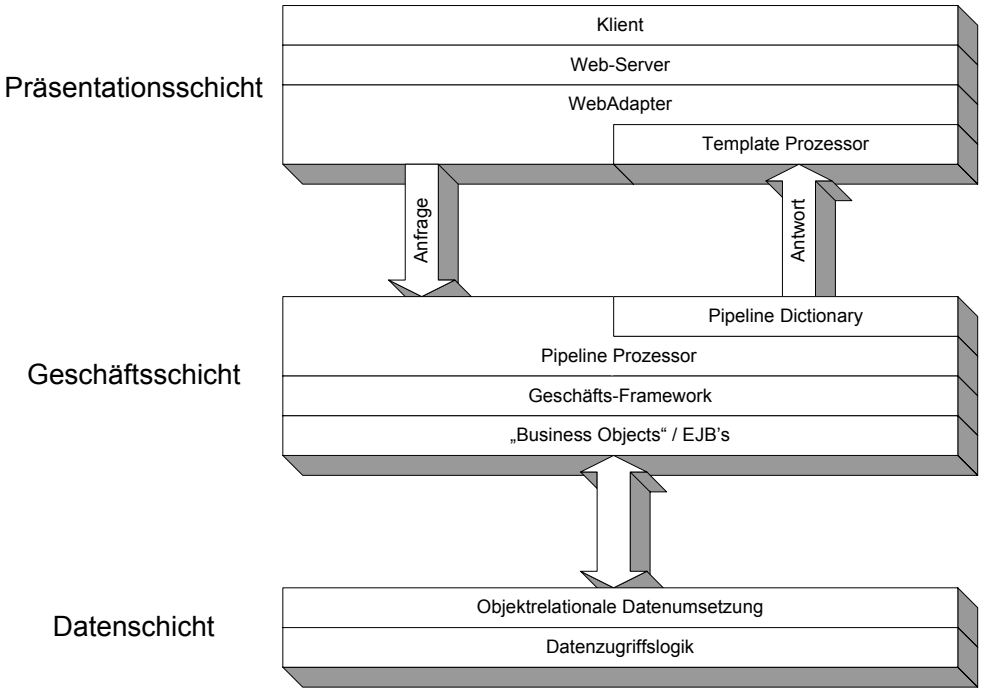

Abbildung 2.6: Das Schichtenmodell der enfinity Architektur

Weiterführende Informationen über die Systemarchitektur und die Komponenten eines enfinity-Systems sowie deren Kommunikation untereinander werden in [INTi, Abschnitt: Komponentenbasiertes Modell, S. 3 – S.17] dargestellt.

2.2.2.2 Anfrageabwicklung

Eine Anfrage an einen enfinity Server stößt eine Pipelineabarbeitung innerhalb des Pipelineprozessors an.

Eine Pipeline besteht aus einem oder mehreren benannten Startknoten, Pipelineknoten (Pipelets) sowie einem abschließenden Interaktionsknoten. Weitere Knotentypen sind: Entscheidungsknoten – Knoten die anhand von booleschen Ausdrücken den weiteren Ablauf einer Pipeline bestimmen, Aufrufknoten – Knoten, die Unterpipelines aufrufen, Vereinigungsknoten – Knoten, die verschiedene eingehende Pipelineabläufe in einen ausgehenden Pipelineablauf zusammenführen, sowie Sprungknoten – Knoten, die eine andere Pipelineabarbeitung starten.

Während der Pipelineabarbeitung existiert ein PipelineDictionary, in dem Objekte unter einem bestimmten Namen gespeichert werden können. Ein Pipelet ist eine leichtgewichtige Komponente, die im Normalfall atomare Funktionen implementiert. Typische Funktionen eines Pipelets sind zum Beispiel „Erzeuge ein neues Käuferobjekt“, „Setze die Adresse des Käufers auf diese Werte“ oder „Erzeuge

aus dem Warenkorb eine Bestellung“. Diese Funktionen setzt ein Pipelet unter Verwendung des darunterliegenden Geschäftsframeworks und der Geschäftsobjekte um. Als Parameter können einem Pipelet die Werte des PipeletDictionaries sowie die Parameter der Anfrage dienen. Pipelets die Rückgabewerte haben, legen diese wiederum in das PipelineDictionary. Abbildung 2.7 zeigt eine solche Pipeline.

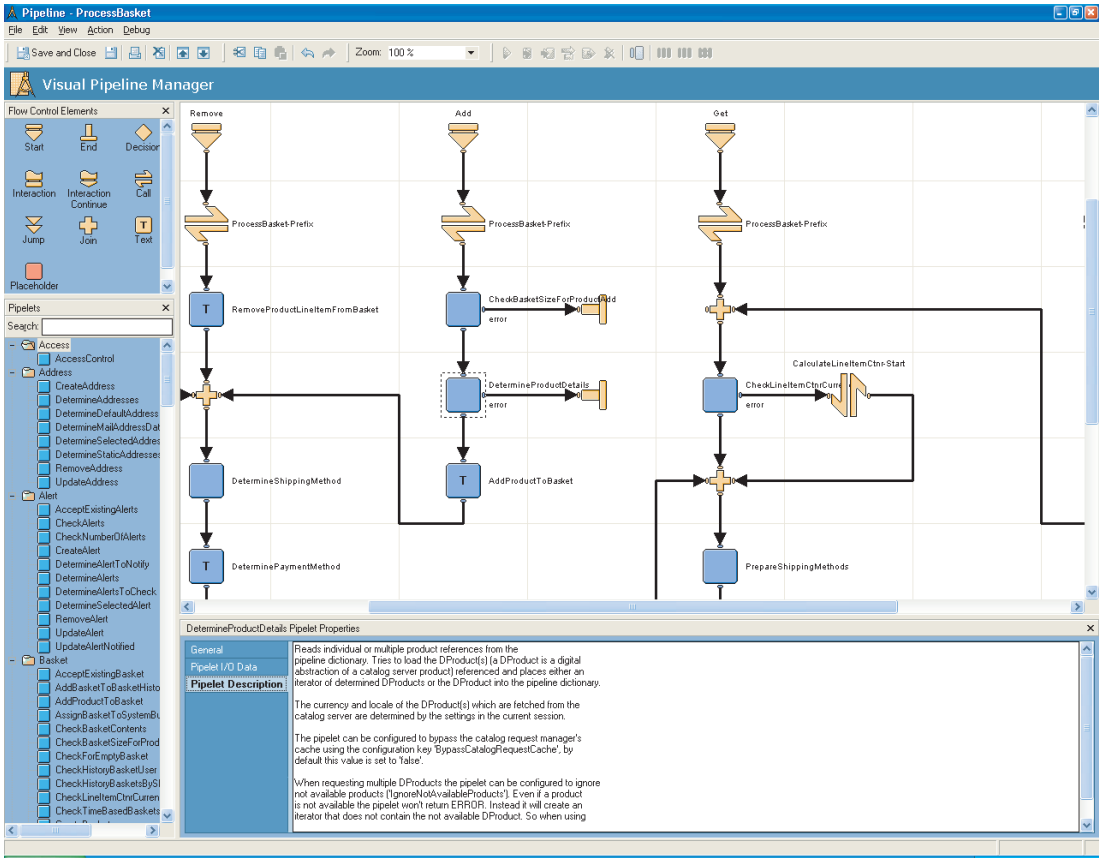

Abbildung 2.7: Visual Pipeline Manager von INTERSHOP enfinity

Der abschließende Interaktionsknoten dient zur Kommunikation mit dem Benutzer. Dieser legt das dazu benötigte Template fest. Dem Templateprozessor dient das PipelineDictionary als Wertetabelle zur Ersetzung der im Template enthaltenen Platzhalter, dabei kann auch auf die Attribute der im PipelineDictionary enthaltenen Objekte zugegriffen werden.

Eine Anfrage wird innerhalb einer Sitzung (*engl.: session*) abgearbeitet. Eine Sitzung besteht aus der Gesamtmenge der Anfragen eines Klienten. Bei der ersten Anfrage eines Klienten wird ein Sitzungsobjekt für diesen Klienten erzeugt. Dieses Objekt besitzt eine eindeutige ID, die der Klient bei den weiteren Anfragen

als Anfrageparameter benutzt. Eine Sitzung wird beendet, sobald ein Klient eine bestimmte Zeit keine Anfrage an den Server abgesetzt hat.

Eine detaillierte Darstellung einer Anfragebearbeitung und der daran beteiligten Softwarekomponenten befindet sich in [INTi, Abschnitt: Fallmodell für eine Anfrage, S. 18 – S. 27].

2.2.2.3 Vorgehensmodell mit INTERSHOP enfinity

Innerhalb des Entwicklungsprozesses lassen sich folgende Rollen und Akteure identifizieren[2]:

Betriebsanalytiker (*engl.: business analyst*): Analysiert die bestehende Infrastruktur und entwickelt auf dieser Grundlage die Geschäftsprozesse und Geschäftslogik des E-Commerce-Systems.

Softwarearchitekt (*engl.: software architect*): Modelliert die Systemarchitektur des E-Commerce-Systems. Als Grundlage stehen ihm die Ergebnisse des Betriebsanalytikers zur Verfügung.

Softwareentwickler (*engl.: software developer*): Implementiert das System aufgrund der vom Softwarearchitekt zur Verfügung gestellten Systemarchitektur.

(Web-)Designer (*engl.: (web-)designer*): Gestaltet die Darstellung der Inhalte / Informationen eines E-Commerce-Systems.

Das Vorgehensmodell mit INTERSHOP enfinity stellt eine Erweiterung des Vorgehensmodell mit Business Objects (siehe Abschnitt 2.1.2.2) dar.

Die Erweiterung des Vorgehensmodells ergibt sich aus der Trennung der Präsentations- und der Geschäftsschicht (siehe Abbildung 2.6). Diese Trennung macht die Geschäftslogik mit den Geschäftsprozessen weitgehend unabhängig von der Kommunikation mit dem Benutzer. Der (Web-)Designer hat die Aufgabe, das Layout der Ein- und Ausgabemasken für die Benutzerkommunikation zu entwerfen. Das Entwicklungsteam (siehe Abbildung 2.2) besteht bei dem Vorgehensmodell mit INTERSHOP enfinity aus den Softwareentwicklern, die die Geschäftslogik implementieren und den (Web-)Designern, die die Ein- und Ausgabemasken entwerfen und umsetzen.

[2]eine komplette Übersicht der Rollen und Akteure, die an der Entwicklung, Administration und des Betriebs eines enfinity-Systems beteiligt sind, ist in [INTf, Abschnitt: Akteure und ihre Rollen, S. 23] zu finden

In Abschnitt 2.1.2.2 wurde als ein Vorteil des Vorgehensmodell mit Business Objects die Möglichkeit zur Frameworkbildung genannt. INTERSHOP enfinity wird mit dem *Beehive*-Framework [INT01] ausgeliefert. Das Beehive-Framework enthält eine Vielzahl zusammenarbeitender Geschäftsobjekte, zum Beispiel zur Verwaltung von Kunden und Adressen, zum Aufbau von Katalogen und zur Bestellabwicklung.

2.3 Geschäftsregeln

Es gibt keine allgemeingültige Definition für Geschäftsregeln (*engl.: business rules*). Die folgende Definition wurde von der Business Rules Group[3] übernommen [Bus00a] und übersetzt. Die Business Rules Group ist eine Interessengemeinschaft von Experten, die sich mit der Modellierung von Unternehmen mittels Regeln und der Umsetzung dieser in Informationssystemen beschäftigt.

> **Definition 2.3.1 (Geschäftsregel)**
> *Eine Geschäftsregel ist eine Anweisung, die bestimmte Aspekte eines Geschäfts festlegt oder erzwingt. Ziel ist es, vorgegebene Geschäftsstrukturen durchzusetzen oder die Art des Geschäfts zu kontrollieren oder zu beeinflussen. Geschäftsregeln sind atomar, das heißt, sie können nicht weiter unterteilt werden.*

Geschäftsregeln sind also keine informationstechnische Spezifikation sondern die Summe von Wissen, Definitionen, Vorgehensweisen, Erfahrungen von Geschäftsbetreibenden, Managern bzw. Experten. Eine gültige Geschäftsregel ist zum Beispiel:

Ein Kunde kann nur Waren in Höhe seines Kreditrahmens bestellen.

Geschäftsregeln stimmen im Allgemeinen mit der Denkweise von Managern überein und beschreiben Sachverhalte deklarativ, sie beschreiben das „Was“ und nicht das „Wie“.

Mittels Geschäftsregeln kann man nicht nur bestimmte Aspekte oder Teile eines Unternehmens spezifizieren, sie durchdringen sämtliche Aspekte der Unternehmensmodellierung. Abbildung 2.8 gibt eine Übersicht über die Unternehmensmodellierung mit Hilfe von Geschäftsregeln.

Sämtliche Geschäftsregeln können in gleichbedeutende Implikationen umgeformt werden. So kann obiges Beispiel in die Implikation

[3] siehe `www.businessrulesgroup.org`

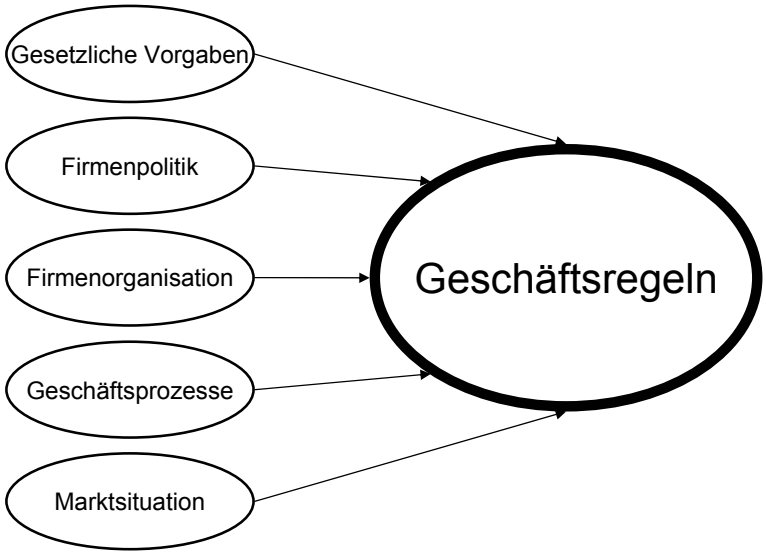

Abbildung 2.8: Geschäftsregeln in einem Unternehmen

Wenn der Wert einer Bestellung eines Kunden dessen Kreditrahmen übersteigt, **dann** lehne die Bestellung ab.

überführt werden.

David Plotkin stellt in [Plo99] weitere Aspekte, denen Geschäftsregeln genügen sollten, vor. Das sind:

Präzision: Eine Geschäftsregel muss eindeutig sein, falls es mehrere Interpretationen einer Regel gibt, muss man diese neu formulieren.

Konsistenz: Eine Gesamtmenge der Geschäftsregeln darf keine sich widersprechenden Regeln enthalten.

Nichtredundanz: Eine Menge von Geschäftsregeln darf keine Regel enthalten, durch die, falls sie gelöscht wird, keine Information verloren geht.

Geschäftsorientierung: Regeln müssen so formuliert sein, dass sie von Managern verstanden werden können. Innerhalb der Regeln müssen Ausdrücke verwendet werden, die eindeutig sind und die in der Geschäftswelt geläufig sind.

zu dem Management gehörend: Geschäftsregeln gehören zu dem Management, nur das kann Regeln erstellen, modifizieren oder löschen.

Geschäftsregeln stellen eine Möglichkeit (von mehreren) dar, Geschäftsprozesse zu spezifizieren. Diese Möglichkeit macht Geschäftsregeln auch für die Entwicklung und den Betrieb von betrieblichen Informationssystemen relevant. Aus informationstechnischer Sicht betreffen Geschäftsregeln die Daten, die innerhalb des

Systems gespeichert werden, die Logik innerhalb des Systems, die diese Daten manipuliert, sowie die Logik, die, falls vorhanden, darüber entscheidet, ob die Daten gespeichert oder verworfen werden[Bus00a].

2.3.1 Typen von Geschäftsregeln

Das Business Rules Model der Business Rules Group sieht eine dreiteilige Partitionierung der Geschäftsregeln nach ihrem Typ vor. Diese sind:

Strukturelle Zusicherung (*engl.: strucural assertion*): Ein Konzept oder eine Feststellung, die eine Aussage über die Struktur einer Unternehmung macht.

Aktionsanweisung (*engl.: action assertion*): Eine Festlegung, die betriebliche Aktivitäten zusichert, Bedingungen festlegt oder diese kontrolliert.

Ableitung (*engl.: derivation*): Eine Aussage, die Wissen aus bekannten betrieblichen Fakten ableitet.

2.3.1.1 Strukturelle Zusicherung

Strukturelle Zusicherungen beschreiben die statischen Aspekte einer Unternehmung. Sie definieren die grundlegenden Konzepte und deren Beziehungen untereinander. Ein Beispiel für eine Strukturelle Zusicherung ist

Eine Rechnung besteht aus einer Rechnungsanschrift und Rechnungsposten.

Im Großen und Ganzen decken Strukturelle Zusicherungen den Teil einer Unternehmsmodellierung ab, die mittels eines Entity-Relationship-Modells [Che76] geschehen kann. Diese Art von Geschäftsregeln können nochmals weiter in *Terme* und *Fakten* unterteilt werden. Terme sind Begriffe oder Aussagen mit einer besonderen Bedeutung für das Unternehmen (Entity). Die Business Rule Group nimmt nochmals eine weitere Unterscheidung der Terme vor. Das sind zum einen die Common Terms – allgemeingültige Begriffe aus dem Alltag, sowie die Business Terms – Begriffe aus der Geschäftswelt. Fakten dienen dazu, die Terme in Beziehung zu setzen, sie decken die Relationships im ER-Modell ab.

2.3.1.2 Aktionsanweisung

Aktionsanweisungen sind Regeln, die den dynamischen Aspekt einer Unternehmung spezifizieren. Während die Strukturellen Zusicherungen Möglichkeiten eröffnen, schränken Aktionsanweisungen die Ergebnisse von betrieblichen Aktivitäten ein. Aktionsanweisungen lassen sich in drei Typen unterscheiden:

Bedingung (*engl.: condition*): Eine Geschäftsregel, die eine Vorbedingung an eine Aktion knüpft, zum Beispiel

Wenn ein Kunde nicht im Zahlungsrückstand ist, dann nehme Bestellungen von ihm an.

Integritätszusicherung(*engl.: integrity constraint*): Eine Geschäftsregel, die die Integrität einer Unternehmung zusichert. Im Gegensatz zur Bedingung muss eine Integritätszusicherung zu *jedem* Zeitpunkt erfüllt sein, zum Beispiel

Jeder Mitarbeiter hat einen gültigen Arbeitsvertrag.

Autorisierung (*engl.: authorization*): Eine Geschäftsregel, die Rechte innerhalb einer Unternehmung vergibt oder einschränkt, zum Beispiel

Nur ein Administrator kann das Netzwerk konfigurieren.

2.3.1.3 Ableitung

Eine Ableitung ist ein Instrument zur Generierung neuer Fakten aus einer bekannten Faktenmenge. Eine Ableitung kann entweder eine mathematische Berechnung oder eine logische Schlussfolgerung sein. Ein Beispiel für eine mathematische Berechnung ist

Die Rechnungssumme setzt sich aus der Summe der Rechnungsposten zuzüglich der Versandkosten zusammen.

2.3.2 Geschäftsregeln im Softwareentwicklungsprozess

Geschäftsregeln sind nicht auf einen bestimmten Teil eines Unternehmens beschränkt. Ihr Geltungsbereich schließt auch die Geschäftsprozesse ein, die durch oder mit Hilfe von Informationssystemen umgesetzt werden sollen. Während des Softwareentwicklungsprozesses sind die Strukturellen Zusicherungen[4] von besonderer Wichtigkeit, sie legen die Konzepte und die Fakten einer Unternehmung fest. Die Änderungshäufigkeit der Strukturellen Zusicherungen ist relativ gering, was sie zum Beispiel als Grundlage zur Identifikation von Objekten und Klassen innerhalb des objektorientierten Entwicklungsprozesses geeignet erscheinen lässt.

[4]siehe Abschnitt 2.3.1.1

Die Informatik entwickelte in den letzten Jahren Metamodelle zur Modellierung von Informationssystemen. Die bekanntesten Vertreter sind hierfür das Entity-Relationsship-Modell [Che76] und die Unfied Modelling Language [Obj00]. Mittels dieser Modelle lassen sich eine Vielzahl der Geschäftsregeln abbilden. Abbildung 2.9 zeigt das Beispiel aus Abschnitt 2.3 als UML-Diagramm, die Zusicherung, dass der Wert der Bestellung den Kreditrahmen des Käufers nicht übersteigen darf, wird mit Hilfe der Object Constraint Language [RMH+97] spezifiziert.

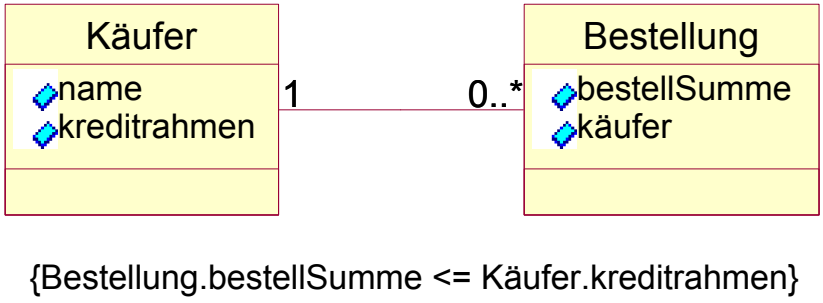

Abbildung 2.9: Beispielmodell einer Geschäftsregel in der UML

Das Hauptproblem, das bei der Entwicklung von Informationssystemen bezüglich der Geschäftsregeln auftritt, ist nicht, dass es besonders schwierig ist diese umzusetzen, sondern, dass es Geschäftsegeln geben kann, die sich sehr häufig ändern. Die Gesamtmenge der Geschäftsregeln ist nicht als statisch anzusehen, in Unternehmensbereichen wie zum Beispiel dem Marketing ist die Wahrscheinlichkeit sehr hoch, dass sich die Regelmenge in kurzer Zeit stark ändert.

2.4 Wissensbasierte Systeme

Was „Wissen“ ist und wie „Wissen“ beschaffen ist beschäftigt die Philosophie seit langer Zeit. Die wohl ersten Philosophen, die sich mit dieser Fragestellung intensiv auseinander gesetzt haben, waren Plato und Aristoteles.

In der Philosophie werden **Epistemologie** (*Erkenntnistheorie – „Wie wir Kenntnis von der Wahrheit erlangen und ob diese Kenntnis auch verlässlich und 'wahr' ist.“* [Wat85]) und **Ontologie** (*„Ontologie ist also die Wissenschaft, die Theorie oder die Untersuchung des Seins, bzw. die Erforschung dessen, was ist, 'wie es ist' usw.“*[von94]) unterschieden. Der Unterschied, sofern eine Trennung der verschiedenen Sichtweisen überhaupt möglich ist, wird von Heinz von Foerster in [von93] am treffendsten herausgearbeitet: *„Ontologie erklärt die Beschaffenheit*

der Welt; Epistemologie erklärt die Beschaffenheit unserer Erfahrung von dieser Welt.“

Wissen ist Bestandteil einer Hierarchie. Abbildung 2.10 zeigt die Einordnung von Wissen in diese Hierarchie.

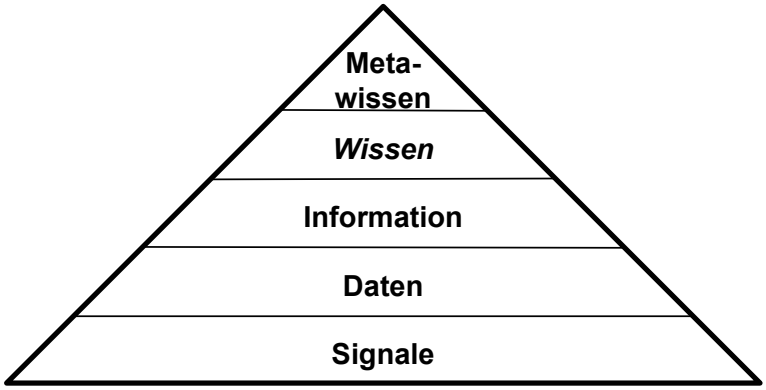

Abbildung 2.10: Hierarchische Einordnung von Wissen

Als Metawissen[5] bezeichnet man *„das Wissen über das Wissen“*. Darunter fallen die Ergebnisse der Epistemologie sowie das Wissen, das beschreibt, wie Wissen anzuwenden ist.

In der Informatik werden Systeme, die aufgrund von Wissen eine bestimmte Situation (definiert durch eine Faktenmenge) auswerten und daraus Schlussfolgerungen ziehen können, als wissensbasiert bezeichnet.

Wissensbasierte Systeme werden oft auch als Expertensysteme bezeichnet [GR98], da die zugrunde liegenden Technologien in erster Linie zur Simulation von Experten benutzt werden. Dabei geht man von der Vorstellung aus, dass ein Experte seine Problemlösungen aus Einzelkenntnissen zusammensetzt, die er selektiert und in passender Anordnung verwendet [Pup88]. Ziel dieser Systeme ist es, das Wissen eines Experten und deren Anwendung zu jeder Zeit einer großen Benutzergruppe zur Verfügung zu stellen.

Innerhalb eines wissensbasierten Systems lassen sich folgende Aufgabenbereiche charakterisieren:

Wissenserwerb oder Wissenakquisition: Wie kann menschliches Wissen in eine rechnerverarbeitbare Form gebracht werden.

[5]meta – *griech.: gibt eine fundamentale Analyse oder Erklärung über sich selbst*

Wissenrepräsentation: Welche Datenstrukturen gibt es, mit deren Hilfe man Wissen effizient in Rechnern speichern und verarbeiten kann.

Inferenz: Wie können Schlussfolgerungen aus dem im Rechner gespeicherten Wissen hergeleitet werden.

Diese Aufgabenbereiche gehen in die verschiedenen Komponenten eines wissensbasierten System ein. Abbildung 2.11 zeigt die verschiedenen Komponenten eines wissensbasierten Systems.

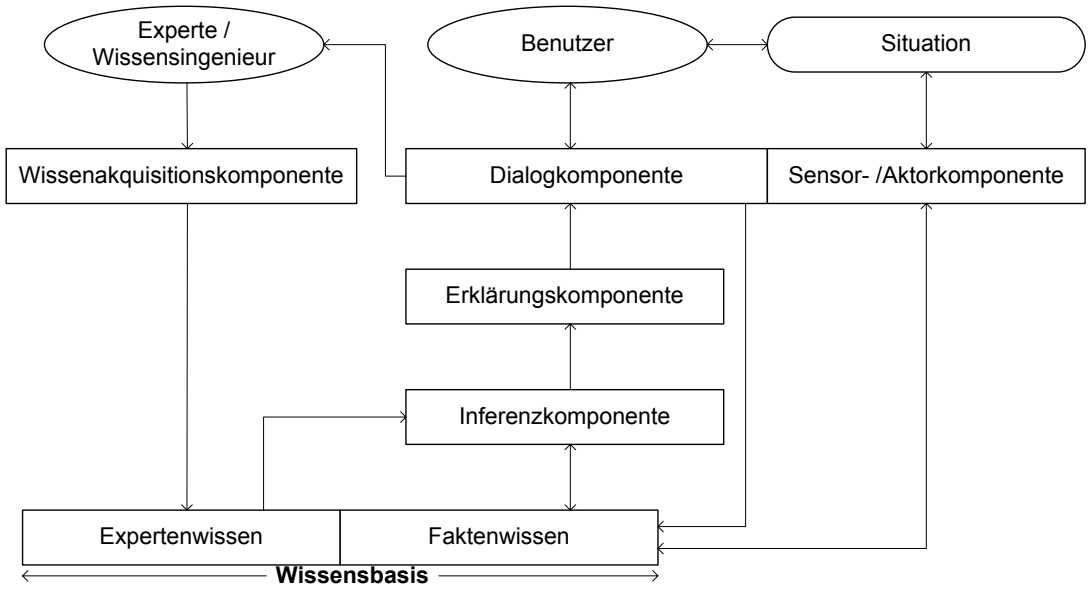

Abbildung 2.11: Komponenten eines wissensbasierten Systems

Die *Wissensakquisitionskomponente* überführt das Wissen des Experten in die interne Wissensrepräsentation des Systems, dabei wird die *Dialogkomponente* als Mensch-Maschine-Schnittstelle benutzt. Die geeigneten Formen der Wissensakquisition werden in Abschnitt 2.4.1 dargestellt. Das so überführte Wissen ist Bestandteil der *Wissensbasis* des Systems. Diese Teilmenge wird auch innerhalb des Systems als *Expertenwissen* bezeichnet. Die weitere Teilmenge der Wissenbasis bildet das sogenannte *Faktenwissen*. Das Faktenwissen ist die interne Repräsentation der aktuellen Situation, diese kann entweder durch die Dialogkomponente, durch Eingaben des Benutzers oder durch einen Regelkreis mittels einer *Sensor-/Aktorkomponente* modifiziert werden. Die *Inferenzkomponente* wendet das Expertenwissen auf die durch das Faktenwissen repräsentierte Situation an. Die Inferenzkomponente entscheidet, welcher Aspekt des Expertenwissens in einer bestimmten Situation angewendet wird. Dieser Entscheidungsprozess kann mittels der *Erklärungskomponente* dem Benutzer transparent gemacht werden.

2.4.1 Wissensakquisition

Die Wissenakquisition besteht aus der Identifikation und der Formalisierung des Wissens. Zur Wissensakquisition wird auch die Wartung des dem wissensbasierten Systems zugrunde liegenden Wissens gezählt. In vielen Fällen liegt das Wissen in einer nicht computerverarbeitbaren Form vor, es existiert nur im Kopf des „Experten“. In diesem Fall gibt es zwei Möglichkeiten das Wissen in eine systemgerechte Form zu bringen:

indirekte Wissensakquisition: Ein sogenannter Wissensingenieur (*engl.: knowledge engineer*) interviewt den Experten, identifiziert das Wissen und formalisiert es. Der Wissensingenieur dient als Mittler zwischen dem Experten und dem wissensbasierten System. Diese Methode hat sich als problematisch herausgestellt [Pup88], da der Wissensingenieur nur sehr schwer Lücken innerhalb des Expertenwissen entdecken kann und Wissen nur sehr schwer und teilweise auch nur unvollständig verbal beschrieben werden kann.

direkte Wissensakquisition: Der Experte formalisiert das Wissen selbst. Dazu muss das wissensbasierte System eine komfortable Komponente zur Verfügung stellen, die es dem Experten ermöglicht, auf einfache Art und Weise sein Wissen zu formulieren.

In manchen Fällen kann es möglich sein, dass ein wissensbasiertes System sein Wissen **automatisch** akquirieren kann. Das System erwirbt sein Wissen durch selbstständiges Extrahieren aus Falldaten oder verfügbarer Literatur. Für das Extrahieren des Wissens aus der Datenbasis ist ein Lernverfahren zu implementieren, das die Rolle des Wissensingenieurs übernimmt. Das Lernverfahren identifiziert Wissen aus den Daten und bildet es in eine entsprechende Wissensrepräsentation ab.

Eine besondere Rolle spielt bei der Wissensakquisition das Metawissen. Das Metawissen steuert die Anwendung des Wissens. Es entscheidet darüber, welcher Wissensaspekt in einer bestimmten Situation von besonderer Wichtigkeit ist. Metawissen unterteilt das Wissen eines wissensbasierten Systems in verschiedene Problemlösungsstrategien.

2.4.2 Wissensrepräsentation

Die Art und Weise, wie ein wissensbasiertes System das ihm zugrunde liegende Wissen repräsentiert, ist von entscheidender Wichtigkeit. Jedes System ist in den meisten Fällen nur für eine bestimmte Form von Wissensrepräsentation ausgelegt. Die folgenden Abschnitte geben eine Übersicht über die wichtigsten Wissensrepräsentationsformen.

2.4.2.1 Logik

Die wohl am besten erforschte Wissensrepräsentationart ist die **Aussagen-** und die **Prädikatenlogik** erster und zweiter Ordnung. In der praktischen Informatik wird diese aber kaum zur Wissensrepräsentation eingesetzt, da die Aussagenlogik und die Prädikatenlogik erster Ordnung zu ausdrucksschwach sind [GR98], die Prädikatenlogik zweiter Ordnung nur sehr schwer zu implementieren ist [Pup88].

2.4.2.2 Semantische Netze

Semantische Netze wurden um die Jahrhundertwende von Psychologen benutzt, um die Repräsentationsstrukturen innerhalb des Gehirns graphisch zu modellieren. Im Bereich der Künstlichen Intelligenz wurden diese als erstes zur Simulation des menschlichen Sprachverständnisses eingesetzt.

Semantische Netze bestehen aus Knoten und Kanten. Die Knoten repräsentieren die Fakten, die Kanten zwischen diesen Knoten bilden die (binären) Relationen zwischen diesen ab (siehe Abbildung 2.12). Aus diesen Relationen können neue Relationen hergeleitet werden. In Abbildung 2.12 können die Geschwisterrelationen die zwischen Sabine, Daniela und Johannes bestehen durch die Relation der Vater- bzw. Mutterschaft von Brigitte und Friedbert geschlossen werden.

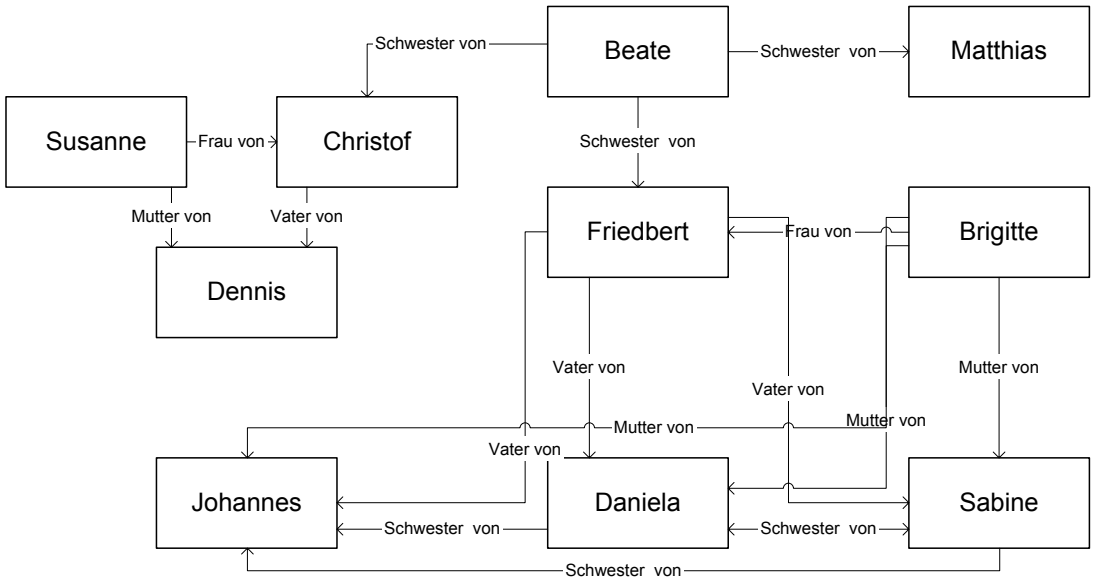

Abbildung 2.12: Semantisches Netz

2.4.2.3 Frames

Frames wurden von Marvin Minsky [Min74] eingeführt, um Wissen strukturieren zu können. Ein Frame repräsentiert dabei eine stereotype Situation, zum Beispiel „*in einem Wohnzimmer sein*“ oder „*zu einem Kindergeburtstag gehen*“. An ein Frame sind verschiedene Informationen gebunden, zum einen die stereotype Situation beschreibende Informationen zum anderen Informationen, wie das Frame „benutzt“ werden soll, was als nächstes passieren wird und wie verfahren werden soll, wenn sich diese Erwartungen nicht erfüllen.

Die informationstechnische Überführung besteht aus einem Frame (Objekt), die die Filler (Werte) in Slots (Attribute) hält. Wie beim objektorientierten Paradigma gibt es generische Frames (abstrakte Superklassen), die als „typisch“ angesehen werden aber keine konkreten Fakten repräsentieren. Als Beispiel soll hier ein Automobil-Frame dargestellt werden:

Frame	
Slot	**Filler**
name	Automobil
spezialisierung von	Frame
typ	(Fließheck, Kombi, Cabrio)
hersteller	(VW, Opel, Ford, Mercedes, BMW)
anzahl räder	4
getriebe	(Automatik, Schaltung)

Ein spezifisches Automobil-Frame kann zum Beispiel folgendes Aussehen besitzen:

Frame	
Slot	**Filler**
name	Frank's Auto
spezialisierung von	Automobil
typ	Kombi
hersteller	Mercedes
farbe	schwarz
besitzer	Frank Schneider

Als eines der größten Probleme bei der Implementierung von framebasierten Systemen hat sich der Umgang mit Vorgabewerten herausgestellt [GR98]. Typisch für ein Automobil ist, dass es vier Räder besitzt. Zur Automobilklasse sollte aber dennoch zum Beispiel ein dreirädriger Kabinenroller gehören. Es ist kein untypischer Fall, dass ein spezifischer Frame zu einem generischen Frame gehört, obwohl das spezifische Frame dem Vorgabewert des generischen Frames widerspricht.

2.4.2.4 Produktionsregeln

Produktionsregelwerke sind in der Informatik ein vielbenutzes Mittel um Wissen zu repräsentieren. In der theoretischen Informatik dienen sie als Grundlage vieler Beweise (zum Beispiel: Turing Maschine, Markov Algorithmen). In der praktischen Informatik werden sie hauptsächlich zur Definition von Sprachen benutzt, wobei hier die Backus-Naur Form [Nau60] zu einer der meistbenutzten Techniken gehört.

Eine Produktionsregel (*engl.: production rule*) beinhaltet eine **Vorbedingung** (Prämisse, left hand side) und eine **Aktion** (Konklusion, right hand side), sie hat folgende Struktur:

WENN <*Vorbedingung*>, **DANN** <*Aktion*>

Die Vorbedingung einer Regel besteht aus einem logischen Ausdruck, der in den meisten Fällen ausschließlich konjunktiv verknüpfte Aussagen enthält [DG99, S. 979]. Die logischen Ausdrücke beziehen sich auf die in der Wissensbasis vorhandenen oder nicht vorhandenen Fakten.

Eine Aktion enthält Anweisungen, die in den meisten Fällen die Wissensbasis des Regelsystems manipulieren.

Ist die Vorbedingung einer Regel erfüllt, so werden die Anweisungen der Aktion der Regel ausgeführt.

2.4.3 Inferenz

Als Inferenz versteht man den Prozess der Schlussfolgerung. Inferenz „berechnet“ anhand von gegebenen Fakten neue Informationen bzw. Fakten. Im Folgenden soll ein Überblick über die verschiedenen Methoden der Inferenz gegeben werden:

Deduktion: Logisches Schließen, bei dem eine Schlussfolgerung aus den Vorbedingungen abgeleitet wird.

Induktion: Inferenz, die einen speziellen Fall in eine allgemeine Aussage überführt.

Intuition: Eine nicht bewiesene Theorie. In der Informatik wird diese zum Beispiel bei Künstlichen Neuronalen Netzen angewendet. Von den „gelernten“ Eingabe-/Ausgabekombinationen schlussfolgert ein Künstliches Neuronales Netz eine Ausgabe für eine beliebige Eingabe.

Heuristik: Die Benutzung von „Daumenregeln“ zur Lösungsfindung. Wie bei der Intuition kann im Allgemeinen die Richtigkeit dieser Daumenregeln nicht bewiesen werden.

Generieren & Testen (*engl.: trial and error*): Generiere Lösungsmöglichkeiten und überprüfe diese auf Richtigkeit

Abduktion: Umkehrung der Deduktion, von einer gegebenen Schlussfolgerung wird auf die diese wahrmachende Vorbedingungen geschlossen.

Ermangelungsschließen (*engl.: default inference*): Das Ermangelungsschließen erlaubt es nicht vorhandenes spezifisches Wissen durch „generelles" Wissen zu approximieren.

Autoepistemische Inferenz (*engl.: autoepistemic inference*): Die autoepistemische Inferenz bezieht auch das „Nichtwissen" in den Schlussfolgerungsprozess ein. Die autoepistemische Logik nutzt das Wissen über das Wissen bzw. über das Nichtwissen.

Nichtmonotone Inferenz: Schlussfolgerungsprozess, bei dem sich vorhandenes Wissen als unwahr herausstellen kann, wenn ein Beweis gefunden wurde, der dieses beweist.

Analogie: Bei der Analogie wird davon ausgegangen, dass eine bekannte Problemlösung auf eine Problemstellung anwendbar ist, sofern die Problemstellungen „vergleichbar" sind bzw. eine „ähnliche" Struktur besitzen.

Die Inferenzmechanismen, die in regelbasierten Systemen zum Einsatz kommen werden im folgenden Abschnitt dargestellt.

2.4.3.1 Inferenz in Regelsystemen

Im Abschnitt 2.4.2.4 wurde die Struktur von Produktionsregeln beschrieben. In diesem Abschnitt werden die Inferenzmechanismen eines Regelsystems dargestellt.

Die wichtigste Klassifikation der Inferenzmechanismen in Produktionsregelsystemen stellt die Richtung der Regelanwendung dar.

Bei der **Vorwärtsverkettung** (*engl.: forward chaining*) wird überprüft, welche Regeln anhand der erfüllten Vorbedingungen angewendet werden können – dies entspricht der Deduktion.

Sind die Vorbedingungen mehrerer Regeln erfüllt, bilden diese eine so genannte *Konfliktmenge*. Der Konflikt (welche Regel wird als erstes ausgeführt) wird mit einer *Konfliktlösungsstrategie* aufgelöst. Ändert der Aktionsteil der ausgeführten Regel die Fakten in der Form, dass die Vorbedingungen von Regeln in der Konfliktmenge nicht mehr erfüllt sind, werden diese aus der Konfliktmenge gelöscht.

Eine häufig angewandte Konfliktlösungsstrategie ist die Zuordnung von Prioritäten zu Regeln. Befinden sich mehrere Regeln mit unterschiedlicher Priorität in der Konfliktmenge, so wird die Regel mit der höchsten Priorität ausgeführt. Befinden sich mehrere Regeln mit der höchsten Priorität in der Konfliktmenge, so wird eine sekundäre Konfliktstrategie angewandt. Als Kriterium für die sekundäre Konfliktstrategie kommt häufig der Eintrittszeitpunkt der Regel in die Konfliktmenge zum Einsatz. In diesem Fall wird die Regel, die zuletzt oder zuerst in die Konfliktmenge eingefügt wurde, ausgeführt.

Bei der **Rückwärtsverkettung** (*engl.: backward chaining*) gibt man das zu erreichende Ziel an. Dabei werden die Aktionen mit diesem Ziel verglichen, wird eine Aktion gefunden, die dieses Ziel herbeiführt, wird das Ziel durch die Vorbedingung der zu dieser Aktion gehörenden Regel ersetzt. Die Rückwärtsverkettung beantwortet Fragen wie „Was muss passieren, um diese Situation herbeizuführen?“ oder „Wie konnte diese Situation entstehen?“ (Abduktion).

2.4.4 Rete-Algorithmus

Der Rete-Algorithmus wurde von Charles L. Forgy 1982 in [For82] vorgestellt. Der Algorithmus ist eine effiziente Methode [NGR88], um eine Menge von Mustern[6] (*engl.: pattern*) mit einer Menge von Objekten zu vergleichen. Der Rete-Algorithmus findet alle Objekte aus der Objektmenge, die diesen Mustern genügen.

Der Rete-Algorithmus gehört zu der großen Menge von Algorithmen, die eine Vorverarbeitung vorsehen um das eigentliche Ziel (Finden aller Objekte, die den Mustern genügen) effizienter erreichen zu können. Dieser Vorverarbeitungsschritt lohnt sich genau dann, wenn folgende Bedingung erfüllt ist:

$$m \cdot C > C_V + m \cdot C'$$

wobei

m	die erwartete Anzahl der nachfolgenden Zielvorgängen,
C	der Aufwand für den Zielvorgang ohne Vorverarbeitung,
C_V	der Aufwand für die Vorverarbeitung und
C'	der Aufwand für den Zielvorgang mit Vorverarbeitung

ist.

In dem Vorverarbeitungsschritt baut der Rete-Algorithmus zwei Netzwerke auf. Das ist zum einen das Pattern Netzwerk, das Fakten auf ihre statischen Eigen-

[6] Beschreibung von Objekteigenschaften

schaften hin untersucht und das Join Netzwerk, das faktenübergreifende Eigenschaften sicherstellt.
Bei Regelsystemen werden im Allgemeinen beim Inferenzvorgang nur sehr wenige Fakten (Objekte) hinzugefügt, geändert oder gelöscht. Diese Eigenschaft nutzt der Rete-Algorithmus aus, in dem Ergebnisse im Muster/Objekt-Vergleich gespeichert werden. Dieses Vorgehen ermöglicht es, nach einem Inferenzvorgang nur diese Fakten neu zu untersuchen, die sich geändert haben.

Diese Vorgehensweise reduziert den Aufwand von $\mathbf{O}(R \cdot F^P)$ auf $\mathbf{O}(R \cdot F \cdot P)$, wobei R die Anzahl der Regeln, P die durchschnittliche Anzahl von Mustern pro Regel und F die Anzahl der Objekte in der Objektmenge ist [FH01, S. 101 – 102].

Im Folgenden werden die beiden Netzwerke des Rete-Algorithums beschrieben sowie ein Beispiel eines kompletten Rete-Netzwerkes dargestellt.

2.4.4.1 Pattern Netzwerk

Das Pattern Netzwerk ist ein Baum, bestehend aus einem Wurzelknoten und aus Knoten, zu denen jeweils nur eine Kante hinführt. Die mit der Wurzel verbundenen Knoten unterscheiden die Objekte nach ihrem Typ. Alle weiteren Nachfolgeknoten untersuchen Muster, bei denen ein Objekt als „autonom“ betrachtet werden kann.

Als Beispiel sei eine Klasse A mit den Attributen a1 und a2 und die Regel

WENN A1.a2 = 2 und A1.a1 > A1.a2 **DANN** ...

gegeben.[7]

Das durch diese Regel resultierende Pattern Netzwerk, wird in Abbildung 2.13 gezeigt.

Alle Blätter[8] des Baumes besitzen ein sogenanntes *alpha memory*, das die Menge der Objekte, die das Blatt erreicht haben, speichert.

2.4.4.2 Join Netzwerk

Das Join Netzwerk baut auf dem Pattern Netzwerk auf. Als Eingänge des Join Netzwerks dienen die Blätter des Pattern Netzwerks. Die Terminalknoten des Join Netzwerks repräsentieren eine erfüllte Regelvorbedingung. Alle Knoten im Inneren (join node) des Join Netzwerks haben zwei Eingänge und mindestens

[7] Zur Vereinfachung werden in den Regeln Variablen mit <Typ><natürliche Zahl> benannt.
[8] „Muster erfüllt“-Knoten

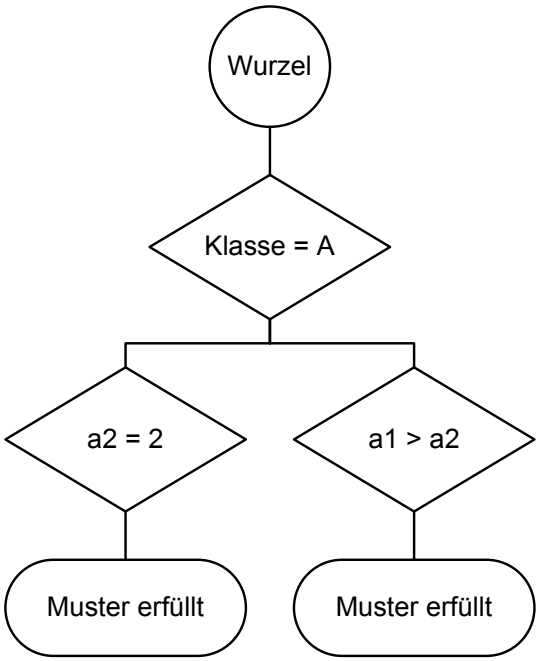

Abbildung 2.13: Pattern Netzwerk

einen Ausgang. Jeder der inneren Knoten repräsentiert ein Teilmuster einer Regelvorbedingung, so dass eine Vorbedingung, die aus N Mustern besteht, durch ein Join Netzwerk von $N - 1$ Knoten realisiert wird.

Als Beispiel sei die Klassen A mit dem Attribut a1 und die Klasse B mit dem Attribut b1 sowie die Regel

WENN A1.a1 = B1.b1 **DANN** ...

gegeben.[9]

Abbildung 2.14 zeigt das daraus resultierende Pattern/Join Netzwerk.

Im Gegensatz zum Pattern Netzwerk besitzt beim Join Netzwerk jeder Knoten einen Speicher, das *beta memory*. Im beta memory werden die Menge der Objektpaare gespeichert, die die Bedingung, die durch den Knoten repräsentiert wird, erfüllen.

Zur Optimierung des Join Netzwerkes werden Knoten von Mustern, die in mehreren Regeln vorkommen, wieder verwendet. Abbildung 2.15 zeigt diese Optimierung.

[9]Zur Vereinfachung werden in den Regeln Variablen mit <Typ><natürliche Zahl> benannt.

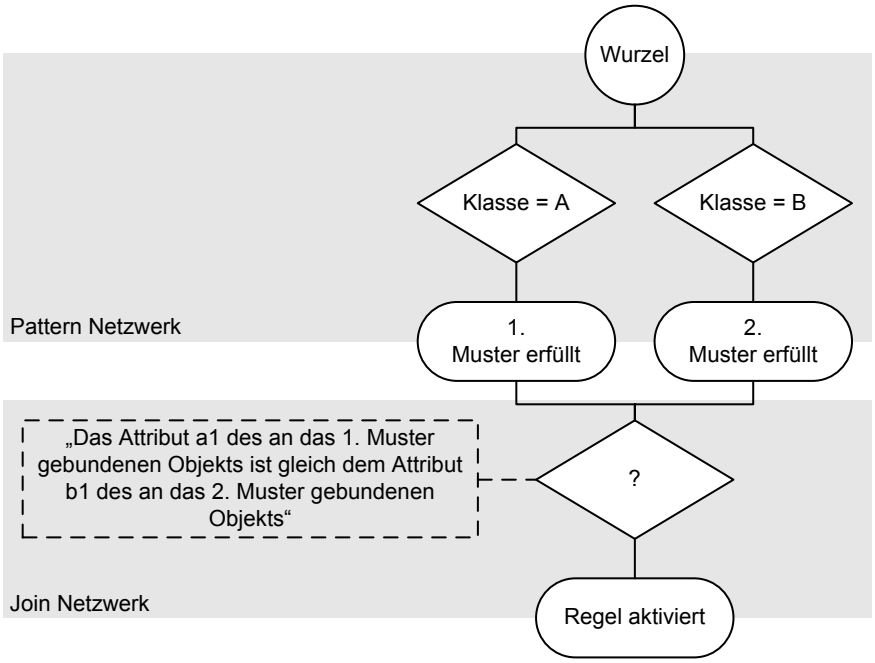

Abbildung 2.14: Pattern- und Join Netzwerk

2.4.4.3 Beispiel

Abschließend soll hier ein komplettes Beispiel eines Rete Netzwerkes dargestellt werden. Es seien die Klassen A mit den Attributen a1 und a2, B mit dem Attribut b1 sowie C mit Attribut c1 gegeben. Das Regelwerk besteht aus den folgenden Regeln

WENN A1.a1 = B1.b1 und A1.a1 = A1.a2 **DANN** ...
WENN A1.a1 = B1.b1 und B1.b1 = C1.c1 **DANN** ...

Als Fakten dienen folgende Objekte:

- **A**(a1 = 1, a2 = 1)
- **A**(a1 = 1, a2 = 2)
- **B**(b1 = 1)
- **C**(c1 = 1)

Abbildung 2.15 zeigt das daraus resultierende Netzwerk bestehend aus dem Pattern und dem Join Netzwerk. Die Belegungen des alpha bzw. des beta memories wird durch die mit gepunktete Linien verbundenen Boxen dargestellt.

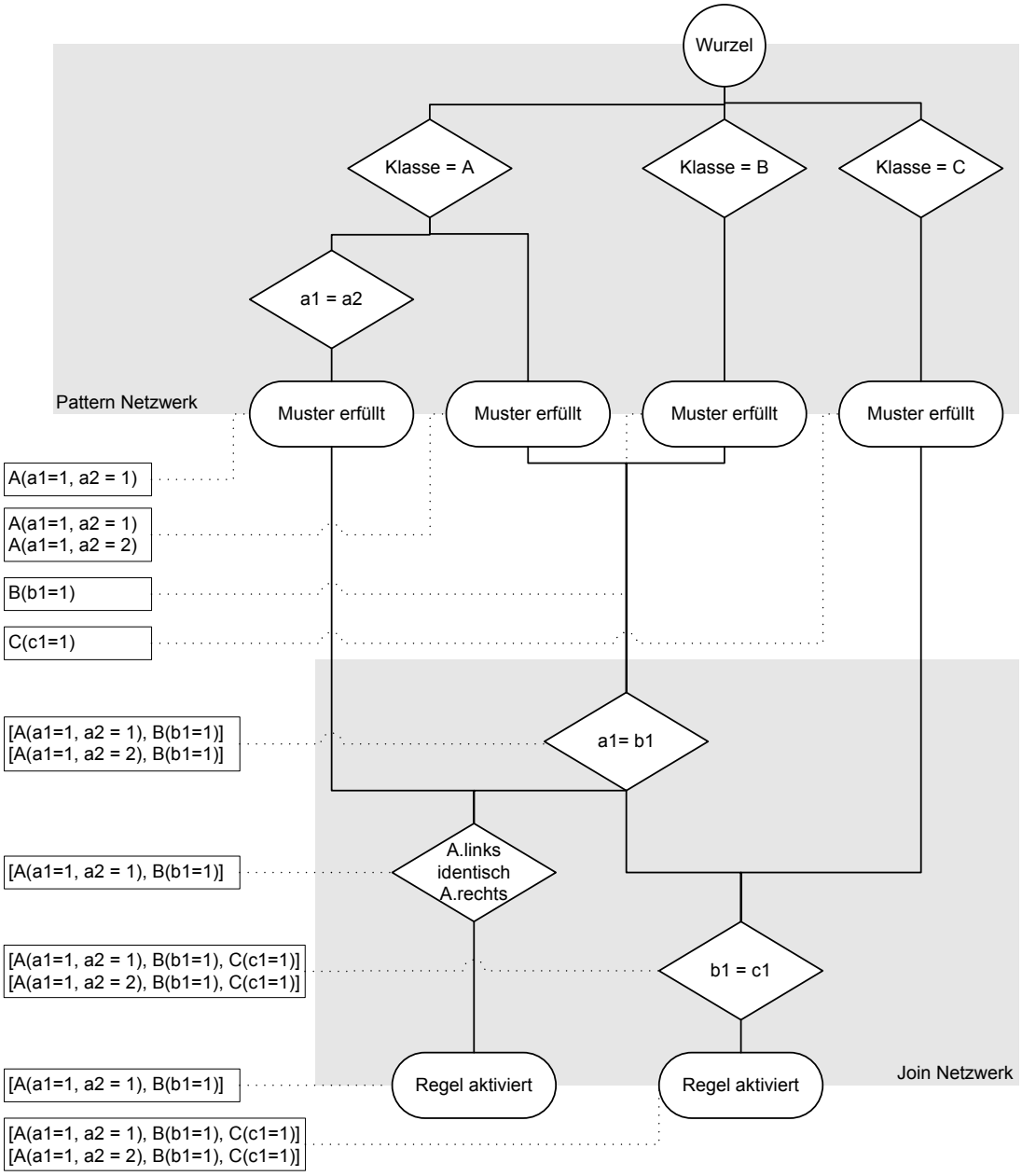

Abbildung 2.15: Komplettes Netzwerk des Rete-Algorithmus mit der Speicherbelegung des alpha- und des beta-memories

Kapitel 3

Konzeption

3.1 Übersicht

Objektbasierte Softwaresysteme, zu denen auch INTERSHOP enfinity gehört, bestehen aus einer Menge von verschiedenen Objekten, die den aktuellen Zustand des Systems repräsentieren. Dieser Zustand soll dem Regel- und Aktionenprozessor als Situationsbeschreibung dienen, er geht als Teilmenge des Faktenwissens in das Expertensystem ein.

Das Expertensystem überwacht diese Objektmenge auf Anwendbarkeit der definierten Regeln. Werden Objekte gefunden, die die Vorbedingungen von Regeln erfüllen, so werden die in den Regeln definierten Aktionen ausgeführt.

INTERSHOP enfinity stellt das objekt-/prozessorientierte Paradigma zur Realisierung der Geschäftsprozesse zur Verfügung. Durch die folgende Konzeption soll dem enfinity-System ein weiteres Paradigma - das objekt-/regelbasierte Paradigma - zur Realisierung von Geschäftsprozessen hinzugefügt werden.

3.1.1 Auswahl der Objektmenge

Moderne Softwaresysteme besitzen eine sehr hohe Komplexität, die oftmals zu einer Anzahl von mehreren hundert Millionen Objekten führt. Aus technischer Sicht ist es sicherlich möglich, eine derartig hohe Anzahl von Objekten zu überwachen, jedoch ist der Betrieb eines solchen Systems so teuer, dass es für den realen Einsatz nicht geeignet ist. Aus dieser Tatsache folgt, dass die Anzahl der Objekte auf ein vernünftig erscheinendes Maß reduziert werden muss.

Wie im Abschnitt 2.1.2.2 beschrieben, basieren moderne E-Commerce-Systeme auf Business Objects, die die Objekte, auf denen Geschäftsprozesse operieren, repräsentieren. Das Ziel der Diplomarbeit ist es, einen Regel- und Aktionenprozessor in ein E-Commerce-System zu integrieren, der **genau** auf dieser Menge

von Business Objects operieren kann.

Business Objects sind innerhalb des enfinity-Systems Teil der Menge der Entity-EJBs[1], also Objekte, die persistent in einem Datenhaltungssystem gehalten werden. Diese Objekte werden mittels Transaktionen geändert, so wird sichergestellt, dass sich die Objektmenge in einem konsistenten Zustand befindet.

Abbildung 3.1 zeigt das zugrunde liegende Grobkonzept mit der Gesamtobjektmenge und der Menge der Business Objects.

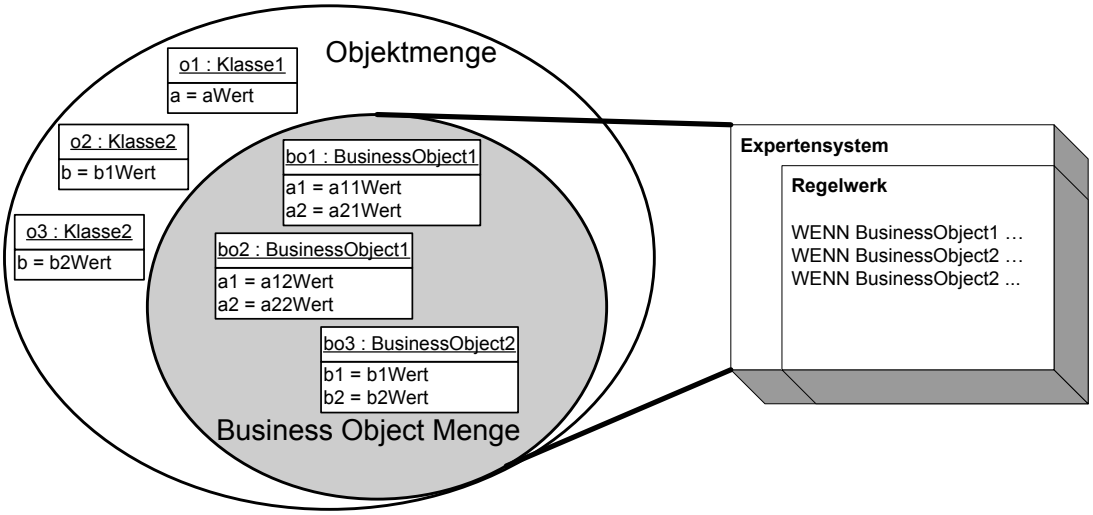

Abbildung 3.1: Vereinfachtes Konzept mit den Objektmengen

3.1.2 Zustandsänderung

Eine Zustandsänderung tritt ein, wenn eine Transaktion erfolgreich abgeschlossen wurde und mittels dieser Transaktion Attribute von Geschäftsobjekten geändert wurden oder Geschäftsobjekte erstellt oder gelöscht wurden. Aus den Transaktionsparadigmen (siehe Tabelle 2.2) folgt, dass die daraus resultierenden Zustände der Objektmenge konsistent sind. Aus Sicht des zu integrierenden Expertensystems ist der Zustand der Objektmenge zu jedem Zeitpunkt konsistent.

3.2 Migration im Vorgehensmodell

Die Migration im Vorgehensmodell wird durch eine Erweiterung des Vorgehensmodells mit Business Objects (siehe Abschnitt 2.1.2.2) erreicht.

[1] siehe Abschnitt 2.2.1.2

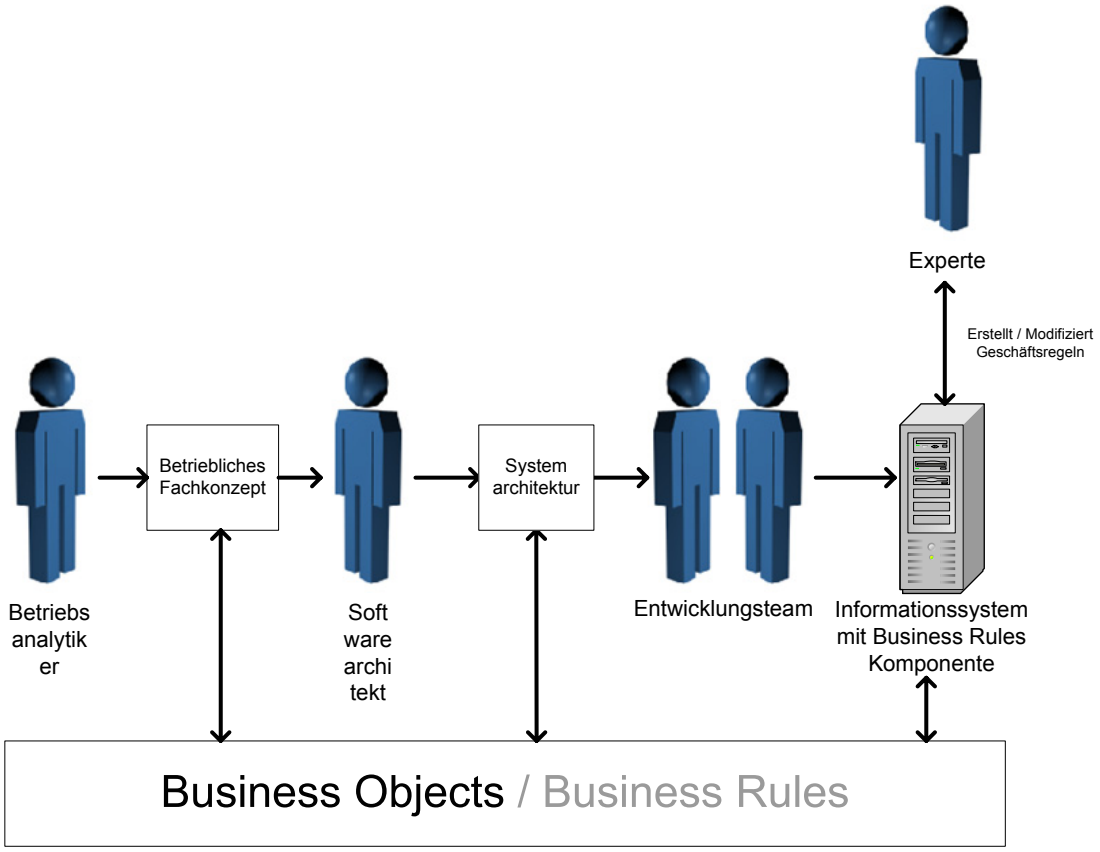

Abbildung 3.2: Das Vorgehensmodell mit der integrierten BusinessRules-Komponente

Abbildung 3.2 zeigt die Erweiterungen, die sich gegenüber dem Vorgehensmodell mit INTERSHOP enfinity[2] bzw. dem Vorgehensmodell mit Business Objects[3] (siehe Abbildung 2.2) ergeben.

Die Konzeption verfolgt die Strategie des generischen Grundkonzeptes (Phase I) und darauf aufbauend die benutzerfreundliche, aufgabenzentrierte und sichtspezifische Konzeption (Phase II). Diese Vorgehensweise ist beim objektorientierten Entwurf nicht unüblich, so beinhaltet ein Objekt in vielen Fällen die Attribute von unterschiedlichen Sichtweisen. Zum Beispiel sind in einem Produkt-Geschäftsobjekt die Eigenschaften des Gegenstandes (Farbe, Gewicht usw.) und die Eigenschaften, die notwendig sind, um ein Produkt in einem Katalog darzustellen, modelliert.

[2]siehe Abschnitt 2.2.2.3
[3]siehe Abschnitt 2.1.2.2

Die Erweiterungen, die sich im Vorgehensmodell durch die Integration der BusinessRules-Komponente ergeben, lassen sich in zwei Phasen einteilen:

Phase I: Die BusinessRules-Komponente wird in ein bestehendes System integriert. Die Geschäftsprozesse wurden mittels des Vorgehensmodells mit *Business Objects* erstellt. Mittels der integrierten BusinessRules-Komponente können nun die bestehenden Geschäftsprozesse manipuliert und geändert werden.

Diese Änderungen können einen weitgehenden Eingriff in das System bedeuten, so dass der Experte nicht nur Wissen über die implementierten Geschäftsprozesse haben muss, sondern auch über deren informationstechnische Realisierung. Der in Abbildung 3.3 gezeigte Experte muss das Wissen über diese beiden Domänen besitzen.

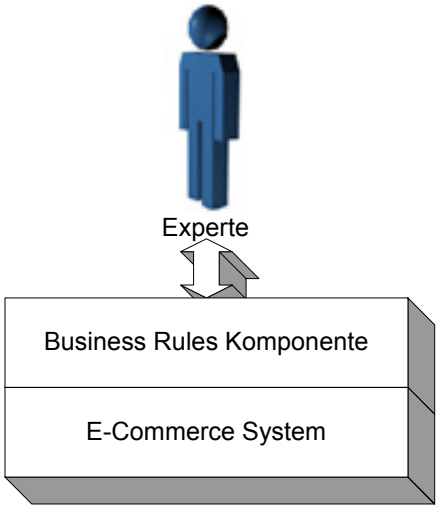

Abbildung 3.3: Schichtenmodell der Phase I

Phase II: Im Gegensatz zu Phase I wird in Phase II das Business Rules Konzept als Teil der Grundlage bei der Entwicklung des E-Commerce-Systems angesehen. Während des Entwicklungsprozesses wird unterschieden, welche Teilmengen der Geschäftsprozesse (z. B. Kundenbindung) durch Regeln realisiert werden. Für diese Teilmengen werden Sichten erstellt. Eine Sicht ermöglicht es, dass ein Experte (z. B. aus dem Bereich Marketing) das Verhalten des Systems ändern kann, ohne dass er Wissen über die Implementierung haben muss. Eine Sicht erfüllt die Funktion einer Wissensakquisitionskomponente[4] sowie einer Dialogkomponente[5]. Sie kann unter anderem

[4]siehe Abschnitt2.4
[5]siehe Abschnitt2.4

folgende Funktionalitäten beinhalten:

- **Tests und Operationen** für den Anwendungsbereich der Sicht
- **Reduktion der Komplexität**, z. B. durch Elimination der Geschäftsobjekte, die in dieser Sicht nicht benötigt werden.
- **Plausibilitätskontrolle** der Regeln
- **Assistenten** zur Benutzerführung

Abbildung 3.4 zeigt die zusätzliche Schicht mit einer Marketing-Sicht, einer Kundenbindungs-Sicht und einer Sicht zur Datenpflege.

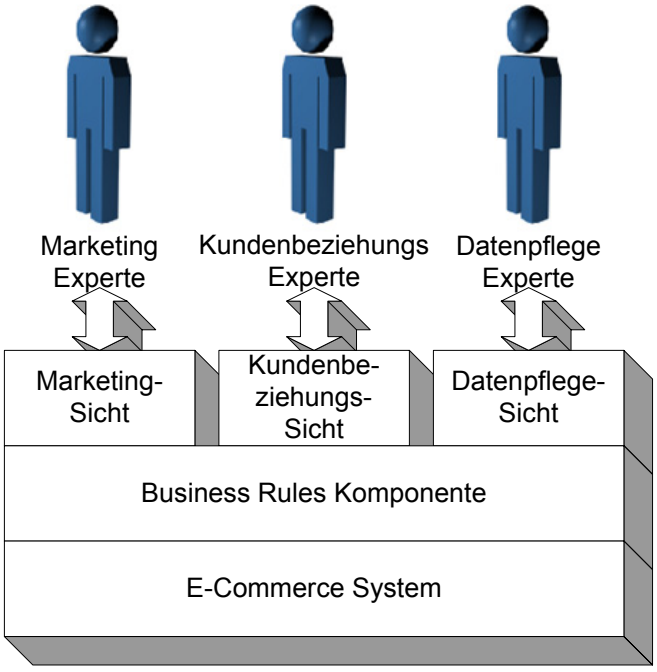

Abbildung 3.4: Schichtenmodell der Phase II

3.3 Systemarchitektur

Der Regel- und Aktionenprozessor wird als eigenständige Komponente (Business-Rules-Komponente) in die enfinity-Plattform integriert. Es ist nicht vorgesehen, dass es innerhalb einer Plattform mehrere Instanzen des Regel- und Aktionenprozessors geben kann. Eine enfinity-Plattform besteht aus zwei Servern, dem enfinity Transactivity Server (eTS) und dem enfinity Catalog Server (eCS). Diese

beiden Server müssen nicht notwendigerweise auf dem gleichen Computer laufen, sondern können auch auf zwei Computer verteilt eingesetzt werden. Sowohl der eTS als auch der eCS beherbergen einen Teil der Geschäftsobjekte. Diese Tatsache macht es zwingend notwendig, dass Funktionen des Regel- und Aktionenprozessors auch von anderen Rechnern aus aufgerufen werden können.

Um eine gewisse Skalierbarkeit zu erreichen, wird der Regel- und Aktionenprozessor so konzipiert, dass er ebenfalls auf einem eigenen Computer einsetzbar ist, d. h. der Regel- und Aktionenprozessor wird nicht direkt in den eTS bzw. den eCS eingebettet. Der so neben dem eTS und eCS entstandene Server heißt BusinessRules-Server.

Abbildung 3.5 zeigt die funktionalen Komponenten einer enfinity-Plattform mit dem BusinessRules-Server, der die BusinessRules-Komponente beinhaltet. Die BusinessRules-Komponente ist die Implementierung des Regel- und Aktionenprozessors.

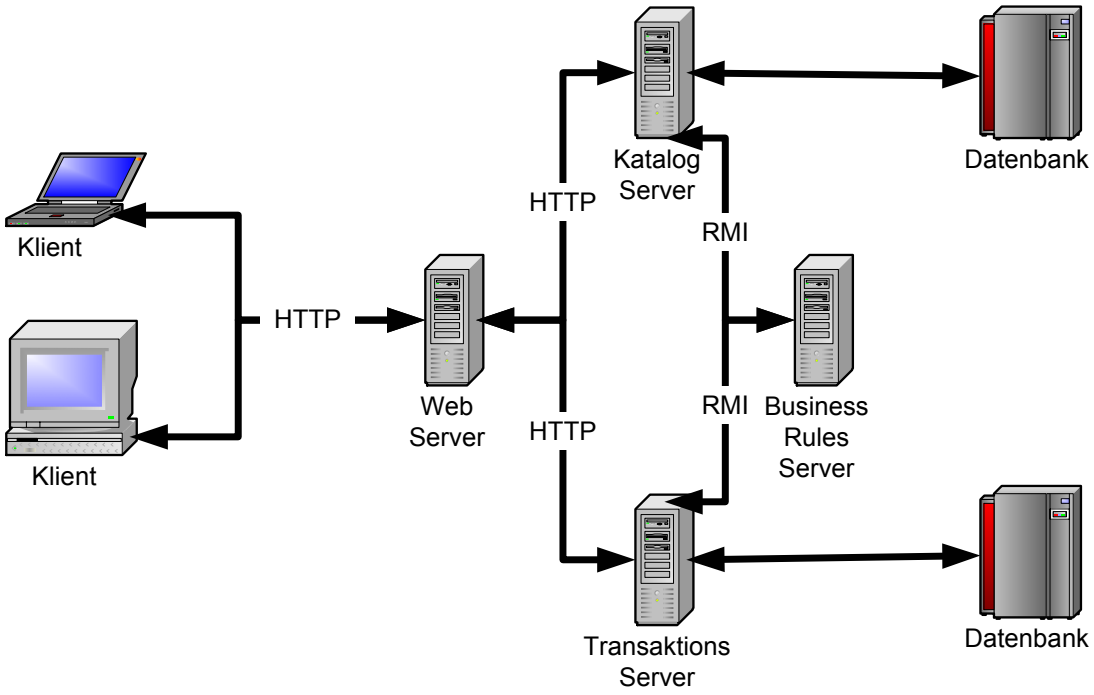

Abbildung 3.5: Die funktionalen Komponenten einer enfinity-Site mit dem BusinessRules-Server

Innerhalb des Schichtenmodells wird die BusinessRules-Komponente in der Geschäftsschicht angesiedelt. Dabei werden folgende Änderungen und Erweiterungen innerhalb der Geschäftsschicht entwickelt:

Pipeline-Ebene: Pipelets und Pipelines zur Steuerung der BusinessRules-Komponente, zur Synchronisation und zum Modifizieren des Regelwerkes. Die Pipelets und die Pipelines ermöglichen den Zugriff von der Anfrageabarbeitung aus auf die Erweiterungen des Geschäfts-Framework. Die Ergebnisse dieser Zugriffe können mittels des Antwort- / Präsentationsmechanismus' des enfinity-Systems dem Benutzer präsentiert werden.

Geschäfts-Framework-Ebene: Erweiterung des Frameworks zur

- Steuerung der BusinessRules-Komponente
- Synchronisation mit der BusinessRules-Komponente
- Modifizieren des „Expertenwissens" der BusinessRules-Komponente. Das Expertenwissen ist das Regelwerk, das auf das Faktenwissen durch die BusinessRules-Komponente angewandt wird.

Business Objects/EJB-Ebene: Die Business Objects dienen der BusinessRules-Komponente als Faktenwissen. Erzeugte, gelöschte oder geänderte Geschäftsobjekte werden auf dieser Ebene der BusinessRules-Komponente durch einen Benachrichtigungsmechanismus bekannt gemacht.

Das so erweiterte Schichtenmodell wird in Abbildung 3.6 dargestellt[6]. Die Erweiterungen, die im Rahmen der Diplomarbeit entstanden sind, sind grau unterlegt dargestellt.

3.3.1 Inter-Server Kommunikation

Zur Kommunikation von eTS und BusinessRules-Server bzw. eCS und BusinessRules-Server wird der Java Remote Method Invocation-Mechanismus (RMI) eingesetzt. Die Inter-Server Kommunikation wird über zwei Objekte bzw. Schnittstellen abgewickelt. Diese Objekte dienen als Endpunkte der Inter-Server Kommunikation.

Innerhalb des jeweiligen enfinity Servers (eTS bzw. eCS) wird ein **Dispatcher**-Objekt installiert. Alle Zugriffe, die vom BusinessRules-Server auf den enfinity Server stattfinden, geschehen mittels eines entfernten Methodenaufrufes auf dieses Dispatcher-Objekt, es bildet das eine Ende des Kommunikationskanales BusinessRules-Server↔enfinity Server. Das Dispatcher-Objekt ist als Singleton-Objekt [GHJV96, S. 157] ausgelegt, d. h. pro enfinity Server existiert **genau ein** Objekt der Dispatcher-Klasse.

[6] das Schichtenmodell der enfinity-Plattform wird in Abbildung 2.6 dargestellt.

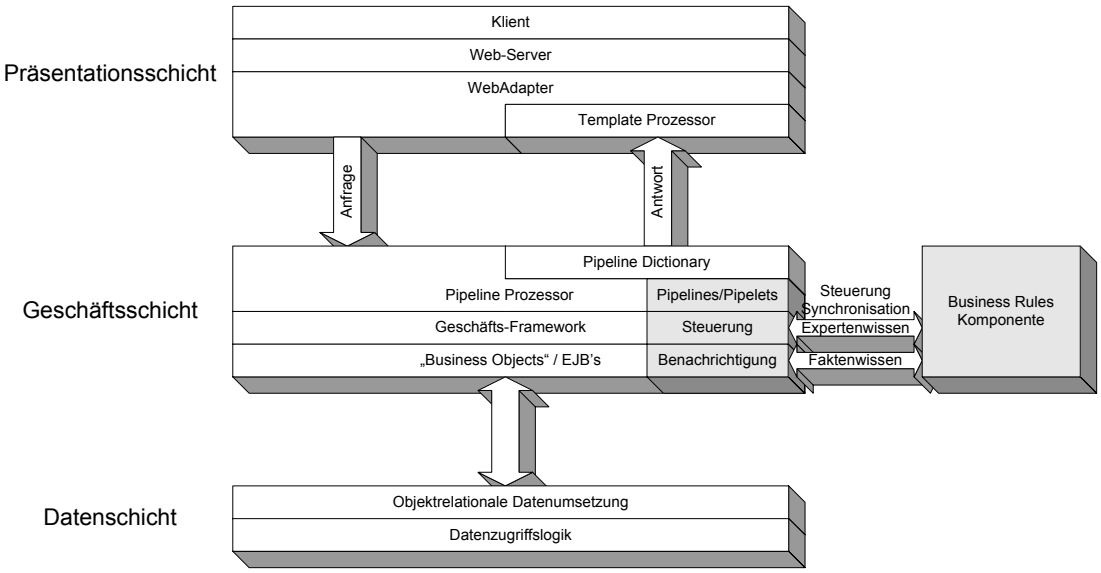

Abbildung 3.6: Die Einbettung der BusinessRules-Komponente in das Schichtenmodell

Auf der Seite des BusinessRules-Server erfüllt diesen Zweck das **BridgeManager**-Objekt. Wie beim Dispatcher-Objekt geschehen sämtliche Zugriffe und Benachrichtigungen, die von einem der enfinity Server ausgehen und den BusinessRules-Server als Ziel haben, durch einen entfernten Methodenaufruf auf das BridgeManager-Objekt. Das BridgeManager-Objekt dient als anderes Ende des Kommunikationskanals BusinessRules-Server↔enfinity Server. Das BridgeManager-Objekt ist ebenfalls ein Singleton-Objekt, es existiert in der durch den BusinessRules-Server erweiterten enfinity-Plattform **genau ein Mal**.

Beim Start des enfinity-Systems wird das BridgeManager-Objekt bei der RMI Registry [Sun99d] registriert. Mittels der RMI Registry können entfernte Objekte gefunden werden. Den Suchmechanismus der RMI Registry benutzt das Dispatcher-Objekt, um beim Start das BridgeManager-Objekt zu finden. Wurde das BridgeManager-Objekt gefunden, so registriert sich das Dispatcher-Objekt beim BridgeManager-Objekt mittels der **registerDispatcher()**-Methode.

3.3.2 Das Proxy/Adapter-Konzept

Wie in 3.1.1 beschrieben sind die Business Objects innerhalb des enfinity-Systems durch Enterprise JavaBeans[7] realisiert. Auf die Attribute, die den Zustand des Objektes beschreiben, kann durch die entsprechende get-Methode zugegriffen und mittels der set-Methode gesetzt werden.

[7]siehe Abschnitt 2.2.1.2

3.3.2.1 Struktur

Um die bestehenden Business Objects an die Forderungen, die ein Expertensystem an die Objekte, die es überwacht, anzupassen, wird das **Adapter-Muster** [GHJV96, S. 171] angewandt. Ein Adapter ist eine Klasse, die eine bestehende (Original-)Klasse an eine erwartete Schnittstelle anpasst, ohne dass die Originalklasse geändert werden muss.

Business Objects sind oftmals Objekte mit sehr vielen Attributen, die nicht alle durch das Expertensystem sichtbar sein müssen. Diese Attribute entstehen zum Großteil durch den Datenhaltungsmechanismus. Dies sind zum einen direkte Datenhaltungsattribute wie zum Beispiel das „Optimistic Control Attribute“ und zum anderen Attribute die durch Einbettung von Objekten in ein Business Object erforderlich sind.

Als Beispiel soll hier das Produktobjekt dienen. Ein Produkt hat das Attribut Preis vom Typ Geld, die Klasse Geld besteht wiederum aus den Attributen Betrag und Währung. Wird der Preis in das Produkt eingebettet, so entstehen das Attribut Preis vom Typ Geld, sowie die für die Datenhaltung notwendigen Attribute PreisBetrag und PreisWährung mit den entsprechenden get- und set-Methoden. Nun kann auf den Betrag auf zwei verschiedene Arten zugegriffen werden, zum einen mit getPreisBetrag() und mit getPreis().getBetrag(). Um diese Redundanz zu vermeiden, ist es sinnvoll, die Attribute PreisBetrag und PreisWährung zu verbergen.

Das **Proxy-Muster** [GHJV96, S. 254] hat den Zweck, den Zugriff auf ein Objekt zu kontrollieren. Ein Proxy ist eine Klasse, die den Zugriff auf die Originalklasse in irgendeiner Form kontrolliert. Hier soll das Proxy-Muster dazu eingesetzt werden, um Attribute, die durch die persistente Datenhaltung entstanden sind, zu verdecken.

Die Kombination des Adapter- und Proxymusters resultiert in einer Klasse, die als Vermittler zwischen der Original Business Object Klasse und der Regelmaschine dient und folgende Vorteile bringt:

- Anforderungen an die Schnittstelle, die durch die Regelmaschine an die überwachten Objekte gestellt werden, können erfüllt werden.

- Das Proxy Objekt ist ein lokales Objekt innerhalb des BusinessRules-Servers – Keine Referenzen auf Objekte, die sich auf entfernten Computern befinden.

- Unnötige Attribute der Originalklassen können verdeckt werden.

Abbildung 3.7 zeigt die Zusammenhänge zwischen dem Original Business Object und dem dazugehörenden Proxyobjekt.

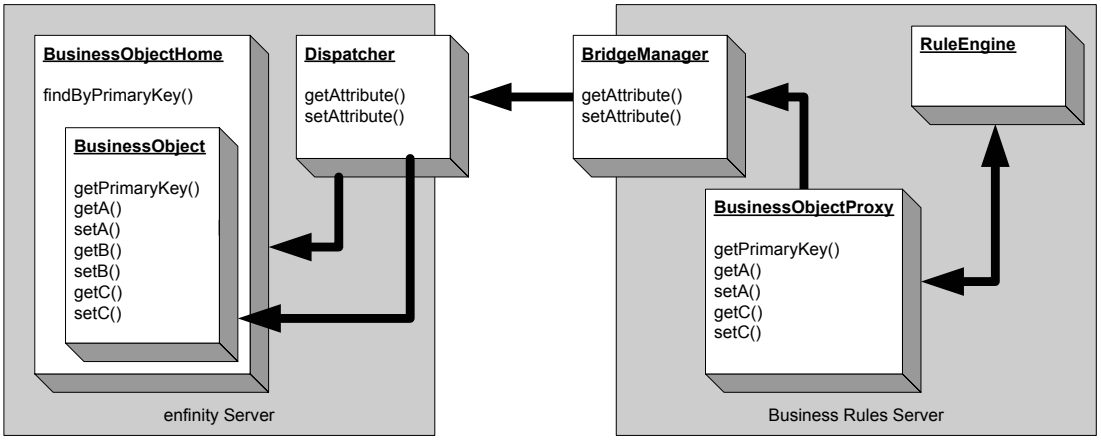

Abbildung 3.7: Übersicht über die Proxy/Adapter-Architektur

Die Struktur der Business Object Klassen bleibt erhalten. Wird eine Business Object Klasse von einer Superklasse abgeleitet, so wird die Proxyklasse ebenfalls von der Proxyklasse der Superklasse abgeleitet. Ist eine Business Object Klasse abstrakt, das heißt, es gibt keine Instanz eines Objektes von dieser Klasse, so wird die Proxyklasse dieser Business Object Klasse ebenfalls als abstrakte Klasse implementiert.

Innerhalb des Proxyobjekts werden nur die Daten gespeichert, die notwendig sind, um das Originalobjekt wieder zu finden. Das sind:

- Kennung des Dispatchers bzw. des Servers auf dem dieser installiert ist und sich das Originalobjekt befindet
- Kennung der Originalklasse (Klassenattribut)
- Primärer Schlüssel der EntityBean

Der primäre Schlüssel des Business Objects wird bei dessen Erzeugung festgelegt und ist während der Lebenszeit des Business Objects konstant. Aus diesem Grund kann eine Kopie des primären Schlüssels erzeugt werden, ohne dass weitere Vorkehrungen, die Änderungen des Originalschlüssels überwachen, getroffen werden müssten.

Methoden, die nicht zu den get- bzw. set-Methoden gehören, werden im Proxyobjekt nicht berücksichtigt. Das Proxyobjekt dient ausschließlich zum Feststellen von Attributwerten und zum Zuweisen von diesen.

3.3.2.2 Methodenaufrufe

Die get- und set-Methoden – sofern sie nicht das Attribut primaryKey betreffen – werden an das Originalobjekt weitergeleitet. Wie in Abschnitt 3.3.1 beschrieben, wird die gesamte Kommunikation über das BridgeManager- und Dispatcher-Objekt abgewickelt. Sowohl im BridgeManager als auch Dispatcher-Objekt existieren für diesen Zweck eine **getAttribute**- und eine **setAttribute**-Methode.

Abbildung 3.8 zeigt das Sequenzdiagramm einer get-Methode (Sequenznummer 1 – 5) und einer set-Methode (Sequenznummer 6 – 10).

Abbildung 3.8: Sequenzdiagramm der get- und set-Methoden

Die Abläufe eines get- und eines set-Methodenaufrufs sind, bis auf die Tatsache, dass bei einem set-Modenaufruf noch der neue Wert des Attributes als zusätzlicher Parameter übergeben wird, äquivalent. Im Folgenden soll nur noch der Ablauf eines get-Methodenaufrufs betrachtet werden.

Jeder get-Methodenaufruf hat einen Aufruf der Methode getAttribute des BridgeManagers zur Folge. Dieser sucht anhand des übergebenen dispatcherID-Parameters den Dispatcher, der auf dem enfinity-Server installiert wurde, auf dem sich

das Original Business Object befindet. Mittels eines Remote-Methodenaufrufs wird die getAttribute-Methode des Dispatchers aufgerufen. Anhand der businessClassID ermittelt der Dispatcher das Home-Interface der Enterprise JavaBean, die das Business Object realisiert. Dies ermöglicht der Registrymechanismus von enfinity, der es gestattet, Home-Interfaces von EJBs durch die Angabe von deren Namen zu finden. Mittels des primaryKey-Parameters kann nun das Original Business Object ermittelt werden. Der übergebene Parameter attributeName bestimmt die get-Methode, die innerhalb des Originalobjektes aufgerufen wird. Dieser Methodenaufruf wird mit Hilfe des Reflectionmechanismus' [Sun99c], mit dem Java-Objekte und -Methoden untersucht werden können, ausgeführt. Der durch diesen Methodenaufruf ermittelte Wert ist der Rückgabewert der aufgerufenen getAttribute-Methoden des Dispatchers und des BridgeManagers sowie des get-Methodenaufrufs des Proxyobjektes.

Folgende Tabelle gibt nochmals eine Übersicht über die benutzten Parameter und deren Zweck:

Parameter	**Zweck**	**benutzt von**
dispatcherID	Auswahl des Dispatchers, der auf dem enfinity-Server installiert ist, auf dem sich das Original Business Object befindet	BridgeManager
businessObject-ClassID	Auswahl der Original Business Object Klasse	Dispatcher
primaryKey	Auswahl des Original Business Objects	BusinessObject-Home
attributeName	Auswahl der entsprechenden get- bzw. set-Methode des Original Business Objects	Dispatcher, BusinessObject

Tabelle 3.1: Übersicht über die benutzten Parameter und deren Zweck

3.3.2.3 Generierung der Proxyklassen

Die Proxyklassen für die Business Object Klassen können generiert werden. Der Proxyklassengenerator untersucht hierfür die Business Object Klassen mit dem Reflection-Mechanismus von Java. Die Attribute der Originalklassen können durch die Existenz der get- bzw. set-Methoden bestimmt werden. Dem Benutzer wird die Liste der Attribute angezeigt, der dann die Attribute auswählen kann, die in der Proxyklasse implementiert werden sollen.

Sämtliche Proxyklassen erben direkt oder indirekt von einer abstrakten Proxyklasse, die die Logik für die Speicherung der dispatcherID, der businessObjectClassID sowie des primaryKeys enthält.

Alle weiteren Methoden der Proxyobjekte werden an das Originalobjekt weitergeleitet. Die einfache Struktur der Weiterleitung erlaubt es, einen effizienten Code für diese get- und set-Methoden zu generieren.

3.3.3 Benachrichtigungsmechanismus

In Abschnitt 3.3.2 wurde dargestellt, wie der Zugriff von dem entstandenen BusinessRules-Server auf die Business Object Menge konzipiert wurde. Dieser Abschnitt zeigt die Konzeption des umgekehrten Falls, der Kommunikation der Original Business Objects mit dem BusinessRules-Server, auf.

Das Proxyobjekt eines Geschäftsobjektes dient als Adapter für die Benachrichtigung des Expertensystems bzw. der Regelmaschine. Dieser Teil der Benachrichtigung ist abhängig von dem Mechanismus, der von der Regelmaschine gefordert wird. An dieser Stelle wird dieser Teil ausgelassen[8] und nur der Teil des Benachrichtigungsmechanismus' beschrieben, der das Proxyobjekt über Änderungen des Originalobjekts informiert.

Ein Proxyobjekt muss von dem Originalobjekt informiert werden, wenn das Originalobjekt

- **erzeugt**
- **geändert** (Attributen neue Werte zugewiesen werden)
- **gelöscht**

wird.

Wird ein Business Object erzeugt, so muss innerhalb der BusinessRules-Komponente ein Proxyobjekt für dieses Business Objekt erzeugt und die Regelmaschine über die neue Existenz des Proxyobjektes benachrichtigt werden.

Durch eine Änderung des Originalobjektes muss eine Benachrichtigung der Regelmaschine über diese Änderung hervorgerufen werden. Dies geschieht, indem die Regelmaschine über eine Änderung des entsprechenden Proxyobjektes benachrichtigt wird.

[8] eine Beschreibung der Anbindung der Proxyobjekte an die Regelmaschine **Jess** befindet sich in Abschnitt 4.1.1

Wird ein Originalobjekt gelöscht, so muss die Regelmaschine über die Löschung des (Proxy-)Objektes informiert und das Proxyobjekt zerstört werden.

Sämtliche Business Objects innerhalb des enfinity-Systems sind von der abstrakten Enterprise JavaBean *com.intershop.beehive.core.common.FWPersistentObject* abgeleitet. Eine Änderung dieser abstrakten Superklasse wirkt sich also auf alle von dieser Klasse direkt oder indirekt abgeleiteten Klassen aus. Dies hat den Vorteil, dass aus den Orginal-enfinity-Klassen ausschließlich die **FWPersistentObject**-Klasse geändert werden müssen, um den Benachrichtigungsmechanismus für alle im System enthaltenen Geschäftsobjekte zu implementieren.

3.3.3.1 Benachrichtigungszeitpunkt

In Abschnitt 3.1.2 wurde festgestellt, dass die Regelmaschine ausschließlich auf konsistente Objektmengen operiert. Aus dieser Forderung folgt, dass eine Benachrichtigung ausschließlich dann erfolgen kann, wenn eine Transaktion sich in der Festschreibphase[9][10] befindet.

3.3.3.2 Benachrichtigungsmechanismus im Enterprise JavaBean-Kontext

Die Implementierung der Logik einer Enterprise JavaBean erfolgt in der <**EJBNAME**>**Bean**-Klasse, sie ist der zentrale Teil einer EJB. Jede <**EJBNAME**>**Bean**-Klasse, die Bestandteil einer EJB zur persistenten Speicherung von Daten (EntityBean) ist, muss das Interface *javax.ejb.EntityBean* zwingend implementieren. Das Interface schreibt vor, dass unter anderem die Methoden **ejbStore** (Speichern des aktuellen Zustandes der EJB in der Datenbank) und **ejbRemove** (Löschen der EJB aus der Datenbank) implementiert werden müssen [Sun99a, Sun00b].

Die Enterprise JavaBeans Spezifikation [Sun99a, S. 109] schreibt vor, dass der EJB Container im Falle einer Änderung oder Erzeugung einer Enterprise JavaBean die Methode **ejbStore** der Bean-Klasse aufruft. Dieser Methodenaufruf wird auch während der Festschreibphase[11] der Transaktion aufgerufen, um die Änderung bzw. die Erzeugung des Objektes persistent zu machen.

Die Enterprise JavaBeans Architektur bietet zu diesem Zeitpunkt keine Möglichkeit, festzustellen, welche Attribute sich geändert haben oder ob eine Enterprise JavaBean neu erzeugt oder nur geändert wurde. Aus diesem Grund kann die Erzeugt-Benachrichtigung nur innerhalb der Start-Phase des Systems benutzt

[9] der Transaktionszustand wechselt vom Zustand **abgeschlossen** in den Zustand **persistent**
[10] siehe Abschnitt 2.2.1.3
[11] die Transaktion befindet sich im Zustand **STATUS_COMMITTING**

werden, wenn alle vorhandenen Business Objects der BusinessRules-Komponente bekanntgemacht werden. Die Entscheidung, ob ein Objekt neu erzeugt oder nur geändert wurde, muss innerhalb des BusinessRules-Servers gefällt werden. Ist ein Proxyobjekt des entsprechenden Typs mit diesem primaryKey innerhalb der BusinessRules-Komponente vorhanden, so wurde das Originalobjekt geändert. Ist es nicht vorhanden, so wurde es neu erzeugt und ein Proxyobjekt mit dem entsprechenden primaryKey muss erzeugt und der Regelmaschine bekannt gemacht werden.

Wird eine EJB gelöscht, so ruft der EJB Container die Methode **ejbRemove** auf. Ein ejbRemove-Methodenaufruf hat wiederum eine Benachrichtigung des Proxyobjektes über die Löschung des Originalobjektes zur Folge.

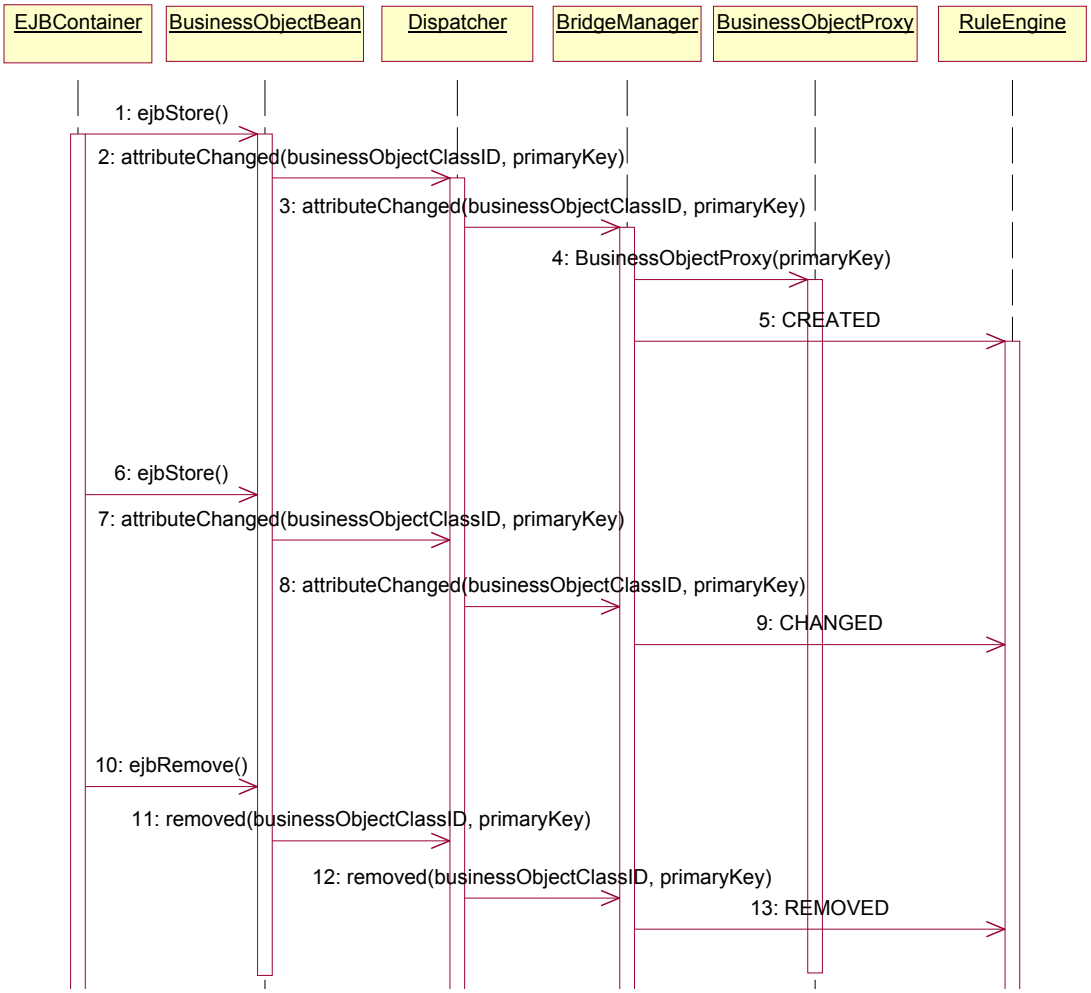

Abbildung 3.9: Sequenzdiagramm des Benachrichtigungsmechanismus'

Abbildung 3.9 zeigt das Sequenzdiagramm des Benachrichtigungsmechanismus'.

Die Sequenznummer 1 – 5 zeigt den Ablauf, wenn ein Business Object erzeugt, Sequenznummer 6 – 9, wenn ein Business Object geändert und Sequenznummer 10 – 13, wenn ein Business Object gelöscht wurde. Die Methoden des Dispatchers werden ausschließlich dann aufgerufen, wenn sich die Transaktion beim Aufruf der Bean-Methoden im Zustand STATUS_COMMITTING befindet.

3.3.4 Asynchronisierung der Methodenaufrufe

Synchrone Methodenaufrufe über Systemgrenzen hinweg erzeugen das Risiko von Verklemmungen (*engl.: deadlocks*). Folgende Abschnitte charakterisieren das Problem und stellen eine Lösung für dieses Problem vor.

3.3.4.1 Problemstellung

Java bietet durch das **synchronized**-Schlüsselwort die Möglichkeit, Methoden sowie Anweisungsblöcke eines Objektes vor der gleichzeitigen Ausführung dieser Methoden und Anweisungsblöcke zu schützen [GJSB00]. Dabei wird eine Sperre (*engl.: lock*) innerhalb eines Javaobjektes errichtet, so dass jeweils nur ein Thread die so markierten Methoden und Blöcke ausführen kann.

Wird in einem synchronisierten Block eines Objektes eine Methode eines anderen Objektes aufgerufen, die wiederum eine synchronisierte Methode des Ursprungsobjektes aufruft (callback), so wird der Block ausgeführt, da der ausführende Thread die Sperre auf das Ursprungsobjekt besitzt.

Anders verhält sich das Szenario, wenn sich die beiden Objekte nicht innerhalb der gleichen Java Virtual Machine befinden. Durch den entfernten Methodenaufruf kann nicht mehr festgestellt werden, dass der Thread von der ursprünglichen VM auf die andere VM wechselt um schließlich wieder zur Ursprungs-Virtual Machine zurückzukehren. In diesem Fall führt der zweite synchronisierte Methodenaufruf zu einer Verklemmung.

Abbildung 3.10 zeigt das oben beschriebene Szenario. Die Sperre wird durch das Schloss-Symbol dargestellt. Der in der Abbildung gezeigte *methodenAufruf2* führt zu der Verklemmung.

3.3.4.2 Problemstellung innerhalb der vorgestellten Systemarchitektur

Innerhalb der hier vorgestellten Systemarchitektur tritt das Problem dann auf, wenn die Regelmaschine während der Aktionsausführung eine Sperre errichtet

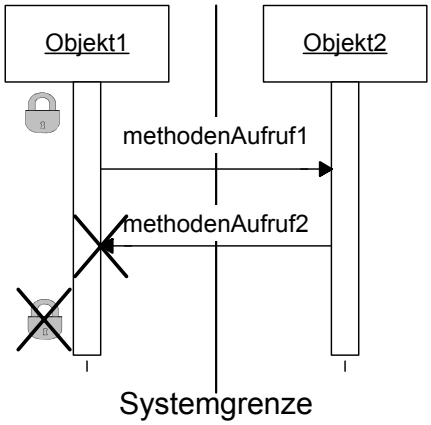

Abbildung 3.10: Verklemmungssituation durch entfernten Methodenaufruf

und während der Aktionsausführung ein Business Object erzeugt, geändert oder gelöscht wird.

Abbildung 3.11 verdeutlicht die Verklemmungssituation innerhalb der Systemarchitektur in dem Fall, in dem ein Business Object geändert wird.

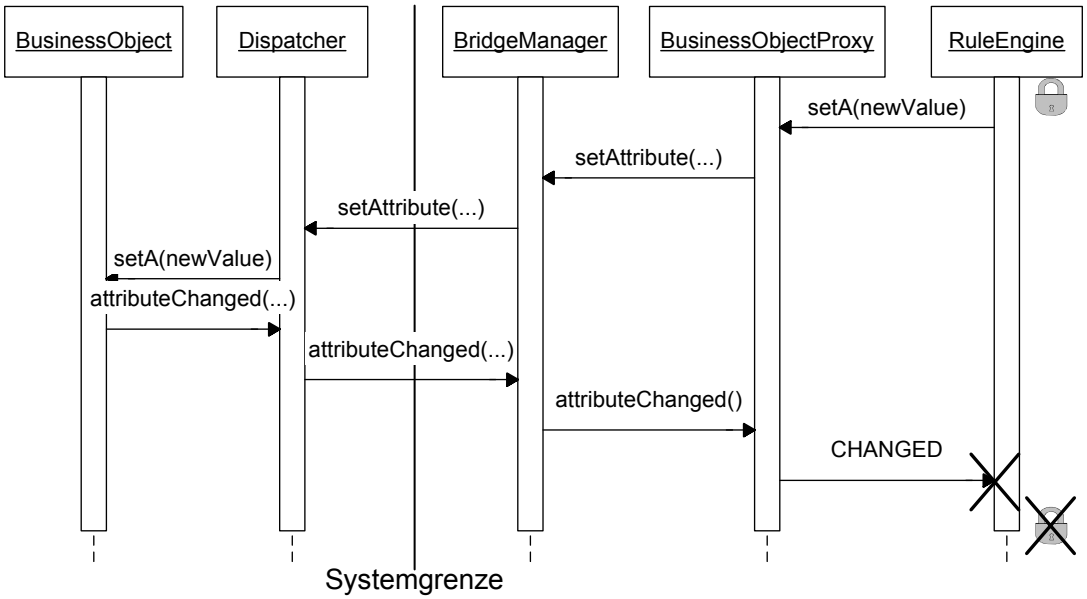

Abbildung 3.11: Verklemmungssituation durch entfernten Methodenaufruf in der vorgestellten Systemarchitektur

3.3.4.3 Problemlösung

Um das Verklemmungsproblem zu lösen, muss dafür gesorgt werden, dass der aufrufende Thread zurückkehrt und die Sperre frei gibt, bevor die zweite synchronisierte Methode aufgerufen wird.

Um dies zu erreichen, wird das Befehls-Muster (*engl.: command pattern*) [GHJV96, S. 273] angewandt. Mittels des Befehls-Musters kann ein Methodenaufruf in einem Objekt gekapselt werden. Dieses Objekt kann zwischengespeichert und in eine Warteschlange [OW93] eingefügt werden. Ein **Prozessor**-Thread nimmt die Befehlsobjekte aus der Warteschlange und führt die so gekapselten Methodenaufrufe aus. Der erste Thread erzeugt nur das entsprechende Objekt, legt es in die Warteschlange, kehrt zurück und kann die Sperre abbauen. Nun kann der zweite Thread ohne Verklemmungsgefahr die in dem Objekt gekapselte Methode aufrufen.

Abbildung 3.12 zeigt das Sequenzdiagramm der Methodenaufrufe mit der Warteschlange (**CommandQueue**) und dem Prozessor (**Processor**) im Falle einer Attributsänderungsaktion. Die Methodenaufrufe des Prozessor-Threads sind rot dargestellt.

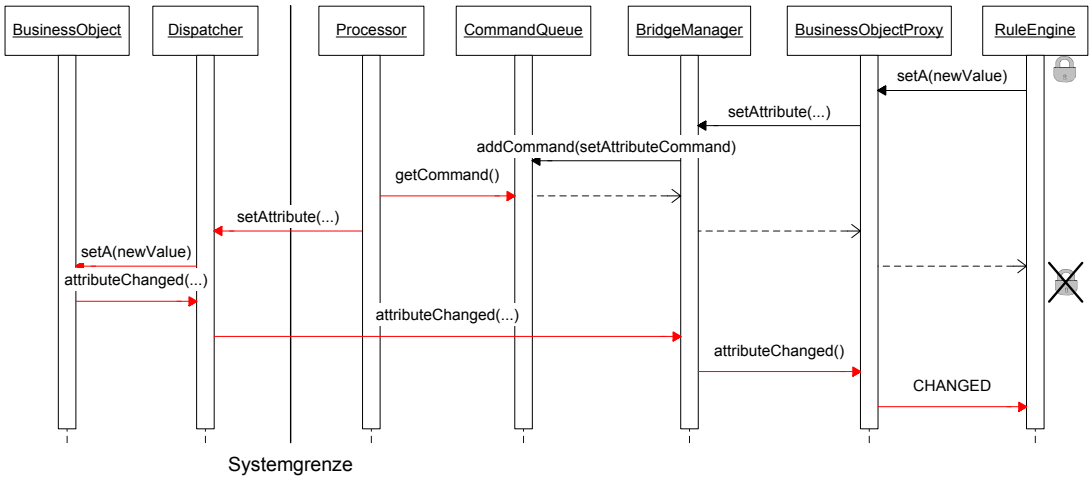

Abbildung 3.12: Sequenzdiagramm der Methodenaufrufe mit Warteschlange

3.3.4.4 Resultierende Forderungen an Aktionen

Um zu verhindern, dass der zurückkehrende Thread nicht mit der Regelauswertung fortfährt, muss am Ende einer Aktion die Regelauswertung gestoppt und ein Befehlsobjekt in die Warteschlange eingefügt werden, das die Regelauswertung

wieder startet. Dies ist notwendig, um eine Regelauswertung, die auf einer nicht aktuellen Faktenbasis basiert, zu verhindern.

Im Folgenden wird angenommen, dass sich in der Konfliktmenge zwei Regeln befinden. Die Konfliktlösungsstrategie wählt Regel 1 und führt den Aktionsteil der Regel 1 aus. Der Aktionsteil der Regel 1 ist so geschaffen, dass die Vorbedingung der Regel 2 ungültig wird. Das Anhalten der Regelauswertung verhindert, dass **unzulässigerweise** Regel 2 ausgeführt wird, weil sich noch die Aktionsbefehle der Regel 1 in der Warteschlange befinden, die die Vorbedingung der Regel 2 ungültig machen.

3.3.5 Persistente Speicherung von Fakten

Für die sinnvolle Anwendung von Regeln ist es notwendig, dass Fakten gespeichert werden können. Aus diesem Grund wird die zusätzliche **Fact**-Enterprise JavaBean eingeführt. Eine Fact-EJB ist im Normalfall an ein Geschäftsobjekt gebunden und hat die in Tabelle 3.2 aufgeführten Attribute.

Attribute	Zweck
objectID	Referenz auf das Geschäftsobjekt. Dieses Attribut enthält die UUID (eindeutige ID) des Geschäftsobjektes, an das das Fact-Object gebunden ist.
key	Zeichenfolgewert, dieses Attribut dient als Schlüssel der (Schlüssel → (Zahlenwert, Zeichenfolgewert))-Zuordnung.
longValue	Zahlenwert, der der Geschäftsobjekt-Schlüssel-Kombination hinterlegt werden kann.
stringValue	Zeichenfolgewert, der der Geschäftsobjekt-Schlüssel-Kombination hinterlegt werden kann.

Tabelle 3.2: Attribute der Fact-EJB

Die Fact-EJB ermöglicht es, bestehende Geschäftsobjekte mit beliebigen Attributen zu erweitern, mit der Einschränkung, dass sie vom Typ long, String bzw. boolean sein müssen. Ein boolesches Attribut wird durch die Existenz bzw. der Nichtexistenz des entsprechenden Schlüssels realisiert.

Sollen globale Fakten, das heißt, Fakten, die nicht an ein Geschäftsobjekt gebunden sind, gespeichert werden, so wird das objectID-Attribut auf eine Zeichenfolgekonstante gesetzt, wobei zu beachten ist, dass die Konstante nicht im Wertebereich der UUIDs liegt. Ein Beispiel hierfür wäre „|GLOBAL|“, da keine UUID im enfinity-System das Zeichen ‘|’ enthalten darf.

Zur Integration der Fact-Enterprise JavaBean in die Systemarchitektur wird ebenfalls das Proxy/Adapter-Konzept[12] und der Benachrichtigungsmechanismus[13] angewandt.

So kann zum Beispiel einem Käufer ein Attribut Typ hinterlegt werden, das die Käufermenge in die Typen „Gold“, „Silber“ und „Bronze“ partitioniert. Eine Regel, die als Vorbedingung voraussetzt, dass der Käufer vom Typ „Gold“ ist, hat folgende Form:

```
WENN
    Käufer (UUID ?uuid)
    Fact (objectID ?uuid) (key „Typ“) (stringValue „Gold“)
DANN
    ...
```

Der Platzhalter ?uuid, der beim Käufer- und beim Fact-Muster zum Einsatz kommt, stellt sicher, dass das Fact-Objekt zu dem Käuferobjekt gehört.

Eine Regel, die die Existenz eine globalen Faktums voraussetzt, hat zum Beispiel folgende Form:

```
WENN
    Fact (objectID „|GLOBAL|“) (key „Schlüssel“)
         (stringValue „Schlüsselwert“)
DANN
    ...
```

3.3.6 Kommunikation mit dem Pipeline Prozessor

Der Pipeline Prozessor führt Pipelines aus, die neben Kontrollstrukturen und Interaktionsknoten aus Pipelets bestehen. Mittels der Pipelets kann auf das Geschäftsframework und die Business Objects zugegriffen werden.

Der Zugriff auf ein FactEJB erfolgt mittels der **GetFact**- und **SetFact**-Pipelets. Diese Pipelets werden dazu benutzt, um während einer Anfrageabarbeitung auf Fakten zuzugreifen oder diese zu erzeugen bzw. zu ändern.

Um Fakten, die nicht an Geschäftsobjekte gebunden sind, sondern die zur Steuerung der dem Benutzer angebotenen Anfragen benötigt werden, wird ein **SessionFact**-EJB eingeführt.

[12] siehe Abschnitt 3.3.2
[13] siehe Abschnitt 3.3.3

Das SessionFact-EJB hat die gleiche Struktur wie das Fact-EJB. Das objectID-Attribut wird jedoch dazu benutzt, um ein SessionFact-Objekt einer Sitzung zuzuordnen. Der Zugriff auf die SessionFact-Objekte erfolgt mit den **GetSessionFact**- und **SetSessionFact**-Pipelets. Wird eine Sitzung beendet, so werden alle SessionFact-Objekte gelöscht, die an diese Sitzung gebunden sind.

Mittels der SessionFact-EJB können Anwendungen entwickelt werden, deren Logik komplett durch Regeln und Pipelines realisiert ist. Die Form der Regeln lässt sich folgendermaßen skizzieren:

WENN
SessionFact (objectID ?sessionID) (key <Schlüssel1>)
(stringValue <Zeichenfolgenwert>)
SessionFact (objectID ?sessionID) (key <Schlüssel2>)
(longValue <Zahlenwert>)
DANN
erzeuge, ändere, lösche SessionFact-Objekte
erzeuge, ändere, lösche Geschäftsobjekte

oder

WENN
SessionFact (objectID ?sessionID) (key <Schlüssel>)
(stringValue <Zeichenfolgenwert>)
Geschäftsobjekt ...
DANN
erzeuge, ändere, lösche SessionFact-Objekte
erzeuge, ändere, lösche Geschäftsobjekte

In diesem Fall besteht eine Pipeline aus folgenden Phasen:

1. **Setzen der SessionFacts:** Parameter der Anfrage, die als Fakten für das Regelwerk dienen, werden mittels des SetSessionFact-Pipelets in SessionFacts gespeichert.

2. **Regelauswertung:** Die Regeln werden auf die Fakten angewendet.

3. **Holen der SessionFacts:** Fakten, die durch Regeln erzeugt oder geändert wurden, werden mittels des GetSessionFact-Pipelets im PipelineDictionary gespeichert. Die im PipelineDictionary gespeicherten Fakten können dazu benutzt werden, um den weiteren Ablauf der Pipelineabarbeitung zu steuern oder dem Benutzer angezeigt werden.

3.3.6.1 Synchronisation

Eine besondere Form der Kommunikation mit dem Pipeline Prozessor stellt die Synchronisation dar.

Durch die Asynchronisierung der Methodenaufrufe[14] entsteht eine Parallelität zwischen der Geschäftslogik, die durch Pipelines realisiert wird, und der Logik, die durch das zu dem Regel- und Aktionenprozessor gehörende Regelwerk realisiert wird.

Der Aktions- und Regelprozessor befindet sich genau dann in einem synchronen Zustand, wenn alle Zustandsänderungen der Objekte, die dem Aktions- und Regelprozessor als Fakten dienen, verarbeitet wurden und keine Regel mehr auf diese Fakten angewandt werden kann.

Um dies zu erreichen, wird ein **Monitor**-Objekt in die Faktenbasis der Regelmaschine eingefügt. Ein Monitor-Objekt besitzt die **waitForNotification**-Methode, die den aufrufenden Thread solange anhält, bis die **notifyAllThreads**-Methode aufgerufen wird. Das Monitor-Objekt ist ein Remoteobject, so dass die jeweiligen Methoden auch von nicht-lokalen Java VMs aufgerufen werden können.

Initial wird eine Regel in das Regelwerk eingefügt, die bei der Existenz eines Monitor-Objektes dieses aus der Faktenbasis löscht und die notifyAllThreads-Methode aufruft. Um sicherzustellen, dass diese Regel nur dann ausgeführt wird, wenn keine andere Regel mehr ausgeführt werden kann, besitzt diese Regel die niedrigste Priorität innerhalb des Regelwerkes. Dieser Prioritätenlevel darf nicht für benutzerdefinierte Regeln verwendet werden.

Abbildung 3.13 zeigt das Sequenzdiagramm dieser Synchronisation. Die Methodenaufrufe des Threads, der auf die Synchronisation wartet, ist schwarz dargestellt. Die rotgezeichneten Methodenaufrufe sind die, die innerhalb des Aktionsteils der oben vorgestellten Regel ausgeführt werden. Zur Vereinfachung wird in dem Diagramm die Warteschlange nicht dargestellt. Das BridgeManager-Objekt fügt den Einfüge-Befehl ebenfalls in die Warteschlange ein, bevor der Befehl durch den Prozessor ausgeführt wird.

Um zu verhindern, dass die Regelmaschine die notifyAllThreads-Methode aufruft, bevor der auf die Synchronisation wartende Thread die waitForNotification-Methode aufgerufen hat, wird eine Sperre errichtet, die innerhalb der waitFor Notification-Methode wieder abgebaut wird. In Abbildung 3.13 wird dies durch das Schlosssymbol dargestellt.

[14] siehe Abschnitt 3.3.4

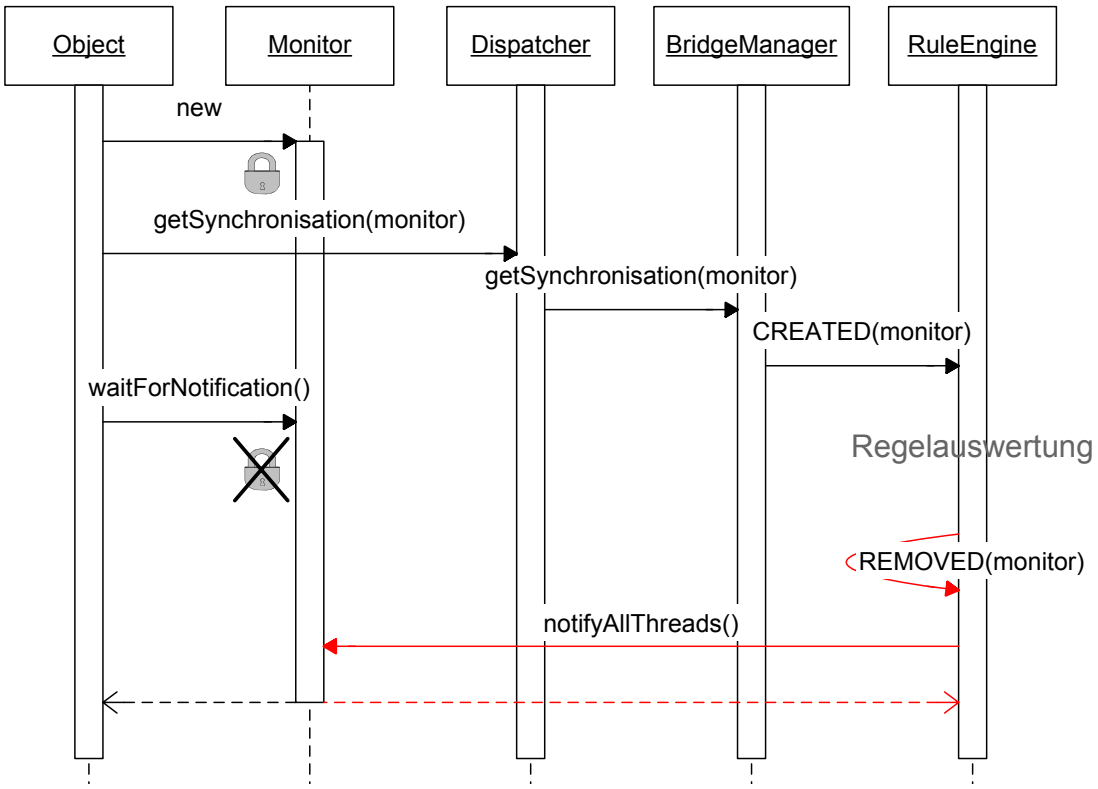

Abbildung 3.13: Sequenzdiagramm des Synchronisationsmechanismus'

3.3.7 Infrastruktur

Abhängig von den verschiedenen Phasen[15] ist es erforderlich, dass eine leistungsfähige Infrastruktur zur effizienten regelbasierten Manipulation von Geschäftsobjekten vorhanden ist.

In Phase I sind ausschließlich Infrastrukturelemente zur Aktionsabwicklung notwendig. In Phase II werden zusätzlich Elemente zur effizienten Formulierung der Vorbedingungen benötigt.

Folgende Funktionen sollen mit Hilfe dieser Infrastruktur ermöglicht werden:

- Phase I
  - Ändern eines Geschäftsobjektes
  - Löschen eines Geschäftsobjektes
  - Erzeugen eines Geschäftsobjektes
  - Transaktionshandhabung

[15] siehe Abschnitt 3.2

- Phase II
  - Sichtspezifische Tests
  - Plausibilitätskontrollen

In den folgenden Abschnitten wird auf die einzelnen Funktionen der Infrastruktur eingegangen.

3.3.7.1 Ändern eines Geschäftsobjektes

Ein Geschäftsobjekt wird geändert, indem ein Attribut mittels eines set-Methoden-Aufrufs neu gesetzt wird. Für die Regelmaschine ist nur das Proxyobjekt des zu ändernden Geschäftsobjektes sichtbar. Da auf das Originalobjekt nur mittels einer Transaktion zugegriffen werden kann, wird vor dem set-Methoden-Aufruf des Originalobjektes eine Transaktion begonnen und nach dem Aufruf der set-Methode wird diese Transaktion festgeschrieben. Der set-Methoden-Aufruf des Proxyobjektes enthält also eine implizite Transaktion.

Abbildung 3.14 zeigt das Sequenzdiagramm eines set-Methodenaufrufs mit der impliziten Transaktion (Sequenznummer 1 – 8).

3.3.7.2 Löschen eines Geschäftsobjektes

Zum Löschen eines Geschäftsobjektes wird die remove-Methode des Proxyobjektes aufgerufen. Dieser remove-Methoden-Aufruf wird wiederum an das Originalobjekt weitergeleitet. Wie beim set-Methodenaufruf enthält der remove-Methoden-Aufruf eine implizite Transaktion.

Abbildung 3.14 zeigt das Sequenzdiagramm eines remove-Methodenaufrufs mit der impliziten Transaktion (Sequenznummer 9 – 16).

3.3.7.3 Erzeugen eines Geschäftsobjektes

Ein neues Geschäftsobjekt wird erzeugt, indem auf das Home-Interface der Enterprise JavaBean zugegriffen und die entsprechende create-Methode mit den zugehörenden Parametern aufgerufen wird. In diesem Fall ist keine implizite Transaktion in den Methodenaufrufen enthalten. Soll mittels einer Transaktion ein Geschäftsobjekt erzeugt werden, so muss diese explizit mittels der integrierten Transaktionshandhabung[16] veranlasst werden.

[16] siehe Abschnitt 3.3.7.4

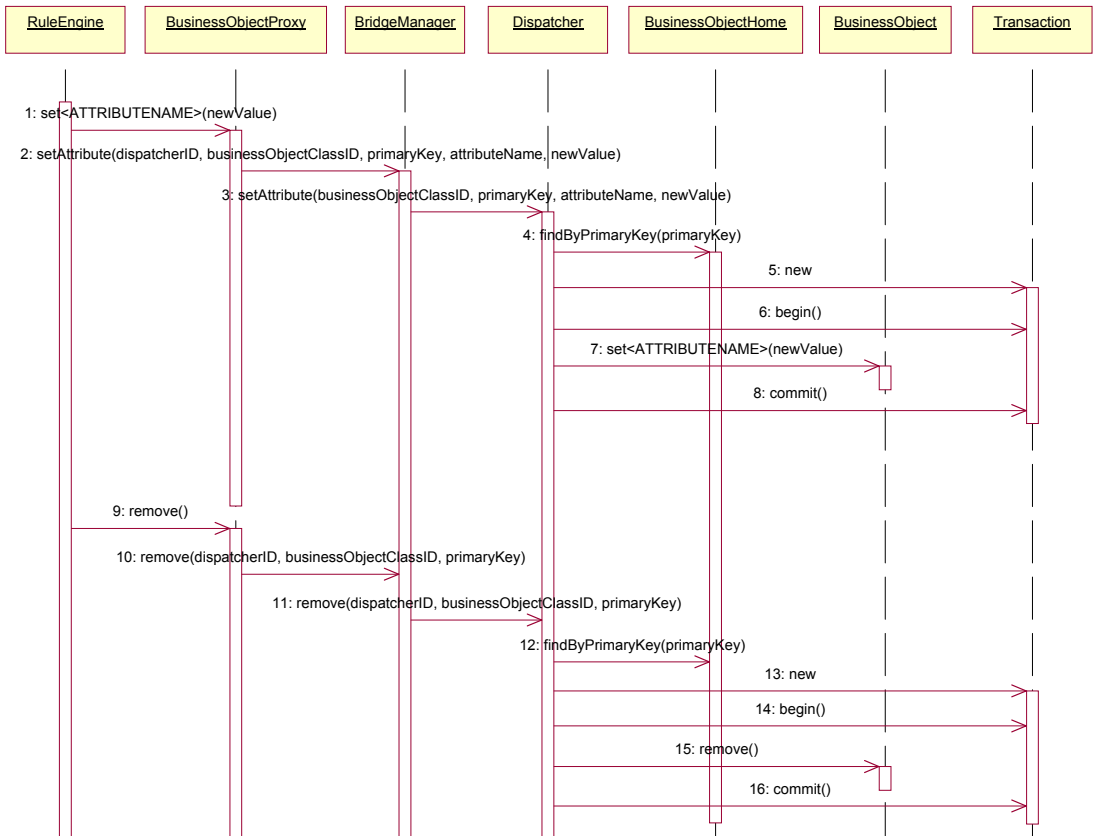

Abbildung 3.14: Implizite Transaktion der set- und der remove-Methoden

3.3.7.4 Transaktionshandhabung

Um innerhalb einer Aktion mittels Transaktionen[17] auf Geschäftsobjekte zuzugreifen, besitzt der BridgeManager die **getUserTransaction**-Methode. Diese ermöglicht es dem Benutzer, explizite Transaktionen innerhalb einer Aktion zu beginnen, festzuschreiben oder diese zu verwerfen.

3.3.7.5 Sichtspezifische Tests

In Abschnitt 3.2 wurden Sichten als anwendungsspezifische Wissensakquisitions- und Dialogkomponenten eingeführt.

Um dem Experten die Möglichkeit zu geben, sein Wissen in einfacher Weise durch entsprechende Regeln zu formulieren, ist es notwendig, dass Tests für den Vorbedingungsteil einer Regel, die über die Attributeigenschaften der Geschäftsobjekte hinausgehen, implementiert werden.

[17] siehe Abschnitt 2.2.1.3

Ein typisches Beispiel für einen sichtspezifischen Test ist der Monatsumsatz eines Käufers zum Beispiel für die umsatz- und käuferabhängige Preisbestimmung. Der Monatsumsatz ist in keinem Geschäftsobjekt als Attribut vorhanden sondern muss aus den erfolgten Bestellungen des Käufers errechnet werden. Eine weitere Anwendungsmöglichkeit der sichtspezifischen Tests ist der Zugriff auf Systeme, die sich außerhalb des E-Commerce-Systems befinden. So kann zum Beispiel auf ein Lagerhaltungssystem zugegriffen werden und Regeln erstellt werden, die die Lieferbarkeit voraussetzen.

3.3.7.6 Plausibilitätskontrollen

Die BusinessRules-Komponente erlaubt es, direkt Geschäftsobjekte zu erzeugen, zu manipulieren und zu löschen. Auf diesem direkten Zugriff beruht die Leistungsfähigkeit der BusinessRules-Komponente, birgt aber auch die Gefahr in sich, dass ein Fehler eines Experten verheerende Folgen für das Gesamtsystem hat.

Angenommen, in einer Datenpflege-Sicht wird eine Regel zur Löschung von Käufern, die sich ohne Nachnamen registriert haben, eingefügt, dann besteht die Vorbedingung aus dem Attributtest: *Vorname des Käufers ist gleich dem Leerstring.* Wird nun ausversehen nicht auf Gleichheit, sondern auf *Enthaltensein* geprüft, so werden sämtliche Käufer aus dem System eliminiert, da der Leerstring in allen Zeichenketten enthalten ist.

Um diese potentielle Fehlerquelle einzugrenzen, ist eine Integration von Plausibilitätskontrollen von Regeln in eine Sicht wünschenswert.

3.4 Benutzerfreundliche Regeldarstellung

Zur benutzerfreundlichen Darstellung und Eingabe der Regeln ist es erforderlich, dass diese in einer leicht verständlichen Form dem Benutzer präsentiert werden. Benutzerfreundlichkeit endet nicht bei der Darstellung von einzelnen Regeln. So ist es zum Beispiel unablässig, dass dem Benutzer die Möglichkeit geschaffen wird, das komplette Regelwerk (die Gesamtmenge der Regeln) komfortabel zu verwalten.

Im Folgenden wird auf die Präsentation einer Regel eingegangen. Dabei werden Darstellungen betrachtet, die diesen Anforderungen genügen, dieses sind:

- die Regeldarstellung in natürlicher Sprache
- die Regeldarstellung durch Symbole

3.4.1 Darstellung durch natürliche Sprache

Bei der Darstellung der Regeln durch natürliche Sprache, werden die Vorbedingungen, denen die Objekte genügen müssen, aus Satzteilen der natürlichen Sprache zusammengesetzt. Ein Beispiel[18] für diese Regeldarstellung ist

If
the shopping cart contains between 2 and 4 items
and the purchase value is greater than 100
and the customer category is Gold
Then
apply a 15% discount
and display the message 'We're giving you a nice discount!'

Der Aktionsteil einer Regel wird ebenfalls durch Zusammensetzen von Satzteilen dargestellt, wobei ein Satzteil eine Teilaktion repräsentiert. In dem Beispiel setzt sich die Aktion der Regel aus den zwei Teilaktionen *„gewähre Rabatt“* und *„zeige dem Käufer Nachricht“* zusammen.

3.4.2 Symbolsprache

Rein textuelle Darstellungen von Regeln haben den Nachteil, dass sie bei komplexen Regeln sehr schnell unübersichtlich werden oder, im Falle einer programmiersprachenähnlichen Notation, der Benutzer sie als kryptisch wahrnimmt.

Die hier vorgestellte Regelsprache hat den Zweck, dem Benutzer Regeln in einfacher, übersichtlicher und gut strukturierter Form zu präsentieren und ihm ebenfalls eine effiziente Art der Regeleingabe zu ermöglichen. Dabei wird darauf geachtet, dass eine normkonforme [Deu98, Teil 12 – Teil 17] Darstellung der Regeln auf dem Bildschirm sichergestellt werden kann.

3.4.2.1 Struktur

Eine Regel besteht aus dem Vorbedingungs- und dem Aktionsteil. Analog der Darstellung von Kalkülen wird der Darstellungsbereich einer Regel vertikal aufgeteilt, wobei sich der Vorbedingungsbereich oben und der Aktionsbereich sich unten befindet. Diese Darstellungsform ist jedem Benutzer vertraut, sie wird zum Beispiel auch bei der handschriftlichen Addition eingesetzt (oberer Bereich: Zahlen, unterer Bereich: Summe der Zahlen).

3.4.2.1.1 Vorbedingungsbereich Im Vorbedingungsbereich werden die einzelnen, konjunktiv verknüpften, Objektmuster horizontal angeordnet. Neben den Objektmustern werden im Vorbedingungsbereich auch Testelemente angezeigt.

[18] entnommen aus [ILO01a]

3.4.2.1.1.1 Objektmuster Ein Objektmuster besteht aus dem Objekttyp, dem Variablennamen des Objektes, das das Muster erfüllt und den Eigenschaften, denen die Attribute genügen müssen. Diese drei Elemente eines Objektmusters werden vertikal angeordnet.

Der Objekttyp wird in Form eines Symboles für den jeweiligen Objekttyp angezeigt. Dies erhöht die Übersichtlichkeit der Regeldarstellung, der Benutzer kann auf einen Blick erkennen, welche Objekttypen im Vorbedingungsbereich Verwendung finden. Vorraussetzung hierfür ist, dass für jeden Objekttyp ein intuitives Symbol definiert wird.

Die Attributseigenschaften werden in einer zweispaltigen Tabelle angeordnet. Die linke Spalte enthält den Namen des Attributs, die rechte eine Konstante, einen booleschen Ausdruck oder einen Variablennamen.

Der Variablennamen des Objektes bzw. des Attributes wird dazu verwendet, um Attributseigenschaften in Abhängigkeit von anderen Attributen zu definieren und um auf das Objekt bzw. die Attributwerte im Aktionsteil zugreifen zu können. Der Gültigkeitsbereich einer Variablen ist auf eine Regeldefinition beschränkt, innerhalb dieser Regel ist sie aber global, d. h. sie kann in sämtlichen Objektmustern, Testelementen und im Aktionsteil einer Regel Verwendung finden.

In Abbildung 3.15 wird die Variable buyerID im shoppinglist-Objekt sowie im Buyer-Objekt verwendet. Diese Konstruktion stellt sicher, dass das Attribut buyer im Shoppinglist-Objekt und das Attribut ID im Buyer-Objekt die gleichen Werte besitzen müssen.

Um die gegenseitige Abhängigkeit der Attribute bzw. Objekte zu visualisieren, werden die von dem aktuellen Objekt bzw. Attribut abhängigen Attribute, Objekte und Aktionen hervorgehoben dargestellt. In Abbildung 3.15 wird durch die Aktivierung des Attributs shoppinglist.buyer auch das Attribut buyer.ID hervorgehoben dargestellt, da beide Attribute die Variable buyerID referenzieren. Diese Hervorhebung ermöglicht es, alle Abhängigkeiten des aktiven Attributes / Objektes auf einen Blick zu erkennen.

Ein Objektmuster entspricht einer booleschen Funktion mit folgendem Aufbau:

$f(o) = f_t(o) \wedge f_{a_{n1}}(o.a_{n1}) \wedge f_{a_{n2}}(o.a_{n2}) \wedge \ldots \wedge f_{a_{nm}}(o.a_{nm})$

wobei

$$f_t(o) = \begin{cases} \textbf{wahr}, & \text{falls Objekt o vom Typ T oder einem von T abgeleiteten Typ ist} \\ \textbf{falsch}, & \text{sonst} \end{cases}$$

und

$f_{a_{nl}}(o.a_{nl})$ eine boolesche Funktion über das Attribut a_{nl} ist.

Hierbei kann zur Vereinfachung der Schreibweise der Ist-Gleich-Operator weggelassen werden. So kann z. B. anstatt

a_{nl} =<Konstante> a_{nl} <Konstante>

oder anstatt

a_{nl} =<Variable> a_{nl} <Variable>

geschrieben werden.

3.4.2.1.1.2 Testelemente Testelemente sind boolesche Funktionen, die nicht dem Aufbau der Objektmuster genügen. Sie bieten die Möglichkeit, allgemeine Tests durchzuführen. Das Hauptanwendungsgebiet von Testelementen sind sichtspezifische Tests[19]. Mittels Testelementen können Tests der Form:

- **istLieferbar**(produktID)

oder

- **monatsUmsatz**(buyerID) > 1000

durchgeführt werden.

Der Aufbau eines Testelementes in der Symbolsprache entspricht dem eines Objektmusters, wobei einem Testelement kein Variablennamen zugeordnet wird und anstatt der Attributseigenschaften wird die Testfunktion des Elementes angegeben. Ein Testelement wird durch das Fragezeichensymbol repräsentiert, das anstatt des Typsymbols angezeigt wird.

3.4.2.1.2 Aktionsbereich Innerhalb des Aktionsbereiches werden die einzelnen Aktionen vertikal angeordnet. Bei Aktionen, die Operationen auf Objekte ausführen, ist der Einzelaktionsbereich horizontal dreigeteilt. Im linken Bereich

[19] siehe Abschnitt 3.2

findet sich das Objekt bzw. die Objektvariable, auf dem die Aktion ausgeführt wird. Zur Erhöhung der Übersichtlichkeit wird in diesem Bereich auch das Symbol des entsprechenden Objekttyps angezeigt. Im mittleren Bereich wird die Operation, die auf dieses Objekt angewendet wird, textuell angezeigt. Die Parameter der Operation befinden sich im rechten Bereich. Abbildung 3.15 zeigt einen so aufgebauten Aktionsbereich.

3.4.2.2 Beispiel

Abbildung 3.15 zeigt die Regel des Beispieles, das in Abschnitt 3.4.1 vorgestellt wurde, in der eingeführten Symbolsprache.

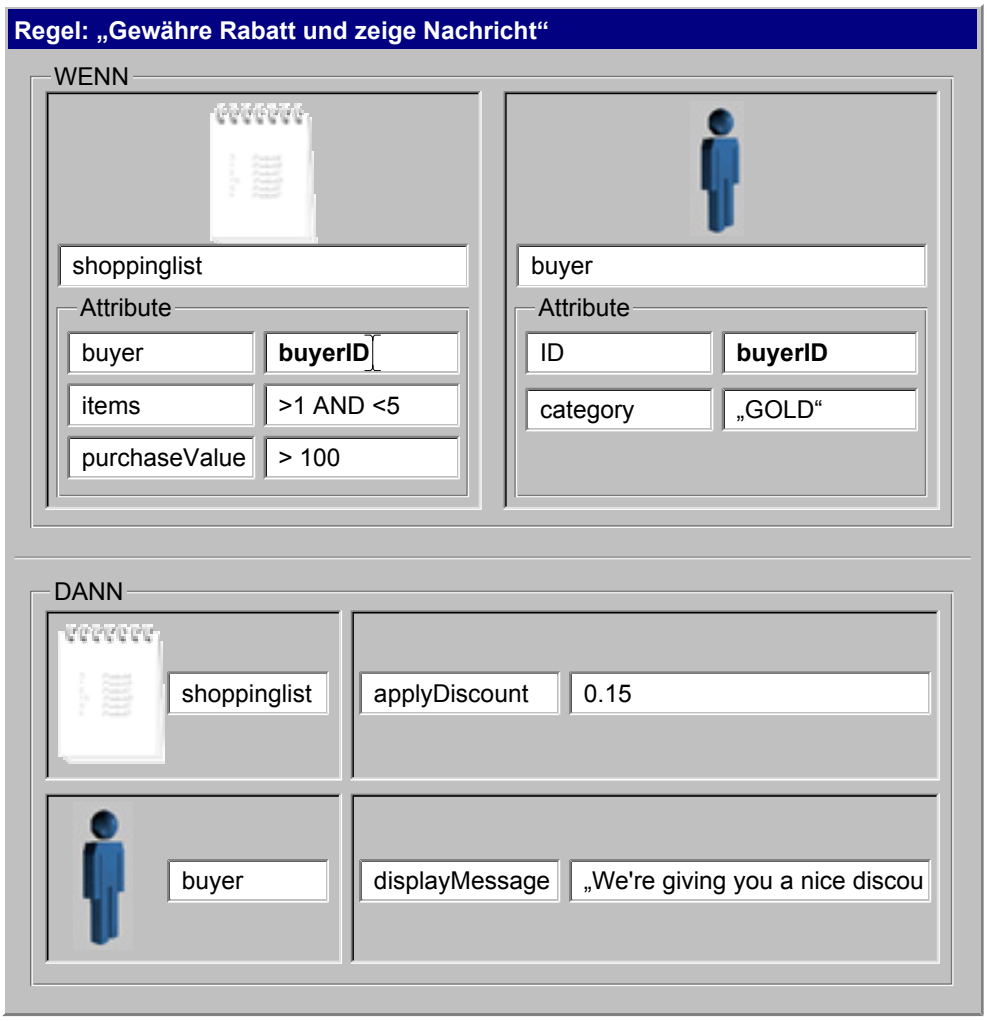

Abbildung 3.15: Darstellung des Beispiels durch die Regelsymbolsprache

Kapitel 4

Realisierung

Im Rahmen der Diplomarbeit wurde ein erster Prototyp der vorgestellten Konzeption implementiert. Der Prototyp umfasst die für die Phase I[1] vorgesehenen Funktionen. Zu Beginn der Diplomarbeit wurden verschiedene Regelmaschinen, die sich in eine Javaumgebung integrieren lassen, evaluiert[2]. Diese Evaluierung ergab, dass die Java Expert System Shell (Jess), die für diesen Zweck am geeigneteste Regelmaschine ist. Im Folgenden wird auf die realisierungsrelevanten Aspekte der in Abschnitt 3 vorgestellten Konzeption eingegangen.

4.1 Integration von Jess

Als Regelmaschine für die Realisierung wurde Jess[3] eingesetzt. In den folgenden Abschnitten werden die von der Regelmaschine abhängigen Mechanismen der Integration vorgestellt.

4.1.1 Benachrichtigungsmechanismus

Jess verlangt von den von ihr überwachten Objekten, dass sie als JavaBean implementiert sind. In der JavaBeans-Spezifikation [Sun97, S. 40] wird der Begriff des *Property* definiert. Ein *Property* ist ein Attribut eines Objektes von einem bestimmten Typ und die dazugehörenden get- und/oder set-Methoden zum Lesen und/oder Setzen dieses Attributes.

Die JavaBeans-Spezifikation [Sun97] sieht vor, dass Objekte, die das *PropertyChangeListener*-Interface [Sun99c] implementieren, mittels der *addPropertyChangeListener*-Methode bei der JavaBean registriert werden können. Die registrierten Listener müssen bei einer Änderung der Attribute eines Objektes mittels

[1] siehe Abschnitt 3.2
[2] siehe Abschnitt A
[3] siehe Abschnitt A.3

eines *PropertyChangeEvent*-Objektes benachrichtigt werden. Zur Vereinfachung des JavaBean-Mechanismus' beinhaltet die Java 2 Plattform die Hilfsklasse *PropertyChangeSupport*, die die Verwaltung und die Benachrichtigung der Listener vereinfacht.

Die Zusammenhänge, der für den Benachrichtigungsmechanismus relevanten Klassen *PropertyChangeListener* und *PropertyChangeEvent* sowie die weiteren, an der Benachrichtigung der Regelmaschine beteiligten Klassen, werden im UML-Klassendiagramm in Abbildung 4.1 dargestellt.

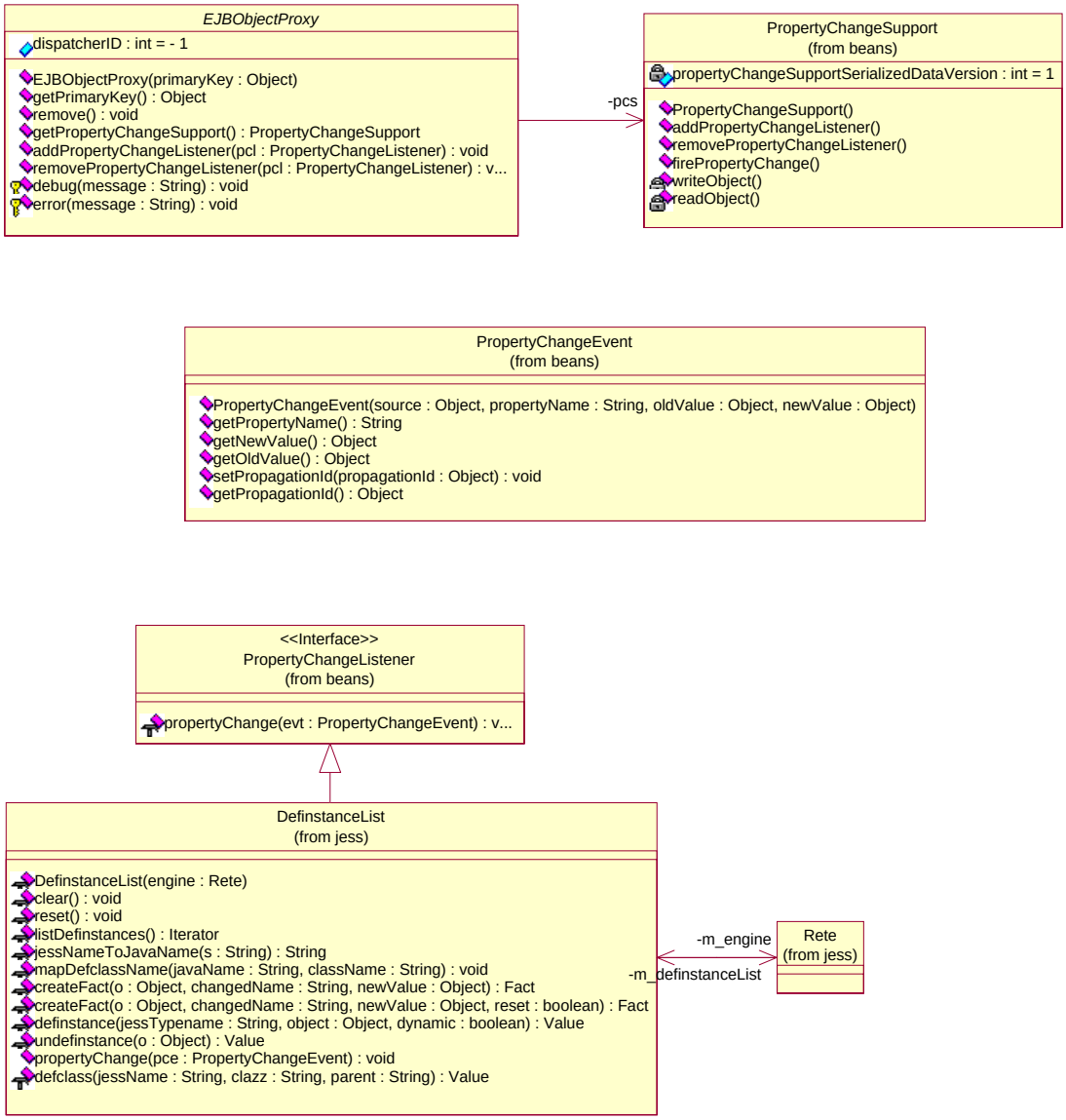

Abbildung 4.1: Klassendiagramm des Benachrichtigungsmechanismus'

Die Schnittstelle des Jess-Systems nach außen ist die *Rete*-Klasse [FH01], mittels ihr wird das komplette Jess-System gesteuert. Um ein JavaBean-Objekt zur Faktenbasis von Jess hinzuzufügen, wird die Methode *definstance* der Klasse *Rete* aufgerufen. Die Klasse *DefinstanceList* implementiert das Interface *PropertyChangeListener* und wird bei jeder JavaBean, die von Jess überwacht werden soll, automatisch als Listener registriert.

Um die Proxyobjekte[4] der Geschäftsobjekte an den JavaBean-Mechanismus anzupassen, referenzieren sie ein *PropertyChangeSupport*-Objekt und implementieren zusätzlich die Methoden *addPropertyChangeListener* und *removePropertyChangeListener*, die an das Support-Objekt weiterdelegiert werden. Diese Methoden werden in der Klasse *EJBObjectProxy* implementiert, diese dient als Superklasse aller Proxyklassen.

Die folgende Abbildung 4.2 vervollständigt die Abbildung 3.9 auf Seite 51. Zu beachten ist, dass die konzeptionellen Klassen *BusinessObjectProxy* und *RuleEngine* durch die konkreten, in der Realisierung verwendeten Klassen *EJBObjectProxy* und *Rete* ersetzt wurden.

Die Sequenznummer 2 – 6 zeigt den konkreten Ablauf der in Abbildung 4.2 skizzierten **CREATED**-Benachrichtigung der Regelmaschine in der hier vorgestellten Realisierung, der Vollständigkeit halber wurde nochmals die Erzeugung des Proxyobjektes (Sequenznummer 1 in Abbildung 4.2, Sequenznummer 4 in Abbildung 4.2) dargestellt. Die Sequenznummern 7 – 10 und 11 – 14 zeigen entsprechend die **CHANGED**- und **REMOVED**-Benachrichtigung (Sequenznummern: 9 und 13 in Abbildung 3.9).

4.1.2 Erweiterung des Jess-Befehlssatzes

Die Funktionalität der Jess-Sprache kann durch eigene Funktionen erweitert werden. Solche Funktionen können

- mit der deffunction-Funktion in der Jess-Sprache
- mittels einer Java-Klasse, die das jess.UserFunction-Interface implementiert

realisiert werden. Die folgenden Funktionen werden durch eine entprechende Java-Klasse in das Jess-System integriert:

[4] siehe Abschnitt 3.3.2

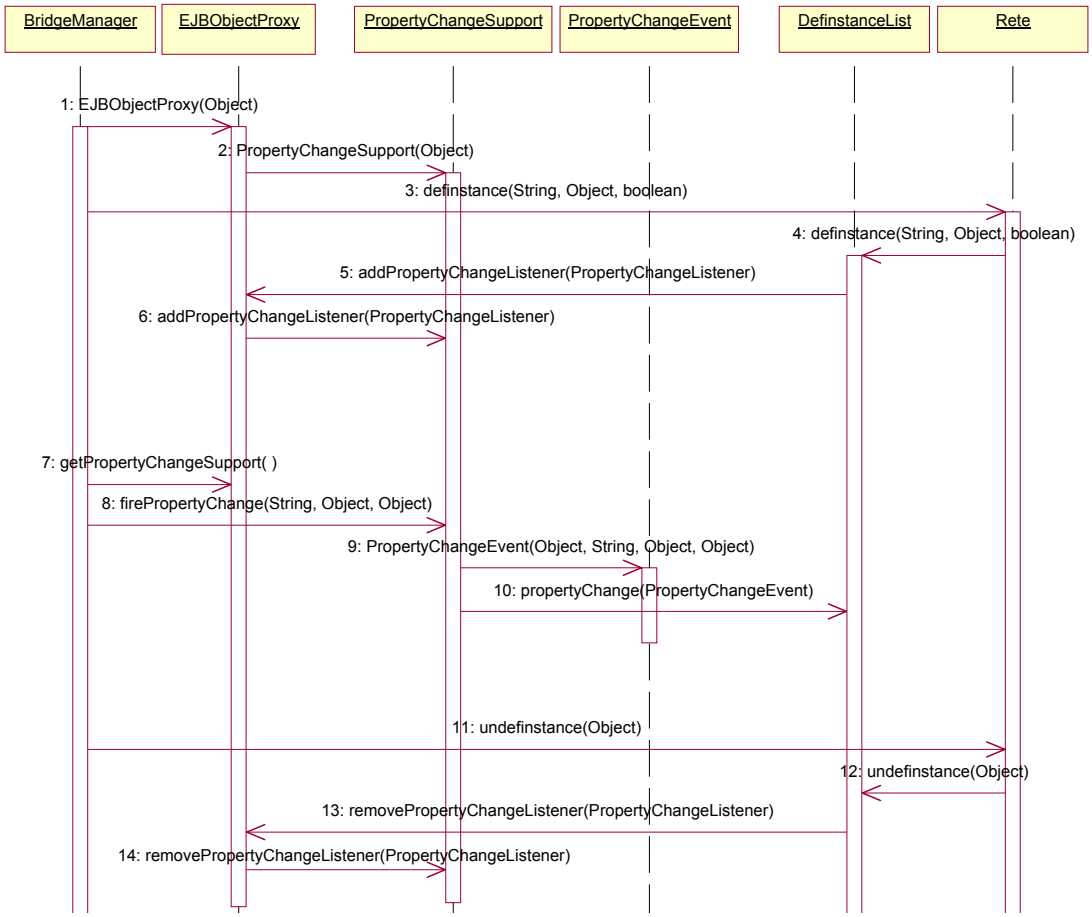

Abbildung 4.2: Sequenzdiagramm des Benachrichtigungsmechanismus'

4.1.2.1 isassert

Klasse: com.intershop.enfinity.cartridges.businessrules.jess.ISAssert

Syntax: (**isassert** <serverName> <objectID> <key> <intValue> <stringValue>)

wobei <serverName>::= ("eTS"|"eCS")

Beschreibung: isassert erzeugt ein neues oder modifiziert ein bestehendes Fact-Objekt[5].

Beispiel: Setze für einen süddeutschen Käufer das entsprechende Verkaufsgebiet.

```
(defrule setzeVerkaufsgebietSüddeutschland
    (BasicAddress (buyerID ?buyerID)
```

[5]siehe Abschnitt 3.3.5

```
        (postalCode ?postalCode &: (and
          (>= ?postalCode 70000) (< ?postalCode 90000)))
      (not (Fact (objectID ?buyerID) (key "Verkaufsgebiet")
        (stringValue "Süddeutschland")))
=>
      (isassert "eTS" ?buyerID "Verkaufsgebiet" 0
        "Süddeutschland")
      (commit)
)
```

4.1.2.2 isassertsession

Klasse: com.intershop.enfinity.cartridges.businessrules.jess.ISAssertSession

Syntax: (**isassertsession** <serverName> <sessionID> <key> <intValue> <stringValue>)

wobei <serverName>::= ("eTS "|"eCS")

Beschreibung: isassertsession erzeugt ein neues oder modifiziert ein bestehendes SessionFact-Objekt[6].

Beispiel: Überprüfe PLZ-Benutzereingabe

```
(defrule ueberpruefePLZ
      (SessionFact (objectID ?sessionID) (key "PLZ")
        (intValue ?plz &: (or (< ?plz 0) (> ?plz 99999)))
=>
      (isassertsession "eTS" ?sessionID "PLZFehler" 0
        "Bitte geben Sie eine richtige Postleitzahl an.")
      (commit)
)
```

4.1.2.3 ejbretract

Klasse: com.intershop.enfinity.cartridges.businessrules.jess.EJBRetract

Syntax: (**ejbretract** <factVariable>)

Beschreibung: Eliminiert ein bestehendes Geschäftsobjekt aus der Faktenbasis der Regelmaschine und löscht das Geschäftsobjekt aus dem Application Server / Datenbank.

Beispiel: Lösche alle Adressen, deren Vornamenattribut den Wert „Matthias“ haben.

[6] siehe Abschnitt 3.3.5

```
(defrule löscheBasicAddressMatthias
     ?address <- (BasicAddress (firstName "Matthias"))
=>
     (ejbretract ?address)
     (commit)
)
```

4.1.2.4 commit

Klasse: com.intershop.enfinity.cartridges.businessrules.jess.Commit

Syntax: (**commit**)

Beschreibung: Zwingender Abschluss einer Regel – die commit-Funktion realisiert den Stopp und den Start der Regelauswertung[7].

4.1.3 Steuerung von Jess

Zur Steuerung der integrierten Regelmaschine wurde ein Web-Interface realisiert. Das Web-Interface umfasst die gleiche Funktionalität wie die Console des Jess-Systems. Mittels des Web-Interfaces können Regeln und Funktionen in der Jess-Sprache definiert und gelöscht werden.

Abbildung 4.3 zeigt das Web-Interface bei einer ungültigen Eingabe. Das gezeigte Textfeld bietet die Möglichkeit zur Eingabe der Befehle, der „Execute Rete Command“-Button setzt den Befehl an die Regelmaschine ab und im Fehlerfall wird unten die von der Regelmaschine erzeugte Fehlermeldung angezeigt.

4.2 Proxyobjekte

Die vorgestellte Integration eines Aktionen- und Regelprozessors benutzt Proxyobjekte, um auf die Geschäftsobjekte zuzugreifen. Die folgenden Abschnitte zeigen auf, wie die Generierung der Proxyklassen und die proxyklassenabhängige Konfiguration in der Realisierung implementiert wurden.

Die Konzeption wurde in der Realisierung um die Möglichkeit der Konfiguration der durch die Regelmaschine überwachten Geschäftsobjekte erweitert. Das Format der Konfigurationsdatei wird in Abschnitt 4.2.2 beschrieben.

[7] siehe Abschnitt 3.3.4.4

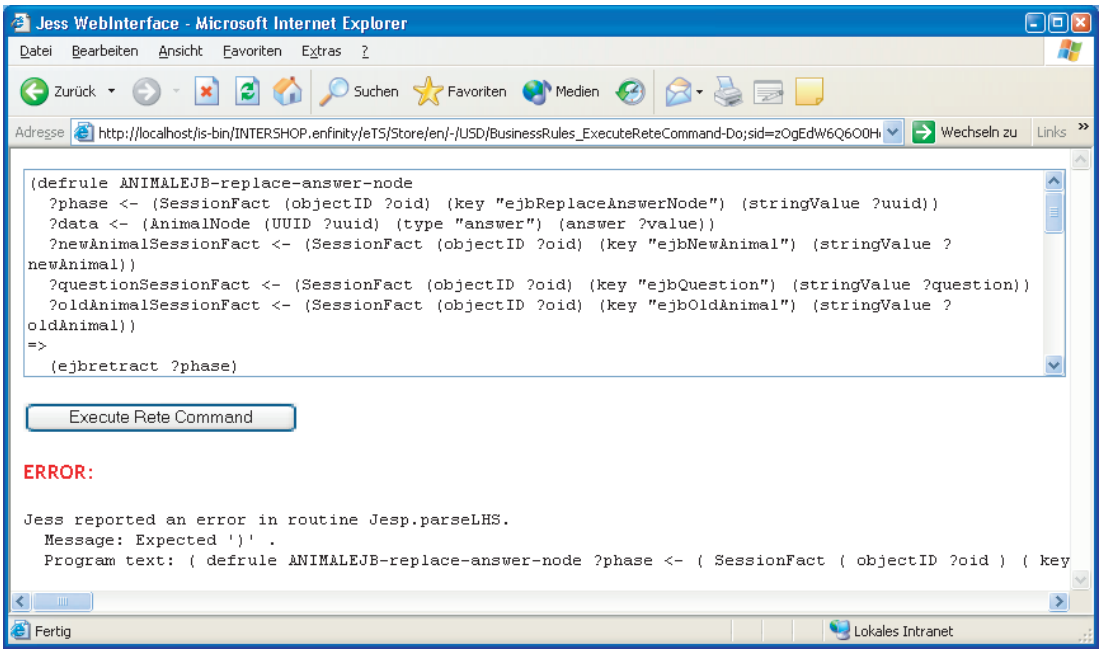

Abbildung 4.3: Web-Interface zur Steuerung des integrierten Jess-Systems

4.2.1 Generierung der Proxyklassen

In Abschnitt 3.3.2.3 wurde die realisierungsunabhängige Konzeption der Proxyklassengenerierung vorgestellt. Diese Konzeption wurde in Abschnitt 4.1.1 durch die von der verwendeten Regelmaschine abhängigen Mechanismen erweitert. Dieser Abschnitt beschreibt die konkrete Implementierung der Proxyklassengenerierung.

Sämtliche generierten Proxyklassen erben den Mechanismus zur Benachrichtigung der Regelmaschine von der Klasse *com.intershop.enfinity.cartridges.businessrules.bridge.proxy.EntityProxy*. In der EntityProxy-Klasse sind alle Attribute und Methoden implementiert, die zur Verwaltung der Proxyobjekte und zur Benachrichtigung der Regelmaschine notwendig sind. Die generierten Klassen implementieren ausschließlich Methoden zum Lesen (get-Methode) oder zum Setzen (set-Methode) der Attribute.

Die get- und die set-Methoden der Proxyobjekte leiten die Anfrage an das BridgeManager-Objekt[8] weiter. Das BridgeManager-Objekt leitet wiederum den Aufruf an das **Dispatcher**-Objekt des entsprechenden enfinity-Servers weiter. Da der Aufruf der Methode des Dispatcher-Objektes über das Netzwerk geschieht, müssen **RemoteException**s entweder verarbeitet oder an die aufrufende Methode weitergeleitet werden. Die hier vorgestellte Lösung leitet diese Fehler an die

[8]siehe Abschnitt 3.3.1

Regelmaschine[9] weiter, dazu ist es notwendig, dass die get- bzw. set-Methoden diese Fehler ebenfalls an die aufrufenden Methoden weiterleiten (dies geschieht durch das Anfügen von „throws java.rmi.RemoteException“ an den Methodenkopf).

Innerhalb des enfinity-Systems werden die Implementierungen der Remoteobjektklassen wie die BridgeManager-Klasse mit <**Remoteinterface**>**Impl** benannt. Das BridgeManager-Objekt ist als Singletonexemplar [GHJV96, S. 157] ausgelegt. Dies stellt sicher, dass nur ein Objekt diesen Typs innerhalb des Servers existiert. Der Zugriff auf dieses Objekt erfolgt über die Klassenmethode **getInstance()**. Zusammengefasst ergibt sich also folgender Zugriff auf das BridgeManager-Objekt:

```
BridgeManagerImpl.getInstance()
```

Der Konstruktor jeder Proxyklasse hat die einfache Struktur:

```
<KNAME>Proxy(Object primaryKey, int dispatcherID)
{
     super(primaryKey, dispatcherID);
}
```

Das Argument *primaryKey* und *dispatcherID* wird wiederum im Konstruktor der EntityProxy-Klasse verarbeitet.

Zum Lesen eines Attributes wird eine get-Methode generiert. Die Struktur der get-Methode ist abhängig davon, ob das Attribut ein Objekt bzw. Feld[10] ist oder ob es sich bei dem Attribut um einen nativen Typ[11] handelt. Handelt es sich bei dem Attribut um ein Objekt oder Feld, wird folgende get-Methode erzeugt:

```
public <TYP> get<ANAME>() throws java.rmi.
  RemoteException
{
     return (<TYP>) BridgeManagerImpl.getInstance().
       getAttribute(dispatcherID, getClass(),
         "<aNAME>");
}
```

Ist ein Attribut von einem nativen Typ, so wird diese get-Methode erzeugt:

[9] die Regelmaschine ruft die get- bzw. set-Methoden auf

[10] Felder, auch von nativen Typen, sind in Java Objekte

[11] **native Java-Typen:** boolean, byte, char, short, int, long, float, double

```
public <NTYP> get<ANAME>() throws java.rmi.
  RemoteException
{
    return ( (<OTYP>) BridgeManagerImpl.getInstance().
      getAttribute(dispatcherID, getClass(),
        "<aNAME>") ).<NTYP>Value();
}
```

Bei dem Typ *OTYP* handelt es sich um die Objektrepräsentation des dazugehörenden nativen Typs *NTYP*. Zum Beispiel ist es für den Typ *int* die Klasse *java.lang.Integer*, für *boolean java.lang.Boolean*. Jede dieser Klassen, die einen nativen Typ repräsentieren, beinhaltet eine Methode mit der Signatur *public <NTYP> <NTYP>Value()*, z. B. die Klasse java.lang.Integer die Methode *public int intValue()*, die Klasse java.lang.Boolean die Methode *public boolean booleanValue()*. Mit dieser Methode kann aus der Objektrepräsentation eine native Repräsentation des Typs erzeugt werden. Die JavaBeans-Spezifikation schreibt vor, dass für ein Attribut ***n**ame* mittels der Methoden *get**N**ame* und *set**N**ame* gelesen bzw. geschrieben werden kann. Die unterschiedliche Schreibweise des Attributnamens wird durch die unterschiedlichen Platzhalter <ANAME> bzw. <aNAME> dargestellt.

Analog zu den get-Methoden wird im Objekttyp-Fall folgende set-Methode erzeugt:

```
public void set<ANAME>(<TYP> value) throws java.rmi.
  RemoteException
{
    BridgeManagerImpl.getInstance().
      setAttribute(dispatcherID, getClass(),
        "<aNAME>", value);
}
```

Ist ein Attribut von einem nativen Typ, so wird diese set-Methode erzeugt:

```
public void set<ANAME>(<NTYP> value) throws java.rmi.
  RemoteException
{
    BridgeManagerImpl.getInstance().
      setAttribute(dispatcherID, getClass(),
        "<aNAME>"), new <OTYP>(value) );
}
```

Die Vererbungshierarchie der Business Objects muss auf die Proxyklassen abgebildet werden. Das heißt, wenn eine Geschäftsobjektklasse von einer anderen Geschäftsobjektklasse abgeleitet ist, so muss auch die Proxyobjektklasse der

Geschäftsobjektklasse von der Proxyobjektklasse der Geschäftsobjektklasse abgeleitet sein. Der Gesamtalgorithmus für die Generierung des Klassen lässt sich dann folgendermaßen skizzieren:

1. Suchen der Klassen im Klassenpfad, die eine Enterprise JavaBean realisieren.
2. Aufbau eines Vererbungsbaumes dieser Klassen
3. **Für alle Klassen im Vererbungsbaum:**
   (a) Generierung der Proxyklasse, unter Berücksichtigung der Vererbungshierarchie
      i. Feststellen der Properties (Attribute mit get- und/ oder set-Methoden)
      ii. **Für alle Properties der Klasse:**
         A. Generierung der get- und/oder der set-Methoden für das Property
      iii. Generierung des Eintrages in die Konfigurationsdatei

Die Vererbungshierarchie der erzeugten Proxyklassen wird in Abschnitt B.3 dargestellt. Dieser Vererbungsbaum enthält alle Proxyklassen der Geschäftsobjektklassen, die mit dem enfinity-System ausgeliefert werden und der Geschäftsobjektklassen, die im Rahmen der Diplomarbeit entstanden sind.

Die Realisierung benutzt Einträge in eine Konfigurationdatei um zum Beispiel festzulegen, welche Proxyobjektklasse zu welcher Geschäftsobjektklasse gehört, auf welchem enfinity-Server sich diese Geschäftsobjekte befinden usw. Bei der Generierung werden solche Konfigurationseinträge soweit möglich miterzeugt, dies soll die Konfiguration des Systems vereinfachen. Der genaue Aufbau dieser Konfigurationsdatei wird im folgenden Abschnitt dargestellt.

4.2.2 Konfiguration der überwachten Business Object Menge

Die Konfiguration der überwachten Business Object Menge wird in der zentralen Konfigurationsdatei *businessrules.properties* vorgenommen. Die Auswahl der Geschäftsobjekte geschieht mittels der Schlüssel *ejbAbstractNames* und *ejbHomeNames*. Sie haben folgenden Syntax:

ejbAbstractNames = (<AEJBHomeName>)*
ejbHomeNames = (<EJBHomeName>)*

wobei <EJBHomeName> und <AEJBHomeName> der Name der EJB[12]ist, mit dem das EJBHome-Interface an der Registry des enfinity-Systems registriert wird.

Bei den *ejbAbstractNames* handelt es sich um die Namen der abstrakten Enterprise JavaBeans. Alle anderen Enterprise JavaBeans, die von der Regelmaschine überwacht werden sollen, werden unter dem Schlüssel *ejbHomeNames* angegeben.

Für jede der *abstrakten* Enterprise JavaBeans müssen wiederum folgende Schlüssel-Wert-Paare in der Konfigurationsdatei vorhanden sein:

<**AEJBHomeName**>**.proxyClassName** =
<Proxyklassenname>
<**AEJBHomeName**>**.templateName** =
<Jess-Templatename>
<**AEJBHomeName**>**.extends** = <Jess-Supertemplatename>

wobei <Proxyklassenname> der vollqualifizierte Klassenname der dazugehörenden Proxyklasse ist, <Jess-Templatename> der Name ist, der innerhalb von Jess verwendet wird und <Jess-Supertemplatename> der Name des Jess-Templates ist, von dem dieses Template abgeleitet worden ist.

Für jede der instanzierbaren Enterprise JavaBeans müssen **zusätzlich** folgende Schlüssel-Wert-Paare vorhanden sein:

<**EJBHomeName**>**.servers** = ("eTS"|"eCS"|"eTS""eCS"|
"eCS""eTS")
<**EJBHomeName**>**.loadOnServers** = ("eTS"|"eCS"|
"eTS""eCS"|"eCS""eTS")

Mit dem <*EJBHomeName*>*.servers*-Schlüssel wird angegeben, auf welchem Server sich diese Geschäftsobjekte befinden, dies können der enfinity Transactivity Server oder der enfinity Catalog Server oder beide sein. Der <*EJBHomeName*>*.loadOnServers*-Schlüssel gibt an, welche Server während der Startphase des enfinity-Systems die Geschäftsobjekte des Typs der Regelmaschine bekannt machen. Die Werte von <EJBHomeName>.servers und <EJBHomeName>.loadOnServers sind im Normalfall gleich.

[12]entspricht der Konstanten <**EJBHomeInterface**>**.HOME_NAME**

4.2.3 Instanzierung eines neuen Proxyobjektes

Wird ein neues Geschäftsobjekt erzeugt, so muss innerhalb des BusinessRules-Servers ein neues Proxyobjekt für dieses Geschäftsobjekt erstellt werden. Dieser Abschnitt zeigt die Erstellung eines neuen Proxyobjektes auf.

Zur Erstellung wird der Reflection-Mechanismus[Sun02c, Sun99c] der Java-Plattform verwendet. Dieser ermöglicht es, neue Instanzen von Objekten zu erzeugen, deren Typ noch nicht während des Kompiliervorgangs bekannt ist. Die Proxyklasse einer Geschäftsobjektklasse wird durch den Schlüssel <**EJBHomeName**>**.proxyClassName** in der Konfigurationdatei des Systems angegeben. Folgender Algorithmus zeigt die Schritte, die notwendig sind, um ein Proxyobjekt zu erzeugen. Die Variable **proxyClassName** (Typ: java.lang.String) enthält den Klassennamen der Proxyklasse, die Variable **primaryKey** (Typ: java.lang..Object) enthält den primären Schlüssel des EJB-Objektes, für das dieses Proxyobjekt erzeugt wird, die Variable **dispatcherID** (Typ: int) enthält die Kennung des Dispatcher-Objektes, das auf dem enfinity-Server installiert ist, auf dem sich das Orginalobjekt des Proxyobjektes befindet. Zur besseren Übersichtlichkeit werden die Erklärungen der einzelnen `Codefragmente` über diesen dargestellt.

Hole Referenz auf Proxyklasse mit dem Namen "proxyClassName"

```
Class proxyClass = Class.forName(proxyClassName);
```

Erzeuge Signatur "constructorSignature" des Konstruktors in Form eines Parametertypfeldes

```
Class[] constructorSignature = new Class[2];
constructorSignature[0] = Class.forName("java.lang.
Object");
constructorSignature[1] = Integer.TYPE;
```

Hole Referenz auf Konstruktor mit dieser Signatur

```
java.lang.reflect.Constructor constructor = proxyClass.
getConstructor(constructorSignature);
```

Erzeuge Feld mit den Parametern "primaryKey" und "dispatcherID"

```
Object[] constructorArguments = new Object[2];
constructorArguments[0] = primaryKey;
constructorArguments[1] = new Integer(dispatcherID);
```

Erzeuge neues Proxyobjekt "proxyObject"

```
Object proxyObject =
constructor.newInstance(constructorArguments);
```

4.3 EJB-Methodenaufrufe

Sämtliche Proxyobjekte leiten die get- bzw. set-Methoden an das BridgeManager-Objekt und das wiederum an das Dispatcher-Objekt des entsprechenden enfinity-Servers weiter. In diesem Abschnitt wird beschrieben, wie in der Realisierung der Aufruf der **getAttribute**- und der setAttribute-Methode des Dispatcher-Objektes in einen Aufruf der **get<AttributName>**- bzw. **set<AttributName>**-Methode des entsprechenden EJB-Objektes umgewandelt wird.

Während der Startphase des enfinity-Systems werden die Property-Eigenschaften der konfigurierten Enterprise JavaBeans[13] untersucht. Zur Untersuchung der EJB-Klassen wird der Introspection-Mechanismus des JavaBean-Frameworks [Sun97, S. 54] eingesetzt und eine Datenstruktur erzeugt, die folgendes Aussehen besitzt:

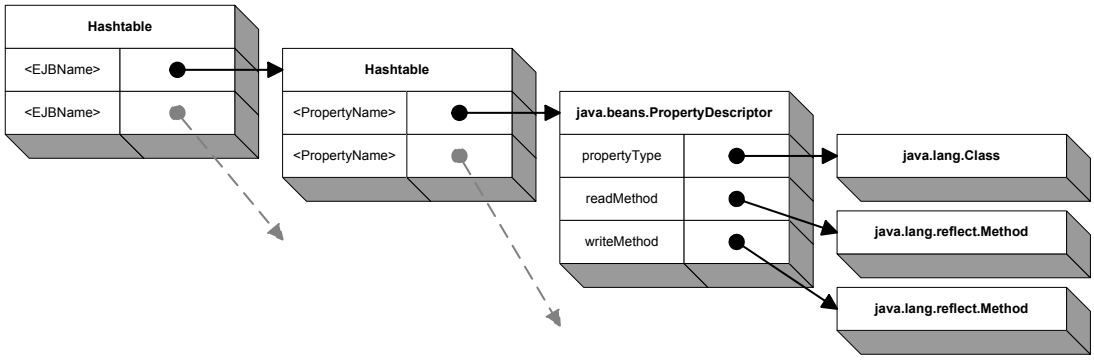

Abbildung 4.4: Datenstruktur für den Zugriff auf die EJB-Methoden

Die Datenstruktur erlaubt für jedes Attribut einer konfigurierten Enterprise JavaBean den Zugriff auf die get- bzw. set-Methode sowie auf den Typ des Attributes. Die Klasse *java.lang.reflect.Method* ist Teil der Reflection API [Sun02c, Sun99c], die mit der Java 2 Standard Edition Plattform ausgeliefert wird. Der Aufruf der get-Methode erfolgt mittels

```
Object returnValue = getMethod.invoke(ejbObject, null);
```

wobei **getMethod** das get-Methoden-Objekt für das entsprechende Property und **ejbObject** das EJB-Object ist, auf das die get-Methode zugreift.

Analog dazu wird die set-Methode mittels

```
setMethod.invoke(ejbObject, arguments);
```

[13] siehe Abschnitt 4.2.2

aufgerufen. Hierbei ist **setMethod** wiederum das set-Methoden-Objekt für das Property, **ejbObject** das EJB-Objekt und **arguments** ein Feld, das als einziges Element den neuen Wert des Attributs beinhaltet. Ist das Attribut von einem nativen Java-Typ, so wird wiederum die Objekt-Repräsentation diesen Typs zur Parameterübergabe benutzt.

Kapitel 5

Leistungsanalyse und Anwendung

5.1 Performancetest

Um die Leistungsfähigkeit der vorgestellten Konzeption und Realisierung zu testen, wurde die Ausführungszeit von elementaren Anwendungsfällen gemessen. Als „Vergleichswert“ wurde die Ausführungszeit von Pipelets, die die gleiche Funktion wie die jeweilige Regel erfüllen, bestimmt. **Zu beachten ist**, dass der Test durch Ausführung eines entsprechenden Pipelets ein *einmaliger, von außen angestoßener* Vorgang aus Prüfung der Vorbedingungen und entsprechende Aktionsausführungen ist. Im Gegensatz dazu wird eine Geschäftregel ausgeführt, *sobald* eine Vorbedingung erfüllt ist. Durch die unterschiedlichen Paradigmen können die Ausführungszeiten nicht als alleinige Grundlage einer Besser-als-Relation dienen.

In den Testfällen Geschäftsobjektmodifikation und -löschung wurden zwei Varianten der jeweiligen Pipelets erstellt. Die Variante *Enumeration* durchsucht alle im System vorhanden Geschäftsobjekte des entsprechenden Typs nach den jeweiligen Eigenschaften. Während die Variante Enumeration J2EE-konform ist, nutzt die *SQLWhere*-Variante die Erweiterung des enfinity-Application Servers, die es erlaubt, mittels SQL-Where-Ausdrücken die Erfüllungsmenge direkt zu bestimmen.

Sämtliche Tests operieren auf dem Geschäftsobjekttyp *TestEJB*. Dieser Typ enthält neben den geerbten Attributen UUID (String), domainID (String), oca (long), lastModified (java.util.Date) die Attribute testName (String) und testInt (int). Zu diesen Attributen kommen nochmals vier Attribute vom Typ boolean hinzu, diese zeigen an, ob die mit NULL belegbaren Attribute lastModified, oca, testInt und testString ein Wert zugewiesen wurde oder nicht. Eine Optimierung der Proxy-Objekte wurde nicht vorgenommen, es sind also alle zehn Attribute des Typs von der Regelmaschine aus sichtbar.

Alle Tests wurden auf einem Rechner, der sämtliche Server (eTS, eCS und BusinessRules-Server) sowie die Datenbank beinhaltet, ausgeführt. Trotz der Bemühung, eine objektive Testumgebung zu schaffen, kann nicht ausgeschlossen werden, dass es zu Messfehlern gekommen ist. Auf dem Testrechner werden den unterschiedlichen Prozessen mal mehr und mal weniger Rechenzeit zu Verfügung gestellt, Speichersegmente können ausgelagert sein, vorhandene Caching-Mechanismen können Operationen beschleunigen oder verzögern usw. – all dies kann sich auf die Testergebnisse auswirken. Um dennoch einigermaßen verlässliche Ergebnisse zu erhalten, wurde der Durchschnitt aus jeweils fünf Testläufen gebildet. Zur Messung der Ausführungszeit der jeweiligen Regel wurden zwei zusätzliche Regeln definiert. Jede Zusatzregel wird pro Test genau einmal ausgeführt, was zu einem, je nach Anzahl der Regelausführungen, mehr oder weniger signifikanten Messfehler führt.

5.1.1 Testfall: Geschäftsobjekterzeugung

5.1.1.1 Beschreibung

Es werden 100 neue Geschäftsobjekte vom Typ *TestEJB* erzeugt.

5.1.1.2 Regeldefinition

```
1  (defrule TEST-performance-do
2   ;; Schleifeninvariante
3    ?ctef <- (createdTestEJB ?cte&:(< ?cte 100) )
4  =>
5   ;; neue Transaktion
6    (bind ?transaction (call ?*bridgeManager*
        getUserTransaction "eTS") )
7   ;; beginne Transaktion
8    (call ?transaction begin)
9   ;; hole defaultDomain
10   (bind ?domainHome (call ?*bridgeManager* getEJBHome "eTS
        " "DomainHome") )
11   (bind ?domain (call ?domainHome findDefaultDomain) )
12  ;; hole TestEJBHome
13   (bind ?testEJBHome (call ?*bridgeManager* getEJBHome "eTS
        " "TestEJBHome") )
14  ;; erzeuge neue TestEJB
15   (bind ?testEJB (call ?testEJBHome create ?domain) )
16  ;; setze Werte der TestEJB
17   (set ?testEJB testName "dummy")
18   (set ?testEJB testInt 333)
```

```
19  ;; erhoehe Zaehler
20   (retract ?ctef)
21   (assert (createdTestEJB (+ ?cte 1)) )
22  ;; schreibe Transaktion fest
23   (call ?transaction commit)
24  ;; beende Aktionsausfuehrung
25   (commit)
26 )
```

5.1.1.3 Testergebnis

Pipelet		Geschäftsregel	
ms	**%**	**ms**	**%**
3534	**100**	8322	235,5

Tabelle 5.1: Zeit für die Erzeugung von 100 Geschäftsobjekten

5.1.1.4 Erläuterung

Der höhere Zeitaufwand ergibt sich aus folgenden Operationen, die nur bei der Geschäftsregelversion notwendig sind:

- Erzeugen eines Proxyobjektes für das neu erzeugte Geschäftsobjekt
- Benachrichtigung der Regelmaschine über das neu erzeugte Objekt
- Verarbeitung durch die Regelmaschine
  - Lesen der Attribute des Objekts
  - Zugriff auf das Orginalobjekt durch das Lesen der Attribute
  - Verarbeitung durch das Rete-Netzwerk
  - Ermittlung der nächsten auszuführenden Regel aus der Konfliktmenge
  - Ausführung der Aktion
- Synchronisation der Aktion für Faktenbasiskonsistenz

Das Einfügen des Proxyobjektes in das Faktenwissen der Regelmaschine verlängert den schon an sich relativ aufwendigen Vorverarbeitungsschritt[1] nochmals, da zuerst ein neues Proxyobjekt erzeugt werden muss und während des Vorverarbeitungsschrittes ein Zugriff auf die Attribute des Orginalobjektes, der mittels eines aufwendigen entfernten Methodenaufrufes abgewickelt wird, stattfindet.

[1] siehe Abschnitt 2.4.4

5.1.2 Testfall: Einfache Geschäftsobjektmodifikation

5.1.2.1 Beschreibung

Die Attribute testName und testInt der Geschäftsobjekte vom Typ *TestEJB* werden geändert, falls testName = „TestPerformance“ und testInt = 3.

5.1.2.2 Regeldefinition

```
(defrule TEST-performance-do
 ;; Vorbedingung
  ?te <- (TestEJB (testName "TestPerformance") (testInt 3)
      )
 =>
 ;; setze neue Werte - implizite Transaktion
  (modify ?te (testName "TestPerformanceNEU") (testInt 0) )
 ;; beende Aktionsausfuehrung
  (commit)
)
```

5.1.2.3 Testergebnis

Anzahl musterfüllender Objekte		**Pipelet**				**Geschäftsregel**	
		Enumeration		**SQLWhere**			
absolut	**%**	**ms**	**%**	**ms**	**%**	**ms**	**%**
1 von 1000	0,1	861	4305	20	**100**	120	600
10 von 1000	1	982	755,4	130	**100**	1372	1055,4
100 von 1000	10	2563	151,5	1692	**100**	22945	1356,1

Tabelle 5.2: Zeit für die Modifikation von Geschäftsobjekten – einfache Regel

5.1.2.4 Erläuterung

Je häufiger eine Regel ausgeführt wird, desto stärker schlägt sich bei der Geschäftsregelversion die Kommunikation zwischen BusinessRules-Server und enfinity Server, die notwendige Synchronisation zur Faktenbasiskonsistenz usw., im Ergebnis nieder. Die relative Verbesserung der Enumerationversion ergibt sich aus der höheren Trefferrate, d.h bei „1 von 1000“ wurde 999 Objekte negativ getestet, bei „100 von 1000“ nur 900.

5.1.3 Testfall: Komplexe Geschäftsobjektmodifikation

5.1.3.1 Beschreibung

Die Attribute testName und testInt der Geschäftsobjekte vom Typ *TestEJB* werden geändert, falls testName = „TestPerformance“ und testInt = 3 und es Geschäftsobjekte vom Typ *TestEJB1* mit den Attributwerten testName = „TestPerformance1“ und testInt = 1, vom Typ *TestEJB2* mit den Attributwerten testName = „TestPerformance2“ und testInt = 2, vom Typ *TestEJB3* mit den Attributwerten testName = „TestPerformance3“ und testInt = 3, vom Typ *TestEJB4* mit den Attributwerten testName = „TestPerformance4“ und testInt = 4, vom Typ *TestEJB5* mit den Attributwerten testName = „TestPerformance5“ und testInt = 5, vom Typ *TestEJB6* mit den Attributwerten testName = „TestPerformance6“ und testInt = 6, vom Typ *TestEJB7* mit den Attributwerten testName = „TestPerformance7“ und testInt = 7, vom Typ *TestEJB8* mit den Attributwerten testName = „TestPerformance8“ und testInt = 8 und vom Typ *TestEJB9* mit den Attributwerten testName = „TestPerformance9“ und testInt = 9 gibt.

Pro Geschäftsobjekttyp (*TestEJB*, *TestEJB1*, ... , *TestEJB9*) gibt es jeweils ein Geschäftsobjekt, das das Muster erfüllt und 999, die es nicht erfüllen.

Anmerkung:
Bei der Realisierung des Tests durch ein Pipelet befanden sich alle Geschäftsobjekte auf dem Server, auf dem der Test ausgeführt wurde. Würde sich zum Beispiel das Geschäftsobjekt *TestEJB* auf dem enfinity Transactivity Server und das Geschäftsobjekt *TestEJB1* auf dem enfinity Catalog Server befinden, so müsste man eine Anfrage an den anderen Server realisieren, mit der auf die dem Muster entsprechenden Objekte zugegriffen werden kann. Diese Anfrage kostet Zeit, die in diesem Test **nicht** berücksichtig wird. Durch die symmetrische Konzeption der Kommunikation des BusinessRules-Servers (siehe Abbildung 3.5) ist bei einer Geschäftsregel, die auf Objekte des eTS und des eCS zugreift, diese Verschlechterung nicht zu beobachten.

5.1.3.2 Regeldefinition

```
1 (defrule TEST-performance-do
2  ;; Vorbedingung
3   (TestEJB1 (testName "TestPerformance1") (testInt 1) )
4   (TestEJB2 (testName "TestPerformance2") (testInt 2) )
5   (TestEJB3 (testName "TestPerformance3") (testInt 3) )
6   (TestEJB4 (testName "TestPerformance4") (testInt 4) )
7   (TestEJB5 (testName "TestPerformance5") (testInt 5) )
8   (TestEJB6 (testName "TestPerformance6") (testInt 6) )
```

```
9    (TestEJB7 (testName "TestPerformance7") (testInt 7) )
10   (TestEJB8 (testName "TestPerformance8") (testInt 8) )
11   (TestEJB9 (testName "TestPerformance9") (testInt 9) )
12   ?te <- (TestEJB (testName "TestPerformance") (testInt 3)
         )
13 =>
14  ;; setze neue Werte - implizite Transaktion
15   (modify ?te (testName "TestPerformanceNEU") (testInt 0) )
16  ;; beende Aktionsausfuehrung
17   (commit)
18 )
```

5.1.3.3 Testergebnis

Anzahl musterfüllender Objekte		**Pipelet**				**Geschäftsregel**	
		Enumeration		**SQLWhere**			
absolut	**%**	**ms**	**%**	**ms**	**%**	**ms**	**%**
1 von 1000	0,1	1662	2026,8	82	**100**	90	109,8

Tabelle 5.3: Zeit für die Modifikation von Geschäftsobjekten – komplexe Regel

5.1.3.4 Erläuterung

Je komplexer eine Geschäftsregel ist, desto besser ist die relative Geschwindigkeit der Geschäftsregelversion. Bei der Pipeletversion erhöht sich die Zeit, die benötigt wird, um die mustererfüllenden Objekte zu bestimmen, da auf zehn verschiedenen EJBHome's der Selektionsprozess ausgeführt werden muss. Bei der Geschäftsregelversion wird schon bei der Erzeugung/Änderung eines Geschäftsobjektes überprüft, ob das Objekt bestimmte Muster erfüllt und an den jeweiligen Knoten im Rete-Netzwerk eingeordnet.

5.1.4 Testfall: Geschäftsobjektlöschung

5.1.4.1 Beschreibung

Geschäftsobjekte vom Typ *TestEJB* werden gelöscht, falls die Attribute testName und testInt mit den Werten „TestPerformance“ und 3 belegt sind.

5.1.4.2 Regeldefinition

```
1 (defrule TEST-performance-do
```

```
2  ;; Vorbedingung
3   ?te <- (TestEJB (testName "TestPerformance") (testInt 3)
        )
4 =>
5  ;; loesche EJB - implizite Transaktion
6   (ejbretract ?te)
7  ;; beende Aktionsausfuehrung
8   (commit)
9 )
```

5.1.4.3 Testergebnis

Anzahl musterfüllender Objekte		Pipelet				Geschäftsregel	
		Enumeration		SQLWhere			
absolut	%	ms	%	ms	%	ms	%
1 von 1000	0,1	250	2500	10	**100**	30	300
10 von 1000	1	270	450	60	**100**	90	150
100 von 1000	10	912	126,5	721	**100**	731	101,4

Tabelle 5.4: Zeit für die Löschung von Geschäftsobjekten

5.1.4.4 Erläuterung

Je häufiger Geschäftsobjekte gelöscht werden, desto größer ist der relative Anteil des Löschungsprozesses gegenüber dem Selektionsprozess. Dies führt zu der relativen Annäherung der gemessenen Zeiten. Gegenüber des in Abschnitt 5.1.2 beschriebenen Testfalls entfällt bei der Löschung eines Objektes bei der Geschäftsregelversion die Kommunikation zwischen BusinessRules-Server und enfinity Server zur Bestimmung der Attributsänderungen und die Verarbeitung durch das Rete-Netzwerk.

5.2 Anwendungsgebiete

Die in der vorliegenden Arbeit entstandene Lösung fügt dem aktuellen objekt-/prozessorientierten Paradigma das objekt-/regelbasierte Paradigma zur Erstellung von E-Commerce-Systemen hinzu. Wie jedes Paradigma hat auch das objekt-/regelbasierte Paradigma spezifische Vor- und Nachteile und eignet sich für bestimmte Anwendungsgebiete mehr und für andere weniger.

Im Folgenden werden die Anwendungsgebiete, die sich besonders gut mit dem objekt-/regelbasierte Paradigma bearbeiten lassen und somit mit der Business

Rules-Komponente implementieren lassen, dargestellt. Die Anwendungsgebiete sind sicherlich nur ein Ausschnitt der gesamten Anwendungsmöglichkeiten, einige weitere Anwendungsgebiete ergeben sich sicherlich durch die Anwendung des Paradimas und durch die Konfrontation mit konkreten Problemstellungen, die sich mit dem objekt-/regelbasierten Paradigma gut bzw. besser als mit den herkömmlichen Paradigmen lösen lassen.

5.2.1 Geschäftsprozesse mit hoher Änderungshäufigkeit

Das Hauptanwendungsgebiet der vorgestellten Konzeption ist sicherlich die Manipulation und Erzeugung von (Teil-)Geschäftsprozessen, die eine hohe Änderungshäufigkeit haben. Das gleiche Anforderungsprofil ergibt sich für Geschäftsprozesse, die zuerst erprobt werden. Es ist heute üblich, dass zum Beispiel Marketingaktionen und die dazugehörenden Geschäftsprozesse in einer Erprobungsphase evaluiert werden. In der Erprobungsphase soll zum Beispiel geklärt werden, ob durch das Versenden von Geburtstagsgeschenken an Kunden der Effekt einer höheren Kundenbindung eintritt.

Die vorgestellte Konzeption und die damit verbundene einfache Struktur des objekt-/regelbasierten Paradigmas erlaubt es, dass Geschäftsprozesse

- mit relativ geringem Aufwand
- auf hoher Abstraktionsebene (direkt von den zuständigen Experten)
- während des Betriebs

geändert und erzeugt werden können. Diese Eigenschaften decken sich mit den Hauptanforderungen, die bedingt durch die hohe Änderungshäufigkeit, von dynamischen Geschäftsprozessen[2] gestellt werden.

Zum momentanen Zeitpunkt ist jedoch aus folgenden Gründen von einer **rein** objekt-/regelbasierten Realisierung der Geschäftsprozesse abzuraten:

- höherer Ressourcenbedarf[3][4] der objekt-/regelbasierten Realisierung gegenüber der objekt-/prozessorientierten Realisierung
- das objekt-/prozessorientierte Paradigma kommt der momentan sehr verbreiteten Sichtweise der Unternehmensmodellierung sehr nahe

[2] *Anmerkung:* Im Gegensatz zu statischen Geschäftsprozessen, wie zum Bespiel Rechnungserstellung, die über Jahre bzw. Jahrzehnte die gleiche Struktur besitzen und die gleichen Informationen verarbeiten.

[3] hauptsächlich Rechenzeit und Kommunikationsvolumen

[4] siehe Abschnitt 5.1

- das objekt-/prozessorientierte Paradigma ist ausreichend erforscht
- das objekt-/prozessorientierte Paradigma erfüllt die Anforderungen, die zur Erstellung von statischen Geschäftsprozessen notwendig sind, sehr gut

Eine ideale Kombination ergibt sich zum jetzigen Zeitpunkt der Entwicklung durch die Verwendung des objekt-/prozessorientierten Paradigmas für die statischen Geschäftsprozesse und durch die Verwendung des objekt-/regelbasierten Paradigmas zur Realisierung von Geschäftsprozessen, die eine hohe Änderungshäufigkeit besitzen oder zur Erprobung ausstehen.

Das in Abschnitt C.2 dargestellte Beispiel *„Geschenk bei Bestellung am Geburtstag“*, demonstriert den Einsatz der BussinessRules-Komponente zur Erstellung von Geschäftsprozessen, die zur Evaluierung anstehen. Weiter wird in dem Beispiel aufgezeigt, wie ein durch das objekt-/prozessorientierte Paradigma erstellter Geschäftsprozess durch das objekt-/regelbasierte Paradigma geändert/erweitert werden kann.

5.2.2 Personalisierung

Unter Personalisierung versteht man Geschäftsprozesse, die käuferspezifische Attribute in Entscheidungsprozesse innerhalb des Geschäftsprozesses mit einbeziehen. Ein Beispiel hierfür ist das Empfehlen bestimmter Produkte anhand der Interessen oder der Herkunft eines Käufers.

Die für die Personalisierung erforderlichen Daten werden entweder explizit vom Benutzer abgefragt oder implizit durch dessen Verhalten auf der E-Commerce-Site erhoben. Bei der impliziten Datenerhebung wird zum Beispiel anhand der angeschauten Katalogseiten, der Reihenfolge der angeschauten Seiten (*engl.: click stream*), der Verweildauer auf bestimmten Seiten und der eingegebenen Suchbegriffe auf das Käuferprofil geschlossen.

Die Personalisierung eignet sich sehr gut für den Einsatz des objekt-/regelbasierten Paradigmas, da sich die Entscheidungsprozesse, ausgehend von den käuferspezifischen Attributen bis hin zum beworbenden Produkt, Produktgruppe, käuferabhängigen Preis usw. sehr gut durch Wenn-Dann-Regeln abbilden lassen.

5.2.2.1 Abgrenzung zu reinen Personalisierungskomponenten

Die heute verfügbaren Personalisierungskomponenten[5] für E-Commerce-Systeme basieren oft auf regelbasierten Mechanismen. Die Mehrzahl dieser Personalisierungskomponenten enthalten wie die entstandene Business Rules-Komponente

[5] wie zum Beispiel WebSphere Personalization [IBM02] für den IBM WebSphere Application Server und die BEA WebLogic E-Business Platform [BEA]

eine Regelmaschine. Die Regelmaschine wird meistens ausschließlich dazu eingesetzt, um **während** der Prozessabarbeitung regelbasiert Entscheidungen zu treffen. Dazu dienen das entsprechende Käuferobjekt und käuferspezifische Objekte als Eingangsparameter für die Regelauswertung. Als Ergebnis der Regelauswertung liegen dann im Falle des oben genannten Beispieles ein entsprechendes Produkt, das dem Käufer empfohlen wird, vor.

Im Gegensatz zu den Personalisierungskomponenten sieht die Diplomarbeit eine Manipulation von Geschäftsobjekten durch entsprechende Regeln vor. Dies ist bei der Mehrzahl der Personalisierungskomponenten nicht vorgesehen. Des Weiteren werden Regeln in der Business Rules-Komponeten nicht nur an einem definierten Punkt während der Prozessabarbeitung ausgeführt, sondern **sobald** eine Regel anwendbar ist.

Die Personalisierung ist **ein** Anwendungsgebiet der vorliegenden Arbeit, während die Personalisierungskomponenten ausschließlich für diesen Zweck entwickelt wurden. Eine weitergehende Nutzung der Personalisierungskomponenten ist entweder unmöglich oder nur sehr schwierig zu realisieren.

5.2.3 Beratungssysteme

Es gibt eine Reihe von Produkten, die nicht oder nur sehr schwierig durch eine Beschreibung der Produkteigenschaften verkaufbar sind. Man denke zum Beispiel an Versicherungen oder auch Baustoffe, die eine Reihe von Anforderungen erfüllen müssen. Diesen Produkten muss ein Beratungsprozess dem Verkaufsprozess vorangestellt werden. Einige Produkte sind hoch konfigurierbar, beim Kauf eines Autos stehen zum Beispiel nur bestimmte Innenausstattungen für bestimmte Lackfarben zur Verfügung oder der Käufer muss sich zwischen einem CD-Wechsler und einem Navigationssystem entscheiden, da beides nicht in das Auto integriert werden kann. Solche Produkte sind oftmals in mehreren 10000 unterschiedlichen Varianten verfügbar. Für solche hochkonfigurierbaren Produktgruppen ist es ratsam, einen, dem Beratungsprozess sehr ähnlichen Produktkonfigurationsprozess zur Verfügung zu stellen.

Solche Beratungs- und Produktkonfigurationsprozesse lassen sich sehr gut mit der BusinessRules-Komponente realisieren. Ein Vertriebsexperte stellt die nötigen Einzelentscheidungsmöglichenkeiten, die auch in einem Beratungs- oder Verkaufsgespräch gegeben sind, zusammen. Dabei entspricht die regelbasierte Realisierung weitgehend der Denkweise des Experten während eines Beratungs- oder Verkaufsgesprächs, die meistens auch in Form von Wenn-Dann-Regeln gegeben ist. Die Einzelentscheidungen bilden die Knoten eines Entscheidungsbaumes, dessen Kanten zu weiteren Entscheidungsmöglichkeiten führen, bis schließlich in den Blättern des Entscheidungsbaumes das vollkonfigurierte Produkt vorhanden ist.

Die regelbasierte Umsetzung solcher Entscheidungsbäume wird im Beispiel *„Tier-Rate-Spiel“* in Abschnitt C.3 demonstriert. Dieses Beispiel zeigt ebenfalls die Kommunikationsmöglichkeiten der BusinessRules-Komponente mit dem Pipeline Prozessor[6] auf.

5.2.4 Datenpflege

Falsche Eingaben durch Käufer oder Administratoren, Import von unvollständigen oder fehlerhaften Daten oder Duplikaten sorgen für eine zunehmende „Verunreinigung“ der Datenbasis [RD00]. Diese Verunreinigung führt zu Inkonsistenzen der Datenbasis und gefährdet im schlimmsten Fall die Systemstabilität. Als Datenpflege (*engl.: data cleansing, data cleaning, scrubbing*) bezeichnet man Vorgänge, die dieser Verunreinigung der Datenbasis entgegenwirken. Es können folgende grundsätzliche Datenpflegemechanismen unterschieden werden:

- Verhinderung, dass falsche Datensätze in die Datenbasis aufgenommen werden
- Korrektur/Löschung falscher/doppelter Datensätze aus der Datenbasis

Die Aufnahme falscher Daten in die Datenbasis kann dadurch verhindert werden, dass die zur Aufnahme anstehenden Datensätze verifiziert werden. So ist es zum Beispiel möglich, dass Adressdaten anhand von Name/Telefonnummer- oder Postleitzahl/Ort/Strasse-Kombinationen durch vorhandene Datenbanken (Telefon- und Strassenverzeichnisse in elektronischer Form) auf Richtigkeit überprüft und gegebenenfalls die korrigierten Daten dem Benutzer zur Bestätigung angezeigt werden.

Datenpflege durch Korrektur einer bestehenden Datenbasis lässt sich in die Schritte *Erkennung von fehlerhaften Datensätzen oder Duplikaten* und *Korrektur der Datensätze* unterteilen. Diese zwei Schritte lassen sich leicht in Regeln der Form

WENN
Eigenschaften der/des Datenfehler(s)
DANN
Korrektur der/des Datenfehler(s)

überführen.

Die an der Universität Singapur entwickelte *IntelliClean*-Applikation [Lup00] benutzt den wissensbasierten Ansatz zur Datenpflege von Datenbasen relationaler

[6] siehe Abschnitt 3.3.6

Datenbanken. Wie die prototypische Implementierung der Diplomarbeit benutzt IntelliClean ebenfalls die Regelmaschine Jess [FH] zur Überwachung der Objekte/Ausführung der Regeln. Die weiterführenden Funktionialitäten von IntelliClean, wie zum Beispiel unscharfe Vergleichsoperationen, können einfach an die Gegebenheiten des enfinity-Systems bzw. BusinessRules-Kompinente adaptiert werden.

Das Beispiel *„Löschung doppelt registrierter Käufer“* in Abschnitt C.1 demonstriert den Einsatz der BusinessRules-Komponente zur Datenpflege. Entgegen des üblichen Datenpflegeansatzes, bei dem im Allgemeinen der Pflegeprozess von Zeit zu Zeit abgearbeitet wird, ist der in diesem Beispiel aufgezeigte Ansatz ein **permanenter** Prozess, d. h. die Aktionen der Datenpflegeregeln werden ausgeführt, **sobald** die entsprechende Vorbedingung erfüllt ist.

5.2.5 Softwaretest

Wie in Abschnitt 2.3.2 beschrieben, sind viele Geschäftsregeln schon während der Entwicklungsphase des E-Commerce-Systems bekannt und müssen in die Systemarchitektur eingehen. Darauf basiert die Idee, die BusinessRules-Komponente nicht nur zur Abwicklung von objekt-/regelbasierten Geschäftsprozessen, sondern schon während der Softwaretestphase - zum Test von Geschäftsprozessen, die mittels des objekt-/prozessorientierten Paradigmas entwickelt wurden - einzusetzen.

Die Einhaltung der Einschränkung des in Abschnitt 2.3 dargestellten Beispieles der Geschäftsregel *„Ein Kunde kann nur Waren in Höhe seines Kreditrahmens bestellen.“*[7] lässt sich durch folgende Regel testen:

WENN
Käufer (id ?kid) (kreditrahmen ?kreditrahmen)
Bestellung (id ?bid) (käufer ?kid)
(bestellSumme ?bestellSumme)
?bestellSumme > ?kreditrahmen
DANN
Fehler-Ausgabe:
`"Für Käufer (id "` ?kid `") existiert eine Bestellung (id "` ?bid `"), deren Bestellwert den Kreditrahmen des Käufers übersteigt."`

Diese Regel wird ausgeführt, sobald ein beliebiger Geschäftsprozess ein die Vorbedingung erfüllendes Bestellungsobjekt erzeugt bzw. ein bestehendes in der Form ändert, dass es die Vorbedingung erfüllt.

[7] das zugehöhrende UML-Diagramm ist in Abbildung 2.9, Seite 24 dargestellt

Der Test durch Regeln ist sehr effizient, da die Testbedingungen entweder schon in Regelform vorliegen oder sehr leicht in entsprechende Regeln transformiert werden können. Die Tests von Bedingungen, die sich auf persistent gehaltene Daten beziehen, gestalten sich im Allgemeinen als recht schwierig. Entweder müssen spezielle Testklassen entworfen, Code-Segmente - die auf diese Daten schreibend zugreifen - geändert oder nach dem Testlauf die Datenbasis auf die Einhaltung der Bedingungen untersucht werden. Der regelbasierte Test erlaubt es, dass Fehler zeitnah (im Gegensatz zum nachträglichen Test der Datenbasis) gemeldet werden, ohne dass dabei spezielle Testklassen entworfen oder Änderungen in den Klassen vorgenommen werden müssten.

Softwaretest ist sicherlich nicht das Hauptanwendungsgebiet der BusinesRules-Komponente und diese wurde auch nicht initial für die Verwendung während des Softwaretests entworfen, trotzdem kann die BusinessRules-Komponente ein sehr hilfreiches und effizientes Mittel in der Testphase sein.

5.3 Fazit

Es gibt eine Reihe von Anwendungsmöglichkeiten, die sich mit der entwickelten BusinessRules-Komponente gut bearbeiten lassen. Nicht zu vergessen ist jedoch, dass der Pattern-Match-Prozess einen relativ hohen Aufwand[8] besitzt.

Es ist notwendig, dass Überlegungen gemacht werden, ob genügend Systemressourcen vorhanden sind, um bestimmte Geschäftsprozesse durch das objekt-/regelbasierte Paradigma realisieren zu können. Die in Abschnitt 5.1 dargestellten Performancetests für elementare Regeln können als Grundlage zur Abschätzung der benötigten Ressourcen dienen.

Werden bestimmte Geschäftsobjekttypen nur im Vorbedingungsteil einer Regel gebraucht und nicht im Aktionsteil manipuliert, so kann oft von einer Überwachung des Geschäftsobjekttyps abgesehen werden. In diesem Fall können dann Objektmuster durch entsprechende Tests ersetzt werden. Folgendes Beispiel demonstriert dieses Vorgehen:

[8] siehe Abschnitt 2.4.4

WENN
... ?productID ...
Product (id ?productID) (price ?price)
?price > 1000
DANN
...

kann ersetzt werden durch

WENN
... ?productID ...
getProductPrice(?productID) > 1000
DANN
...

Zu beachten ist, dass die Funktion **getProductPrice** (liefert den Preis für ein bestimmtes Produkt) zuerst implementiert werden muss. Dies kann durch eine Erweiterung des Jess-Befehlsatzes, durch eine entsprechende Java-Klasse oder durch eine Funktionsdefinition in der Jess-Sprache geschehen[9].

Das Produkt-Geschäftsobjekt ist ein typischer Fall dieser Vorgehensweise, da Produkte im Normalfall sehr statisch sind, d. h. Produkte und deren Beschreibung werden durch Geschäftsprozesse nicht manipuliert, es wird nur lesend auf die Objekte zugegriffen.

Unter anderem werden E-Commerce-Systeme durch ihre Zielgruppe unterschieden. Von **Business-To-Business**-Sites (B2B) spricht man, wenn über das E-Commerce-System Waren und Dienstleistungen an Firmenkunden vertrieben werden. **Business-To-Customer**-Systeme (B2C) sind E-Commerce-Systeme, über die Produkte an Privatpersonen verkauft werden. Im Allgemeinen kann davon ausgegangen werden, dass bei B2B-Systemen durch die sehr eingeschränkte Zielgruppe durchschnittlich weniger Geschäftsprozesse pro Zeit abgearbeitet werden als bei B2C-Systemen. Dafür sind die implementierten Geschäftsprozesse oftmals ungleich komplexer als bei B2C-Systemen, bei denen im Allgemeinen *einfache* Geschäftsprozesse *häufig* abgearbeitet werden.

Aus den oben genannten Eigenschaften und Einschränkung kann impliziert werden, dass die erarbeitete Lösung besser für den Einsatz in B2B-Systemen geeignet ist als für den Einsatz in B2C-Systemen. Gründe dafür sind in der geringeren Häufigkeit der Abarbeitung von Geschäftsprozessen und der daraus resultierenden geringeren Änderungshäufigkeit der Geschäftsobjekte zu finden. Durch die geringere Änderungshäufigkeit der Geschäftsobjekte kann davon ausgegangen

[9] siehe Abschnitt 4.1.2

werden, dass im Allgemeinen genügend Systemressourcen zur Verfügung stehen, um die von den Änderungen ausgelösten Geschäftsregeln zeitnah abarbeiten zu können. Ein weiterer Grund sind die komplexeren Geschäftsprozesse, diese lassen sich oftmals sehr gut durch das objekt-/regelbasierte Paradigma realisieren.

Kapitel 6

Zusammenfassung und Ausblick

6.1 Zusammenfassung

Moderne betriebliche Informationssysteme bilden immer mehr Geschäftsprozesse innerhalb von Unternehmen ab. Gleichzeitig müssen Unternehmen ihre Prozesse flexibel an die sich zum Teil sehr stark verändernde Marktsituation anpassen. Dies hat zur Folge, dass auch die Systeme, die diese Geschäftsprozesse abbilden, immer häufiger Änderungen unterliegen.

Besonders große Auswirkungen hat diese zunehmende Flexibilisierung auf *E-Commerce-Systeme*, betriebliche Informationssysteme für den Vertrieb von Produkten über das Internet, da diese im Allgemeinen Geschäftsprozesse implementieren, die besonders häufig Änderungen unterliegen.

Ein Ansatz, dieser zunehmenden Veränderungen Herr zu werden, ist die Modellierung von Unternehmungen durch Regelwerke, die relativ einfach an die aktuelle Situation angepasst werden können. Diese Regelwerke werden von Experten der einzelnen Unternehmensbereiche, wie z. B. Marketing oder Vertrieb, erstellt und modifiziert. Die Business Rules Group erarbeitet grundlegende Konzepte [Bus00a], die sich dieser Art der Unternehmensmodellierung und deren Umsetzung durch Informationssysteme widmen.

In der Informatik sind wissensbasierte Systeme bekannt, die auf Regeln der Form „**WENN** *Vorbedingung erfüllt*, **DANN** *löse Aktion aus*“ beruhen. Die *Wissensbasis* dieser auch *Regelmaschinen* genannten Systeme besteht zum einen aus dem *Expertenwissen*, das durch die Regeln gegeben ist und dem *Faktenwissen*, das die aktuelle Situation repräsentiert. Durch Regelanwendung (*Inferenz*) können Fakten manipuliert, gelöscht oder neu erzeugt werden. Eine Regel ist anwendbar, sobald es Fakten gibt, die die Vorbedingung der Regel erfüllen. Sind mehrere Regeln anwendbar, entsteht eine Konfliktmenge, aus der mittels einer Konfliktlösungs-

strategie die Regel ausgesucht wird, die als nächstes zur Anwendung kommt.

Die vorliegende Arbeit zeigt auf, wie ein wissensbasiertes System in ein bestehendes E-Commerce-System zur regelbasierten Manipulation von Geschäftsobjekten integriert werden kann. Zum Nachweis der Umsetzbarkeit der erarbeiteten Konzeption und Systemarchitekur wurde eine prototypische Implementierung erstellt. Das auf der *Java 2 Plattform Enterprise Edition* (J2EE) [Sun00b, Sun00a] basierende E-Commerce-System *enfinity* [INTi, INTf, INTh, INTk] von INTERSHOP Communications liegt dieser Realisierung zugrunde. Als Regelmaschine, die im Zuge der Diplomarbeit zur Auswertung der Regeln in enfinity-Plattform integriert wurde, dient die *Java Expert System Shell* (Jess) [FH], die von Ernest J. Friedman-Hill an den Sandia National Laboratories, Abteilung Distributed Computing Systems, entwickelt wurde.

Die in dieser Diplomarbeit erarbeitete Konzeption sieht vor, dass sämtliche im enfinity-System vorhandenen Geschäftsobjekte (wie zum Beispiel Käufer, Adresse, Produkt, Bestellung) regelbasiert manipuliert werden können. Geschäftsobjekte sind innerhalb des enfinity-Systems als Enterprise JavaBeans (EJBs) [Sun99a, Sun00a, Sun98, DP00, Rom99] realisiert. INTERSHOP enfinity stellt das objekt-/prozessorientierte Paradigma zur Abbildung der Geschäftsprozesse zur Verfügung. Durch die erarbeitete Lösung wird ein zusätzliches Paradigma, das *objekt-/regelbasierte Paradigma*, zur Implementierung der Geschäftsprozesse zur Verfügung gestellt.

Die aus der Konzeption resultierende Systemarchitektur sieht vor, dass neben den beiden enfinity Servern *enfinity Transactivity Server* (eTS) und *enfinity Catalog Server* (eCS) ein dritter, der neu geschaffene *BusinessRules-Server*, die enfinity-Plattform bilden. Der BusinessRules-Server beinhaltet die Regelmaschine, die die Geschäftsobjekte, die sich auf dem eCS bzw. eTS befinden, regelbasiert manipuliert.

Die Kommunikation zwischen BusinessRules-Server und dem eTS bzw. eCS geschieht mittels der Remote Method Invocation (RMI) [Sun99d]. Als Schnittstellenobjekt dient auf Seiten des BusinessRules-Servers das *BridgeManager-Objekt*, als Kommunikationsendpunkt in den enfinity Servern wird das *Dispatcher-Objekt* im eTS und eCS installiert.

Für jedes Geschäftsobjekt wird das *Proxy-* [GHJV96, S. 254] und *Adapter-Muster* [GHJV96, S. 171] angewandt und ein Proxyobjekt im BusinessRules-Server erzeugt, das der Faktenbasis der Regelmaschine hinzugefügt wird. Das Proxyobjekt speichert ausschließlich Informationen zum Finden des Orginal-Geschäftsobjektes, das sich entweder auf dem eTS oder auf dem eCS befindet. Wird auf Attribute

des Proxyobjektes von der Regelmaschine aus lesend oder schreibend zugegriffen, so werden die Methodenaufrufe an das Orginal-Geschäftsobjekt weitergeleitet.

Die Proxyobjektklassen können automatisch generiert werden. Es ist ebenfalls möglich, dass in der Geschäftsobjektklasse vorhandene Methoden für den Zugriff auf bestimmte Attribute nicht in der Proxyobjektklasse generiert werden. Das ist dann sinnvoll, wenn Attribute vorhanden sind, die nicht zum eigentlichen Modell des Geschäftsobjektes gehören, sondern zum Beispiel durch den Datenhaltungsmechanismus notwendig geworden sind. Werden Methoden für den Zugriff auf Attribute in einer Proxyobjektklasse nicht generiert, so ist das Attribut für die Regelmaschine auch nicht sichtbar. Die Vererbungshierarchie bleibt durch die Anwendung des Proxy- und Adapter-Musters voll erhalten und innerhalb von Regeln anwendbar.

Wird ein Geschäftsobjekt erzeugt, geändert oder gelöscht, so muss die Regelmaschine über den jeweiligen Vorgang benachrichtigt werden. Im Falle einer Erzeugung muss innerhalb des BusinessRules-Servers ein Proxyobjekt für das Geschäftsobjekt erzeugt und der Faktenbasis der Regelmaschine hinzugefügt werden. Wird ein Geschäftsobjekt gelöscht, dann muss das Proxyobjekt aus der Faktenbasis der Regelmaschine gelöscht und das Proxyobjekt zerstört werden. Im Falle einer Geschäftsobjektänderung ist nur eine Benachrichtigung der Regelmaschine über die Änderung notwendig.

Synchrone Methodenaufrufe über Systemgrenzen hinweg, bergen das Risiko von Verklemmungen in sich. Um diese Verklemmungsgefahr zu beseitigen wurden die entsprechenden Methoden asynchronisiert und die daraus folgenden Implikationen umgesetzt.

Da nicht alle Fakten in den Geschäftsobjekten gespeichert werden können, erfordert die sinnvolle Anwendung von Regelwerken die persistente Speicherung von Fakten. Im Rahmen dieser Diplomarbeit wurde deshalb die *Fact*-Enterprise JavaBean realisiert. Mittels dieser Fact-EJB können unter einem Zeichenkettenschlüssel beliebige Zeichenketten und Zahlenwerte gespeichert werden. Die Fact-EJB kann ebenfalls dazu benutzt werden, um an ein bestehendes Geschäftsobjekt eine Schlüssel-Wert-Kombination „anzuhängen“.

Innerhalb des enfinity-Systems werden Geschäftsprozesse, die durch das objekt-/prozessorientierte Paradigma implementiert wurden, in Form von *Pipelines*, die der Pipeline Prozessor abarbeitet, realisiert. Jede Pipelineabarbeitung findet in einem Sitzungskontext statt. Um Daten zwischen einer Sitzung bzw. Piplineausführung und der Regelmaschine austauschen zu können wurden *SessionFact*-Objekte eingeführt. Zur Synchronisation zwischen Pipelineausführung und Regelmaschine/-auswertung wurden Mechanismen entwickelt, die sicherstellen, dass

an dem Synchronisationspunkt der Pipeline alle ausführbaren Regeln ausgeführt wurden. Die Kombination aus SessionFact-Objekten und Synchronisation erlaubt die Implementierung von rein regelbasierten Webanwendungen.

Die Jess-Sprache wurde erweitert, um einfach Geschäftsobjekte ändern und löschen zu können. Es wurde sichergestellt, dass Transaktionen im Aktionsteil einer Regel verfügbar sind und dass neue Geschäftsobjekte erzeugt werden können.

Die prototypische Implementierung wurde einer Leistungsanalyse unterzogen. Zum einen wurde die Performance der Realisierung anhand von elementaren aber nicht untypischen Testfällen untersucht. Zum anderen wurden Anwendungsgebiete und -möglichkeiten, die mit der entwickelten Lösung gut zu bearbeiten sind, vorgestellt. Um die Funktionialität zu demonstrieren, wurden für die meisten Anwendungsgebiete entsprechende Beispiele realisiert.

Auf konzeptioneller Ebene wurde eine benutzerfreundliche Symbolsprache zum Anzeigen und Ändern von Regeln entworfen. Konzeptionelle Erweiterungen und die notwendigen Schritte, die die entwickelte Lösung in der Form erweitern, dass Experten, zum Beispiel aus Bereich Marketing, direkt und benutzerfreundlich Regeln und damit auch Geschäftsprozesse ihrer Domäne erstellen und modifizieren können, wurden entworfen.

Zusammenfassend kann festgestellt werden, dass die in dieser Diplomarbeit erarbeitete Lösung die in der Aufgabenstellung definierten Ziele erfüllt. Die vorgestellte Komponente erlaubt die regelbasierte Manipulation von Geschäftsobjekten in E-Commerce-Systemen. Dabei wurde großer Wert darauf gelegt, dass nicht nur die zum Standardumfang gehörenden Geschäftsobjekte regelbasiert manipuliert werden können, sondern auch die Geschäftsobjekte, die zukünftig in das System integriert werden. Die Integration der Komponente erfolgt durch eine minimale Änderung (Änderung einer Klasse) der Klassen, die das E-Commerce-System bilden. Ein Austausch der Regelmaschine, die in der prototypischen Implementierung zum Einsatz kommt, ist leicht möglich.

Das in das E-Commerce-System neu hinzugefügte objekt-/regelbasierte Paradigma hat gegenüber des vorhandenen objekt-/prozessorientierten Paradigmas den Nachteil, dass es verhältnismäßig viele Systemressourcen benötigt. Das liegt zum einen an der komplexen Natur des Mustererfüllungsproblems, das mit dem Rete-Algorithmus gelöst wird und zum anderen an den entfernten Methodenaufrufen, die durch die verteilte Struktur des enfinity-Systems bedingt sind. Trotz dieses Nachteils kann festgestellt werden, dass die vorgestellte Lösung sehr wohl sinnvoll eingesetzt werden kann.

Mittels der erarbeiteten Lösung können nicht nur Geschäftsprozesse mit hoher Änderungshäufigkeit und Personalisierungprozesse in Form von Regelwerken erstellt und manipuliert werden, es wurden zusätzliche Anwendungsmöglichkeiten identifiziert. Dies sind Beratungssysteme, Anwendungen zur Datenpflege und Anwendungen zum Test der Software während des Softwareentwicklungsprozesses.

Zur Darstellung und Änderung der Regeln wurde eine Symbolsprache entwickelt, dabei wurde auf eine einfache Struktur, Übersichtlichkeit und Einheitlichkeit geachtet, so dass sie durch den Benutzer schnell erlernt und fehlerfrei angewandt werden kann.

6.2 Ausblick

Es wurden über die *„Integration eines Aktions- und Regelprozessors“* hinausgehende Vorgehensweisen und Mechanismen entwickelt. Diese Teile der Diplomarbeit, die als konzeptioneller Entwurf vorliegen, müssen implementiert und einem Test unterzogen werden. Das sind zum Großteil die Technologien, die Phase II von Phase I unterscheiden[1]. Das sind zum einen die verschiedenen Sichten für die entsprechenden Anwendungsgebiete/Domänen und die Benutzeroberfläche zur Anzeige und Eingabe der Regeln durch die vorgestellte Symbolsprache.

Weitere Erweiterungs- und Optimierungsmöglichkeiten wurden angedacht aber noch keiner genaueren Überprüfung unterzogen. Das sind:

- Die automatische Generierung optimaler Proxyklassen, die den von den Java ServerPages [Sun99g] bekannten Mechanismus zur dynamischen Kompilierung und Integration der Klassen nutzt, um vom aktuellen Regelwerk abhängige Proxyklassen zu erzeugen und in das System zu integrieren. Der Kern der Idee ist die Untersuchung des aktuellen Regelwerks auf die verwendeten Geschäftsobjektklassen und deren Attribute sowie die Generierung von Proxyklassen, die ausschließlich die Methoden zur Manipulation von Attributen enthält, die im Regelwerk Verwendung finden.

- Die Versendung von Benachrichtigungen[2], die die geänderten Attributwerte enthalten. Diese Lösung wäre aber vom verwendeten Applicationserver abhängig, da mit einer spezifikationskonformen Realisierung dies nicht möglich ist.

- Die Nutzung des Java Messaging Service (JMS) [Sun02b] zum Nachrichtenaustauch[3] zwischen BusinessRules-Server und den enfinity-Servern. Dafür

[1] siehe Abschnitt 3.2
[2] siehe Abschnitt 3.3.3
[3] siehe Abschnitt 3.3.1

besteht kein akuter Handlungsbedarf, falls jedoch in Zukunft auch Geschäftsobjekte von anderen Systemen, wie zum Beispiel Lagerhaltungssystemen in die Faktenbasis eingebunden werden, könnte es sinnvoll sein, den Nachrichtenaustausch über den standardisierten Java Message Service abzuwickeln.

Im Bereich der Java-basierten betrieblichen Informationssysteme und Regelmaschinen findet eine ständige, teils hochdynamische Weiterentwicklung statt. Eine Reihe von Produkten und Spezifikationen befinden sich im Spezifikationsprozess oder wurden vor kurzer Zeit freigegeben und in der Zwischenzeit noch nicht in das enfinity-System integriert. Folgende Produkte bzw. Spezifikationen berühren direkt die in der Diplomarbeit entwickelte Lösung:

Java 2 Plattform, Standard Edition, Version 1.4 [Sun02a]: Die neue Java 2 Plattform, Version 1.4, bringt einige Neuerungen mit sich, die auch in der nächsten Version der Java 2 Plattform, Enterprise Edition, integriert sein werden. Vor allem der Reflection-Mechanismus, der an vielen Stellen der entwickelten Lösung und auch beim entfernten Methodenaufruf (RMI) Verwendung findet, soll mehrmals [Sun02a] bzw. dreimal [Sch01] schneller arbeiten als dies bei den Vorgängerversionen der Java Plattform der Fall war.

Enterprise JavaBeans Specification, Version 2.0 [Sun01]: Die Spezifikation 2.0 der Enterprise JavaBeans stellt eine komplette Überarbeitung der EJB-Architektur dar und beinhaltet sehr viele Erweiterungen [Sun]. Sollte das neue Komponentenmodell in den zukünftigen enfinity-Versionen verwendet werden, muss überprüft werden, ob die Erweiterungen vorteilhaft für die BusinessRules-Komponente nutzbar sind.

JSR 94: Java Rule Engine API [Sel02]: Mittels des Java Community Process' werden zukünftige Spezifikationen erarbeitet. Die Expertengruppe des Java Specification Request 94 hat sich zum Ziel gesetzt, eine einheitliche Schnittstelle für Regelmaschinen zu spezifizieren. Folgende Firmen stellen Experten zu diesem Zweck: Art Technology Group Inc. (ATG), BEA Systems, Blaze Software, Fujitsu Limited, IBM, ILOG, Oracle, SilverStream Software und Unisys.

Eine einheitliche Schnittstelle zur Ansteuerung von Regelmaschinen würde viele Vorteile für die vorgestellte Lösung bringen. Der größte Vorteil wäre sicherlich die Austauschbarkeit der Regelmaschine. Die für diese Aufgabe am geeigneteste könnte integriert werden, ohne dass zum Beispiel Änderungen im Mechanismus zur Benachrichtigung der Regelmaschine gemacht werden müssten.

Es bleibt abzuwarten, ob, wie schnell und in welcher Form sich regelbasierte Ansätze für die Modellierung von Unternehmen und Geschäftsprozessen sowie deren Umsetzung innerhalb von Informationssystemen durchsetzen werden. Nicht zuletzt die Zusammensetzung der an der Java Rule Engine API-Spezifizierung beteiligten Firmen, unter denen auch viele Hersteller von betrieblichen Informationssystemen und E-Commerce-Systemen sind, zeigt, dass das Interesse für die Benutzung von regelbasierten Mechanismen in Informationssystemen hoch ist.

Anhang A

Evaluierung verschiedener Regelmaschinen

In der Startphase der Diplomarbeit wurden verschiedene Regelmaschinen (*engl.: rule engines*) getestet. Bei der Evaluierung wurden vor allem die Kriterien *Stabilität*, *Entwicklungsfähigkeit*, *Javaintegration* und *Dokumentation* betrachtet. Leider stellten nicht alle kontaktierten Firmen oder Organisationen in dem vorgegebenen Zeitplan eine Testversion ihrer Regelmaschine zur Verfügung, zu diesen gehören Brokat (ehemals BlazeSoft) mit dem Produkt Advisor und Haley Enterprise mit „Rules for Java“. In den folgenden Abschnitten werden die getesteten Regelmaschinen vorgestellt. Im Abschnitt A.4 werden die verschiedenen Regelmaschinen einem Leistungstest unterzogen.

A.1 IBM CommonRules

CommonRules wurde im Rahmen des Forschungsprojektes „Extended Enterprise coalition for integrated COllaborative Manufacturing Systems“ (EECOMS) des National Institute of Standards and Technology im Rahmen des Advanced Technology Programms von IBM entwickelt und ist zur Evaluierung frei verfügbar.

CommonRules wurde speziell für den betrieblichen und betriebsübergreifenden Einsatz konzipiert. Dabei wurde vor allem auf die Verarbeitbarkeit von verschiedenen Regelformaten wie BRML (Business Rules Meta Language) [Cov01], KIF (Knowledge Interchange Format) [Gen], XSB Dateien [SSW+00, WR00], SModels [Sim00] sowie die in diesem Projekt entstandene CLPs (Courteous Logic Programs) [Gro99b, Gro99a] geachtet.

Die Evaluierung der Version 2.1 der in Java programmierten CommonRules Software ergab, dass die Dokumentation nur sehr mangelhaft ist und die Stabilität nicht den notwendigen Anforderungen entspricht.

A.2 ILOG JRules

ILOG JRules [ILO01b] lag zur Evaluation in der Version 3.1 vor. JRules ist eine in Java programmierte Komponente, deren Kern den Rete-Algorithmus[1] implementiert. Standard-Javaobjekte können der JRules Komponente als Faktenwissen dienen.

Hervorzuheben ist, dass neben einer java-ähnlichen Regeldefinitionssprache eine „Business Action Language“ existiert, mit der Regeln in fast natürlicher Sprache definiert werden können. Dazu werden Objekteigenschaften Satzteile und die dazugehörenden Parameter zugeordnet, mittels dieser Satzteile werden die Regeln definiert, die dann wiederum durch die Zuordnung in die java-ähnliche Sprache übersetzt werden können.

Neben der Regelmaschine besitzt JRules eine „Builder“ genannte graphische Oberfläche zur Definition der Regeln und zur Zuordnung der Satzteile für die „Business Action Language“.

A.3 Java Expert System Shell (Jess)

Jess [FH, FH01] wurde von Ernest Friedman-Hill am Sandia National Laboratories in Livermore entwickelt. Als Grundlage für Jess diente das C Language Integrated Production System (CLIPS) [Ver, Gia], das ursprünglich am Johnson Space Center der NASA entwickelt wurde.

Die Syntax und die Semantik der CLIPS-Sprache wurde in Jess übernommen. Zusätzlich zu der Funktionalität von CLIPS bietet Jess die Möglichkeit, Java-Objekte und -Beans in das System einzubinden und Methoden von diesen aufzurufen. Die Regelmaschine basiert ebenfalls auf dem in Abschnitt 2.4.4 vorgestellten Rete-Algorithmus.

Eine Besonderheit von Jess ist, dass interpretierte Skripts ausgeführt werden können, die direkt auf Javaobjekte zugreifen und diese manipulieren können. Jess ist die einzige Regelmaschine im Testfeld, bei der der Sourcecode verfügbar ist.

Jess wurde schon in einer Vielzahl von Projekten erfolgreich eingesetzt. Die Bandbreite der Anwendungen reicht von Moleküldesign in der pharmazeutischen Forschung über Helpdesksysteme bis zu Agentensystemen für das Internet.

[1] siehe Abschnitt 2.4.4

A.4 Leistungstest

Neben der Untersuchung, die den Umfang der Funktionalitäten der Regelmaschinen betrachtet, wurde auch eine Leistungsanalyse erstellt. Als Testszenario wurde das Sortierproblem ausgewählt. Es sollen 100 Javaobjekte anhand eines Attributes sortiert werden. Um dies zu erreichen sind zwei Regeln notwendig[2]. Für die Sortierung der Javaobjekte sind ca. 4900 Regelanwendungen notwendig.

Aufgrund der oben beschriebenen Mängel von IBM CommonRules wurde diese Regelmaschine nicht dem Leistungstest unterzogen.

Das Ergebnis des Tests wird in Abschnitt A.4.3 vorgestellt.

A.4.1 Jess Regeldefinitionen

```
1  ;;
2  ;; Java-Klasse Element als Template in JESS einfuegen
3  ;;
4  (defclass element Element)

6  ;;
7  ;; Regel: Erhoehe die position eines Elementes
8  ;;
9  ;; WENN
10 ;; value von Element ?e2 groesser ist als die von Element ?
       e1 und position von Element ?e2 und Element ?e1 gleich
       sind
11 ;; DANN
12 ;; Element ?e2 die position von Element ?e1 + 1 zuweisen
13 ;;
14 (defrule incrementPosition
15   ?e1 <- (element (value ?v1) (position ?p1))
16   ?e2 <- (element (value ?v2&:(> ?v2 ?v1)) (position ?p2
         &:(= ?p2 ?p1)))
17  =>
18   (modify ?e2 (position (+ ?p1 1)))
19 )

21 ;;
22 ;; Regel: Vertausche die position von 2 Elementen
23 ;;
24 ;; WENN
```

[2]siehe Abschnitt A.4.1 und A.4.2

```
25 ;; value von Element ?second groesser ist als von Element ?
     first und position von Element ?second kleiner als von
     Element ?first
26 ;; DANN
27 ;; Element ?first die position von Element ?second zuweisen
      und Element ?second die position von Element ?first
     zuweisen
28 ;;
29 (defrule switchPosition
30   ?e1 <- (element (value ?v1) (position ?p1))
31   ?e2 <- (element (value ?v2&:(> ?v2 ?v1)) (position ?p2
        &:(< ?p2 ?p1)))
32  =>
33   (modify ?e1 (position ?p2))
34   (modify ?e2 (position ?p1))
35 )

37 ;;
38 ;; Regel: Zeige benoetigte Zeit an
39 ;;
40 ;; WENN
41 ;; es Fakt (startsec ?start) gibt und es Fakt (stopsec ?
     stop) gibt
42 ;; DANN
43 ;; gebe ?stop - ?start aus
44 ;;
45 (defrule showtime
46   (startsec ?start)
47   (stopsec ?stop)
48  =>
49   (printout t "=> TIME needed (ms): " (- (float ?stop) (
        float ?start)) crlf)
50 )

52 ;;
53 ;; Erzeuge 100 Elemente
54 ;;
55 (bind ?i 1)
56 (while (< ?i 101) do
57   (bind ?e (new Element ?i) )
58   (definstance element ?e)
59   (bind ?i (+ ?i 1) )
60 )

62 ;; Erzeuge (startsec <aktuelle Systemzeit>)-Fakt
```

```
(assert (startsec (call java.lang.System currentTimeMillis)
   ))
;; Starte Regelauswertung (Sortierung)
(run)
;; Erzeuge (stopsec <aktuelle Systemzeit>)-Fakt
(assert (stopsec (call java.lang.System currentTimeMillis))
   )
;; Starte Regelauswertung (Zeitausgabe)
(run)
```

A.4.2 ILOG JRules Regeldefinitionen

```
import sort.*;

/**
 * Initialisierung: Erzeuge 100 unsortierte Elemente
 */
setup
{
  assert Control(Control.InitTask) {}
  assert Element(1);
  assert Element(2);
  assert Element(3);
...
  assert Element(98);
  assert Element(99);
  assert Element(100);
};

/**
 * Regel: Starte Sortierregelauswertung
 */
rule startSort
{
  when
  {
    ?c: Control(id==InitTask);
  }
  then
  {
    modify ?c { id=SortTask; }
  }
};

/**
```

```
127  * Regel: Vertausche die position von 2 Elementen
128  *
129  * WENN
130  * value von Element ?second groesser ist als von Element ?
        first und position von Element ?second kleiner als von
        Element ?first
131  * DANN
132  * Element ?first die position von Element ?second zuweisen
         und Element ?second die position von Element ?first
        zuweisen
133  */
134 rule switchPosition
135 {
136   when
137   {
138     Control(id==SortTask);
139     ?first: Element(?p1:position);
140     ?second: Element(value>?first.value; ?p2:position &< ?
           p1);
141   }
142   then
143   {
144     modify ?first { position = ?p2; }
145     modify ?second { position = ?p1; }
146   }
147 };

149 /**
150  * Regel: Erhoehe die position eines Elementes
151  *
152  * WENN
153  * value von Element ?e2 groesser ist als die von Element ?
        e1 und position von Element ?e2 und Element ?e1 gleich
        sind
154  * DANN
155  * Element ?e2 die position von Element ?e1 + 1 zuweisen
156  */
157 rule incrementPosition
158 {
159   when
160   {
161     ?c: Control(id==SortTask);
162     ?e1: Element();
163     ?e2: Element(value>?e1.value; position==?e1.position);
164   }
```

```
165   then
166   {
167     modify ?e2 { position = ?e1.position + 1; }
168   }
169 };
```

A.4.3 Testergebnis

Der Leistungstest wurde zehn mal pro Produkt ausgeführt. Die Tabelle A.1 zeigt die minimale, maximale und mittlere Ausführungszeit für diesen Test. Die prozentuale Angabe bezieht sich auf die mittlere Ausführungszeit.

Produkt	Min. Zeit	Max. Zeit	Mittlere Zeit	Prozent
Jess	2,832 Sek.	2,875 Sek.	2,853 Sek.	100 %
ILOG JRules	4,475 Sek.	4,916 Sek.	4,616 Sek.	161,8 %

Tabelle A.1: Ergebnis des Leistungstests

A.5 Zusammenfassung

Die aus der Evaluierung und dem Leistungstest gewonnenen Erkenntnisse legen zu diesem Zeitpunkt die Verwendung von Jess als Regelmaschine nahe.

IBM CommonRules eignet sich sehr gut für die Zusammenführung und den Austausch von Regelwerken. Die Mängel sind vor allem in der Dokumentation und der Stabilität zu sehen.

ILOG JRules bietet eine komplette Entwicklungsumgebung sowie die „fast“ natürlichsprachige Business Action Language mit der auch Nicht-Informatiker leistungsfähige Regelwerke entwerfen können. Als Minuspunkt ist die Geschwindigkeit der Implementierung des Rete-Algorithmus, die nur ca. 60 % der Leistungsfähigkeit der Regelmaschine von Jess erreicht, zu werten.

Jess ist die schnellste Regelmaschine im Testfeld. Ein weiterer Pluspunkt ist sicherlich die Möglichkeit, innerhalb von Jess interpretierte Skripts ausführen zu können. Im Gegensatz zu ILOG JRules wird sie aber ohne graphische Entwicklungsumgebung ausgeliefert.

Anhang B

Klassen

Die folgenden Hierarchien zeigen die Einordnung der im Rahmen der Diplomarbeit entstandenen Klassen und Interfaces. Abgeleitete Klassen werden unter ihren Superklassen eingerückt dargestellt. Im enfinity-System vorhandene Klassen bzw. Interfaces sind normal gedruckt, die zur Diplomarbeit gehörenden Klassen sind fettgedruckt.

B.1 Klassenhierarchie

- class java.lang.Object
  - class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**AbstractCommand** (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
    - class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**AttributeChanged** (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
    - class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Commit** (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
    - class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Commited** (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
    - class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Create** (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
    - class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Created** (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)

- class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Remove**
  (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
- class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Removed**
  (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
- class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**SetAttribute**
  (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
- class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Synchronize**
  (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
- class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Synchronized**
  (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.Command*)
- class com.intershop.enfinity.cartridges.businessrules.ejb.**AbstractFactKey**
  (implementiert *java.io.Serializable*)
  - class com.intershop.enfinity.cartridges.businessrules.ejb.**FactKey**
  - class com.intershop.enfinity.cartridges.businessrules.ejb.**SessionFactKey**
- class com.intershop.enfinity.cartridges.businessrules.ejb.**AbstractFactState**
  (implementiert *com.persistence.container.EntityState*)
  - class com.intershop.enfinity.cartridges.businessrules.ejb.**FactState**
  - class com.intershop.enfinity.cartridges.businessrules.ejb.**SessionFactState**
- class com.intershop.beehive.core.cartridge.Cartridge
  (implementiert *com.intershop.beehive.core.cartridge.ICartridge*)
  - class com.intershop.enfinity.cartridges.businessrules.**BusinessRulesCartridge**
- class com.intershop.enfinity.cartridges.businessrules.jess.date.**Clock**
  (implementiert *java.lang.Runnable*)
- class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**CommandQueue**
- class com.intershop.enfinity.cartridges.businessrules.jess.**Commit**
  (implementiert *jess.Userfunction*)
- class com.intershop.enfinity.cartridges.businessrules.bridge.generate.**DirectoryTreeGenerator**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**EJBObjectProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**EntityProxy**
- class com.intershop.enfinity.cartridges.businessrules.jess.**EJBRetract**
  (implementiert *jess.Userfunction*)
- class com.persistence.container.internal.EntityBeanImpl
  (implementiert *javax.ejb.EntityBean*)
  - class com.intershop.enfinity.cartridges.businessrules.ejb.**AbstractFactBean**
    - class com.intershop.enfinity.cartridges.businessrules.ejb.**FactBean**
    - class com.intershop.enfinity.cartridges.businessrules.ejb.**SessionFactBean**
  - class com.intershop.beehive.core.common.FWPersistentObjectBean
    - class com.intershop.enfinity.cartridges.businessrules.ejb.demo.**AnimalNodeBean**

- class com.intershop.beehive.core.common.FWPersistentObjectKey
  (implementiert *java.io.Serializable*)
  - class com.intershop.enfinity.cartridges.businessrules.ejb.demo.**AnimalNodeKey**
- class com.intershop.beehive.core.common.FWPersistentObjectState
  (implementiert *com.persistence.container.EntityState*)
  - class com.intershop.enfinity.cartridges.businessrules.ejb.demo.**AnimalNodeState**
- class com.intershop.enfinity.cartridges.businessrules.jess.**ISAssert**
  (implementiert *jess.Userfunction*)
- class com.intershop.enfinity.cartridges.businessrules.jess.**ISAssertSession**
  (implementiert *jess.Userfunction*)
- class com.intershop.enfinity.cartridges.businessrules.bridge.generate.**JarTreeGenerator**
- class com.intershop.enfinity.cartridges.businessrules.bridge.generate.**Leaf**
  (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.generate.TreeElement*)
  - class com.intershop.enfinity.cartridges.businessrules.bridge.generate.**Node**
    (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.generate.TreeElement*)
- class com.intershop.enfinity.cartridges.businessrules.bridge.generate.**Node.TreeElementComparator**
  (implementiert *java.util.Comparator*)
- class com.intershop.beehive.core.pipeline.Pipelet
  - class com.intershop.enfinity.cartridges.businessrules.pipelet.**DetermineSessionID**
  - class com.intershop.enfinity.cartridges.businessrules.pipelet.**ExecuteReteCommand**
  - class com.intershop.enfinity.cartridges.businessrules.pipelet.**GetFact**
  - class com.intershop.enfinity.cartridges.businessrules.pipelet.**GetSessionFact**
  - class com.intershop.enfinity.cartridges.businessrules.pipelet.**PutFact**
  - class com.intershop.enfinity.cartridges.businessrules.pipelet.**PutSessionFact**
  - class com.intershop.enfinity.cartridges.businessrules.pipelet.**SynchronizeWithRete**
- class com.intershop.beehive.tools.dbinit.Preparer
  (implementiert *com.intershop.beehive.tools.dbinit.IPreparable*)
  - class com.intershop.enfinity.cartridges.businessrules.init.**AnimalNodePreparer**
  - class com.intershop.enfinity.cartridges.businessrules.init.**BusinessRulesPreparer**
- class com.intershop.enfinity.cartridges.businessrules.bridge.**PropertyDescriptions**
- class com.intershop.enfinity.cartridges.businessrules.bridge.**ProxyDescription**
- class com.intershop.enfinity.cartridges.businessrules.bridge.generate.**ProxyGenerator**
- class java.rmi.server.RemoteObject
  (implementiert *java.rmi.Remote, java.io.Serializable*)
  - class java.rmi.server.RemoteServer
    - class java.rmi.server.UnicastRemoteObject
      - class com.intershop.enfinity.cartridges.businessrules.bridge.**BridgeManagerImpl**
        (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.BridgeManager*)

    - class com.intershop.enfinity.cartridges.businessrules.bridge.**DispatcherImpl**
      (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.Dispatcher, com.intershop.enfinity.cartridges.businessrules.observable.IEJBObserver*)
    - class com.intershop.enfinity.cartridges.businessrules.bridge.synchronize.**MonitorImpl**
      (implementiert *com.intershop.enfinity.cartridges.businessrules.bridge.synchronize.Monitor*)
- class com.intershop.enfinity.cartridges.businessrules.bridge.**RMIHelper**
- class com.intershop.beehive.core.service.Service
  (implementiert *com.intershop.beehive.core.service.IService*)
  - class com.intershop.enfinity.cartridges.businessrules.core.service.**BusinessRulesService**
    (implementiert *com.intershop.enfinity.cartridges.businessrules.core.service.IBusinessRulesService*)
- class com.intershop.enfinity.cartridges.businessrules.core.**SessionFactRemover**
  (implementiert *com.intershop.beehive.core.session.ISessionObserver*)
- class java.lang.Thread
  (implementiert *java.lang.Runnable*)
  - class com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Processor**
  - class com.intershop.enfinity.cartridges.businessrules.jess.**ReteThread**
- class com.intershop.enfinity.cartridges.businessrules.jess.**Userfunctions**
  (implementiert *jess.Userpackage*)

B.2 Interfacehierarchie

- interface *com.intershop.enfinity.cartridges.businessrules.bridge.commandqueue.**Command***
- interface *com.intershop.enfinity.cartridges.businessrules.**Constants***
- interface *com.intershop.enfinity.cartridges.businessrules.observable.**IEJBObserver***
- interface *com.intershop.beehive.core.service.IService*
  - interface *com.intershop.enfinity.cartridges.businessrules.core.service.**IBusinessRulesService***
- interface *java.rmi.Remote*
  - interface *com.intershop.enfinity.cartridges.businessrules.bridge.**BridgeManager***
  - interface *com.intershop.enfinity.cartridges.businessrules.bridge.**Dispatcher***
  - interface *javax.ejb.EJBHome*
    - interface *com.persistence.container.EntityHome*
      - interface *com.intershop.enfinity.cartridges.businessrules.ejb.**AbstractFactHome***
      - interface *com.intershop.enfinity.cartridges.businessrules.ejb.demo.**AnimalNodeHome***
      - interface *com.intershop.enfinity.cartridges.businessrules.ejb.**FactHome***
      - interface *com.intershop.enfinity.cartridges.businessrules.ejb.**SessionFactHome***

- interface *javax.ejb.EJBObject*
  - interface *com.persistence.container.EntityObject*
    - interface *com.intershop.enfinity.cartridges.businessrules.ejb.**AbstractFact***
      - interface *com.intershop.enfinity.cartridges.businessrules.ejb.**Fact***
      - interface *com.intershop.enfinity.cartridges.businessrules.ejb.**SessionFact***
    - interface *com.intershop.beehive.core.common.FWPersistentObject*
      - interface *com.intershop.enfinity.cartridges.businessrules.ejb.demo.**AnimalNode***
  - interface *com.intershop.enfinity.cartridges.businessrules.bridge.synchronize.**Monitor***
- interface *com.intershop.enfinity.cartridges.businessrules.bridge.generate.**TreeElement***

B.3 Proxyhierarchie

Diese Hierarchie zeigt die von dem Proxygenerator generierten Klassen. Als Grundlage der Generierung dienten die Enterprise JavaBeans, die zum Standardumfang von enfinity gehören und die EJBs, die im Rahmen der Diplomarbeit erstellt wurden.

- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**EntityProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**AbstractFactProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**FactProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**SessionFactProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**AlertListProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**AttributeValueProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BasicSeriesEntryProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BundleAssignmentProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**CartridgeInformationProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**CatalogCategoryLinkProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**CurrencyProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**DomainInformationProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**FWLocaleMappingProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**FWLocaleProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**FWPersistentObjectProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**AlertConditionProxy**
      - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**PriceAlertConditionProxy**

- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**StockAlertConditionProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**AlertProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**AnimalNodeProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BasicTaxServiceConfigProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**DiscountCalculationModelProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**FixedCalculationModelProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**FixedAmountCalculationModelProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**FixedPriceCalculationModelProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**PercentageCalculationModelProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**QuantityCalculationModelProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**DiscountProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductDiscountProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**DiscountSetProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ExtensibleObjectProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BasicAddressProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BasicExchangeRateServiceConfigProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BasicProfileProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BookmarkListEntryProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductBookmarkProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BookmarkListProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BundledProductLineItemProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BusinessPartnerProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BuyerProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**SellerProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**CatalogCategoryProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**DiscountTemplateProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**DomainProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**JobConfigurationProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**LineItemCtnrProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**BasketProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**InvoiceProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**OrderProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**PackingSlipProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**PlacedOrderProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**QuoteProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**LineItemProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductLineItemProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ServiceLineItemProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**MessageTableProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**MultiSellerLineItemCtnrProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**MultiSellerBasketProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**MultiSellerPlacedOrderProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**PaymentInstrumentInfoProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**PaymentMethodProxy**

- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductAttributeEnumerationProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductAttributeProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductAttributeVariationProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductTypeProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductVariationProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductVariationValueProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProfileGroupProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**RegionalSettingsProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ShippingMethodProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ShoppingListProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**StaticAddressProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**StoreProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**TaxJurisdictionProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**TransactionStoreCommunityProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**TransactionStoreProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**FixedDiscountAmountProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**JobConditionProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**JobTimeConditionProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**JobIntermittentTimeConditionProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**JobRelativeTimeConditionProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**JobMonitorProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**LogConfigurationProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**MethodCalculationModelProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**OrderSuperActorProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**PaymentTransactionProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**PricingGroupProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductLinkProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductSuperActorProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**RuleActionProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**SampleRuleActionProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**RuleConditionProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**RuleProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**RuleSetProxy**
  - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**TimeConditionProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**TimeIntervalConditionProxy**
    - class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**WeekdayConditionProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ShippingServiceConfigProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ShoppingListItemProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**TaxClassDescriptorProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**TaxClassProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**VariableVariationAttributeProxy**

- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ManufacturerProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**MethodCalculationModelAmountProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**OSVersionProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**PricingGroupAssignmentProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductCategoryAssignmentProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductListPriceProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductPriceProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductSuperActorDetailProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProductVariationAssignmentProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ProfileGroupAssignmentProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**RegistryEntryProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**ServerGroupConfigurationProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**SessionInformationProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**StagingInformationProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**StoreDirectoryProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**TaxRateProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**TransactionStoreDirectoryProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**URLConfigurationProxy**
- class com.intershop.enfinity.cartridges.businessrules.bridge.proxy.**UserInformationProxy**

B.4 Geänderte Klassen

Folgende (fettgedruckten) Klassen des enfinity-Systems wurden im Rahmen der Diplomarbeit geändert bzw. erweitert:

- class java.lang.Object
  - class com.persistence.container.internal.EntityBeanImpl
    (implementiert *javax.ejb.EntityBean*)
    - class com.intershop.beehive.core.common.**FWPersistentObjectBean**

Anhang C

Beispiele

C.1 Löschung doppelt registrierter Käufer

Zum Nachweis der Anwendbarkeit der BusinessRules-Komponente im Bereich der Datenpflege des E-Commerce-Systems wurde folgendes Beispiel erstellt. Einträge doppelt registrierter Käufer sollen eliminiert werden. Dabei soll der ältere Eintrag gelöscht werden, so dass nur noch der jüngere Eintrag erhalten bleibt. Für die zu eliminierenden Käuferobjekte sollen ebenfalls das Profile und die zu diesem Käufer gehörenden Adressen gelöscht werden.

Folgende Regeldefinitionen erfüllen diese Anforderungen:

C.1.1 Regeldefinitionen

```
1  ;;
2  ;; Regel: Finde doppelt registrierte Kaeufer, loesche den
       aelteren und dessen Profile, speichere die ProfileID in
        "adressremove"-Fakt zwischen
3  ;;
4  (defrule DOUBLEBUYERREMOVE-finddoublebuyer
5    ?basicProfile1 <- (BasicProfile (UUID ?b1ProfileID) (
       firstName ?firstName) (lastName ?lastName) )
6    ?basicProfile2 <- (BasicProfile (UUID ?b2ProfileID) (
       firstName ?firstName) (lastName ?lastName) )
7    (test (neq ?b1ProfileID ?b2ProfileID) )
8    ?buyer1 <- (Buyer (UUID ?b1UUID) (profileID ?b1ProfileID)
         (lastModified ?b1LastModified) )
9    ?buyer2 <- (Buyer (UUID ?b2UUID) (profileID ?b2ProfileID)
         (lastModified ?b2LastModified) )
10   (test (neq ?b1UUID ?b2UUID) )
11   (test (call ?b1LastModified after ?b2LastModified) )
```

```
12   ?b1Address <- (BasicAddress (profileID ?b1ProfileID) (
        country ?country) (state ?state) (region ?region) (
        prefecture ?prefecture) (province ?province) (
        postalCode ?postalCode) (city ?city) (street ?street)
         (postBox ?postBox) (phoneHome ?phoneHome) )
13   ?b2Address <- (BasicAddress (profileID ?b2ProfileID) (
        country ?country) (state ?state) (region ?region) (
        prefecture ?prefecture) (province ?province) (
        postalCode ?postalCode) (city ?city) (street ?street)
         (postBox ?postBox) (phoneHome ?phoneHome) )
14  =>
15   (assert (addressremove ?b2ProfileID) )
16   (printout t "Removing BasicProfile (UUID: " ?b2ProfileID
        ")" crlf)
17   (ejbretract ?basicProfile2)
18   (printout t "Removing Buyer (UUID: " ?b2UUID ")" crlf)
19   (ejbretract ?buyer2)
20   (commit)
21 )

23 ;;
24 ;; Regel: Loesche die Adress-Objekte des in dem "
     addressremove"-Fakt zwischengespeicherten ProfileIDs
25 ;;
26 (defrule DOUBLEBUYERREMOVE-cleanaddresses
27   (addressremove ?profileID)
28   ?address <- (BasicAddress (UUID ?aID) (profileID ?
        profileID) )
29  =>
30   (printout t "Removing Address (UUID: " ?aID ")" crlf)
31   (ejbretract ?address)
32   (commit)
33 )

35 ;;
36 ;; Regel: Loesche temporaeres "addressremove"-Fakt
37 ;;
38 (defrule DOUBLEBUYERREMOVE-addressescleaned
39   ?flag <- (addressremove ?profileID)
40   (not (BasicAddress (profileID ?profileID)) )
41  =>
42   (retract ?flag)
43   (commit)
44 )
```

C.2 Geschenk bei Bestellung am Geburtstag

Dieses Beispiel demonstriert die Möglichkeit der Modifikation von bestehenden Geschäftsprozessen durch die BusinessRules-Komponente. Der herkömmliche Geschäftsprozess der Bestellung bleibt dabei unverändert. Ordert ein Käufer eine Bestellung wird ein neues Order-Objekt erstellt. Wird das Order-Objekt am Geburtstag des Käufers erstellt, wird durch das untenstehende Regelwerk automatisch ein Geburtstagsgeschenk der Order hinzugefügt. Um zu verhindern, dass ein Käufer in einem Jahr mehrere Geschenke bekommt, wenn er an seinem Geburtstag mehrere Bestellungen ordert, wird das Jahr an dem der Käufer ein Geburtstagsgeschenk bekommen hat, in einem, an das Buyer-Objekt des Käufers gebundene, Fact-Objekt[1] gespeichert.

C.2.1 Regeldefinitionen

```
1  ;;
2  ;; Funktion: Entspricht der Monat (MM) und der Tag (DD) in
       ?dateString (Typ: String, Format: MM/DD/YYYY) dem in ?
      date (Typ: java.util.Date)?
3  ;;
4  (deffunction BIRTHDAYPRESENT-equalStringWithDateMMDD (?
      dateString ?date)
5    (if (<> (str-length ?dateString) 10 ) then
6      (return FALSE)
7    )
8    (bind ?month (+ (get ?date "month") 1) )
9    (bind ?dayofmonth (get ?date "date") )
10   (bind ?ms (sub-string 1 2 ?dateString) )
11   (bind ?ds (sub-string 4 5 ?dateString) )
12   (bind ?mI (new Integer ?ms) )
13   (bind ?dI (new Integer ?ds) )
14   (bind ?mi (call ?mI intValue) )
15   (bind ?di (call ?dI intValue) )
16   (return (and (= ?month ?mi) (= ?dayofmonth ?di) ) )
17 )

19 ;;
20 ;; Funktion: Ist ?yearInt (Typ: int) kleiner als das Jahr
      in ?date (Typ: java.util.Date)?
21 ;;
22 (deffunction BIRTHDAYPRESENT-lessIntWithDateYYYY (?yearInt
      ?date)
23   (bind ?year (+ (get ?date "year") 1900) )
```

[1] siehe Abschnitt 3.3.5

```
24   (return (< ?yearInt ?year) )
25 )

27 ;;
28 ;; Funktion: Ist ?yearInt (Typ: int) gleich dem Jahr in ?
       date (Typ: java.util.Date)?
29 ;;
30 (deffunction BIRTHDAYPRESENT-equalIntWithDateYYYY (?yearInt
       ?date)
31   (bind ?year (+ (get ?date "year") 1900) )
32   (return (= ?yearInt ?year) )
33 )

35 ;;
36 ;; Regel: Erzeuge PROCESSED BIRTHDAY YEAR-Fact fuer Kaeufer
37 ;;
38 ;; WENN
39 ;; eine Order eines Kaeufers vorliegt und kein Fact-Objekt
       mit dem Schluessel "PROCESSED BIRTHDAY YEAR" üfr den
       Kaeufer vorhanden ist,
40 ;; DANN
41 ;; erzeuge neues Faktobjekt üfr den Kaeufer (Schluessel: "
       PROCESSED BIRTHDAY YEAR", Wert: <aktuelles Jahr -1>).
42 ;;
43 (defrule BIRTHDAYPRESENT-addbirthdayfact
44   (Order (buyerID ?buyerID) )
45   (Clock (year ?actualYear) )
46   (not (Fact (objectID ?buyerID) (key "PROCESSED BIRTHDAY
         YEAR")) )
47  =>
48   (isassert "eTS" ?buyerID "PROCESSED BIRTHDAY YEAR" (- ?
         actualYear 1) "")
49   (commit)
50 )

52 ;;
53 ;; Regel: Fuege Geburtstagsgeschenk einer Bestellung hinzu
       , setze PROCESSED BIRTHDAY YEAR-Factobjekt auf aktuelles
        Jahr
54 ;;
55 (defrule BIRTHDAYPRESENT-addpresent
56   (Order (UUID ?orderID) (documentNo ?documentNo) (
         creationDate ?orderDate) (buyerID ?buyerID) (
         primaryKey ?orderPK) (shipFromAddressID ?shipFrom) (
         shipToAddressID ?shipTo) )
```

```
57  (unique (Buyer (UUID ?buyerID) (profileID ?profileID) ) )
58  (unique (BasicProfile (UUID ?profileID) (birthday ?
       birthdayString) (firstName ?firstName) (lastName ?
       lastName) ) )
59  (Product (name "Birthday Present") (SKU ?productID) (
       storeID ?storeID) (shortDescription ?shortDescription)
        (minOrderQuantity ?minOrderQuantity) (stepQuantity ?
       stepQuantity) (taxClassID ?taxClassID) )
60  ?birthdayFact <- (Fact (objectID ?buyerID) (key "
       PROCESSED BIRTHDAY YEAR") (intValue ?fYear) )
61  (Clock (year ?actualYear) )
62  (test (BIRTHDAYPRESENT-equalStringWithDateMMDD ?
       birthdayString ?orderDate) )
63  (test (BIRTHDAYPRESENT-lessIntWithDateYYYY ?fYear ?
       orderDate) )
64  (test (BIRTHDAYPRESENT-equalIntWithDateYYYY ?actualYear ?
       orderDate) )
65  =>
66  (try
67    (bind ?domainHome (call ?*bridgeManager* getEJBHome "
         eTS" "DomainHome") )
68    (bind ?domain (call ?domainHome findDefaultDomain ) )
69    (bind ?orderHome (call ?*bridgeManager* getEJBHome "eTS
         " "OrderHome") )
70    (bind ?productLineItemHome (call ?*bridgeManager*
         getEJBHome "eTS" "ProductLineItemHome") )
71    (bind ?transaction (call ?*bridgeManager*
         getUserTransaction "eTS") )
72    (call ?transaction begin)
73    (bind ?order (call ?orderHome findByPrimaryKey ?orderPK
         ) )
74    (bind ?productLineItem (call ?productLineItemHome
         create ?domain ?productID ?storeID "eCS" ?order) )
75    (set ?productLineItem quantity (new com.intershop.
         beehive.core.common.quantity.Quantity "1.0" "pcs.")
          )
76    (set ?productLineItem minOrderQuantity ?
         minOrderQuantity)
77    (set ?productLineItem stepQuantity (new  com.intershop.
         beehive.core.common.quantity.Quantity "1.0" "pcs.")
          )
78    (set ?productLineItem productName "Birthday Present" )
79    (set ?productLineItem productShortDescription ?
         shortDescription )
```

```
80      (set ?productLineItem singlePricePC (new com.intershop.
           beehive.core.common.money.Money "USD" 0.0) )
81      (set ?productLineItem singleBasePricePC (new com.
           intershop.beehive.core.common.money.Money "USD" 0.0)
            )
82      (set ?productLineItem netPricePC (new com.intershop.
           beehive.core.common.money.Money "USD" 0.0) )
83      (set ?productLineItem netPriceLC (new com.intershop.
           beehive.core.common.money.Money "USD" 0.0) )
84      (set ?productLineItem taxPC (new com.intershop.beehive.
           core.common.money.Money "USD" 0.0) )
85      (set ?productLineItem taxLC (new com.intershop.beehive.
           core.common.money.Money "USD" 0.0) )
86      (set ?productLineItem taxClassID ?taxClassID)
87      (set ?productLineItem shipFromAddressID ?shipFrom)
88      (set ?productLineItem shipToAddressID ?shipTo)
89      (set ?productLineItem basedOnNetPrice TRUE)
90      (call ?order addProductLineItem ?productLineItem)
91      (call ?transaction commit)
92      (modify ?birthdayFact (intValue ?actualYear) )
93      (printout t "Birthday present added to order (
           documentNo: " ?documentNo ") of buyer (firstName
           lastName: " ?firstName " " ?lastName ")" crlf )
94     catch
95      (printout t "RULE: BIRTHDAYPRESENT-addpresent ERROR
           : " (call ?ERROR toString) crlf)
96      (if (neq ?transaction NULL) then
97        (call ?transaction rollback)
98      )
99    )
100   (commit)
101 )
```

C.3 Tier-Rate-Spiel

Das Tier-Rate-Spiel ist ein bekanntes Beispiel zur Demonstration der Funktionsweise von regelbasierten Expertensystemen. Die auf Grundlage der Diplomarbeit implementierte Version demonstriert die Anwendung von regelbasierten Applikationen für das World Wide Web. Weiter zeigt das Beispiel, wie aus der Präsentationsschicht heraus auf die durch die Regelmaschine manipulierte Objekte zugegriffen werden kann sowie die Funktionsweise des Synchronisationsmechanismus'.

C.3.1 Regeldefinitionen

```
;;;
;;;
;;; Beispiel eines selbstlernenden Programms
;;;
;;; Dieses Programm versucht durch Fragen das Tier
    herauszufinden, an das der Benutzer denkt.
;;;
;;; Die Fakten werden in einem Entscheidungsbaum
    gespeichert. Die Knoten des Entscheidungsbaums werden in
     einer EJB (Typ: AnimalNode) persistent gehalten. Zur
    Kommunikation mit dem Pipeline Prozessor werden
    SessionFact-Objekte eingesetzt.
;;;
;;; Dieses Regelwerk wird ausfuehrlich in [Giarratano,
    Joseph C.; Riley, Gary: Expert systems: principles and
    programming, S. 516 - 526] beschrieben.
;;;
;;;

(defrule ANIMALEJB-initialize (declare (salience 1) )
  ?fact <- (SessionFact (key "ejbCurrentNode") (stringValue
      "root") )
  (AnimalNode (root TRUE) (UUID ?uuid) )
 =>
  (modify ?fact (stringValue ?uuid) )
  (commit)
)

(defrule ANIMALEJB-ask-decision-node-question
  (SessionFact (objectID ?oid) (key "ejbCurrentNode") (
     stringValue ?uuid) )
  (AnimalNode (UUID ?uuid) (type "decision") (question ?
     question))
  (not (SessionFact (objectID ?oid) (key "ejbAnswer")) )
 =>
  (isassertsession eTS ?oid "ejbAskUser" 0 ?question)
  (commit)
)

(defrule ANIMALEJB-proceed-to-yes-branch
  ?node <- (SessionFact (objectID ?oid) (key "
     ejbCurrentNode") (stringValue ?uuid) )
  (AnimalNode (UUID ?uuid) (type "decision") (yesNode ?
     yesNode) )
```

```
33   ?answer <- (SessionFact (objectID ?oid) (key "ejbAnswer")
         (stringValue "ja") )
34  =>
35   (ejbretract ?answer)
36   (modify ?node (stringValue ?yesNode) )
37   (commit)
38 )

40 (defrule ANIMALEJB-proceed-to-no-branch
41   ?node <- (SessionFact (objectID ?oid) (key "
        ejbCurrentNode") (stringValue ?uuid) )
42   (AnimalNode (UUID ?uuid) (type "decision") (noNode ?
        noNode) )
43   ?answer <- (SessionFact (objectID ?oid) (key "ejbAnswer")
         (stringValue "nein") )
44  =>
45   (ejbretract ?answer)
46   (modify ?node (stringValue ?noNode) )
47   (commit)
48 )

50 (defrule ANIMALEJB-ask-if-answer-node-is-correct
51   ?node <- (SessionFact (objectID ?oid) (key "
        ejbCurrentNode") (stringValue ?uuid) )
52   (AnimalNode (UUID ?uuid) (type "answer") (answer ?value)
         )
53   (not (SessionFact (objectID ?oid) (key "ejbAnswer") ) )
54  =>
55   (isassertsession eTS ?oid "ejbAskUser" 0 (str-cat "Ich
        denke, dass es ein/e " ?value " ist. Habe ich Recht?")
         )
56   (commit)
57 )

59 (defrule ANIMALEJB-answer-node-guess-is-correct
60   ?node <- (SessionFact (objectID ?oid) (key "
        ejbCurrentNode") (stringValue ?uuid) )
61   (AnimalNode (UUID ?uuid) (type "answer") )
62   ?answer <- (SessionFact (objectID ?oid) (key "ejbAnswer")
         (stringValue "ja") )
63  =>
64   (isassertsession eTS ?oid "ejbAskTryAgain" 0 "")
65   (ejbretract ?node)
66   (ejbretract ?answer)
67   (commit)
```

```
68 )

70 (defrule ANIMALEJB-answer-node-guess-is-incorrect
71   ?node <- (SessionFact (objectID ?oid) (key "
       ejbCurrentNode") (stringValue ?uuid) )
72   (AnimalNode (UUID ?uuid) (type "answer") (answer ?value)
       )
73   ?answer <- (SessionFact (objectID ?oid) (key "ejbAnswer")
       (stringValue "nein") )
74   ?askUser <- (SessionFact (objectID ?oid) (key "ejbAskUser
       ") )
75  =>
76   (isassertsession eTS ?oid "ejbReplaceAnswerNode" 0 ?uuid)
77   (isassertsession eTS ?oid "ejbOldAnimal" 0 ?value)
78   (ejbretract ?answer)
79   (ejbretract ?askUser)
80   (ejbretract ?node)
81   (commit)
82 )

84 (defrule ANIMALEJB-ask-try-again
85   (SessionFact (objectID ?oid) (key "ejbAskTryAgain") )
86   (not (SessionFact (objectID ?oid) (key "ejbAnswer") ) )
87  =>
88   (isassertsession eTS ?oid "ejbAskUser" 0 "Nochmal?")
89   (commit)
90 )

92 (defrule ANIMALEJB-one-more-time
93   ?phase <- (SessionFact (objectID ?oid) (key "
       ejbAskTryAgain") )
94   ?answer <- (SessionFact (objectID ?oid) (key "ejbAnswer")
       (stringValue "ja") )
95   ?askUser <- (SessionFact (objectID ?oid) (key "ejbAskUser
       ") )
96  =>
97   (ejbretract ?phase)
98   (ejbretract ?answer)
99   (ejbretract ?askUser)
100  (isassertsession eTS ?oid "ejbCurrentNode" 0 "root")
101  (commit)
102 )

104 (defrule ANIMALEJB-no-more
105  (SessionFact (objectID ?oid) (key "ejbAskTryAgain") )
```

```
106   (SessionFact (objectID ?oid) (key "ejbAnswer") (
        stringValue "nein") )
107  =>
108   (assert (ejbanimalcleanup ?oid) )
109   (commit)
110 )

112 (defrule ANIMALEJB-cleanup (declare (salience 1) )
113   ?c <- (ejbanimalcleanup ?oid)
114   ?f <- (SessionFact (objectID ?oid) )
115  =>
116   (ejbretract ?f)
117   (commit)
118 )

120 (defrule ANIMALEJB-end
121   ?c <- (ejbanimalcleanup ?oid)
122   (not (SessionFact (objectID ?oid) ) )
123  =>
124   (retract ?c)
125   (commit)
126 )

128 (defrule ANIMALEJB-replace-answer-node
129   ?phase <- (SessionFact (objectID ?oid) (key "
        ejbReplaceAnswerNode") (stringValue ?uuid) )
130   ?data <- (AnimalNode (UUID ?uuid) (type "answer") (answer
         ?value) )
131   ?newAnimalSessionFact <- (SessionFact (objectID ?oid) (
        key "ejbNewAnimal") (stringValue ?newAnimal) )
132   ?questionSessionFact <- (SessionFact (objectID ?oid) (key
         "ejbQuestion") (stringValue ?question) )
133   ?oldAnimalSessionFact <- (SessionFact (objectID ?oid) (
        key "ejbOldAnimal") (stringValue ?oldAnimal) )
134  =>
135   (ejbretract ?phase)
136   (try
137     (bind ?animalHome (call ?*bridgeManager* getEJBHome "
          eTS" "AnimalNodeHome") )
138     (bind ?domainHome (call ?*bridgeManager* getEJBHome "
          eTS" "DomainHome") )
139     (bind ?domain (call ?domainHome findDefaultDomain) )
140     (bind ?transaction (call ?*bridgeManager*
          getUserTransaction "eTS") )
141     (call ?transaction begin)
```

```
142     (bind ?yesChild (call ?animalHome create ?domain) )
143     (set ?yesChild type "answer")
144     (set ?yesChild answer ?newAnimal)
145     (bind ?noChild (call ?animalHome create ?domain) )
146     (set ?noChild type "answer")
147     (set ?noChild answer ?oldAnimal)
148     (call ?transaction commit)
149     (modify ?data (type "decision") (question ?question) (
           yesNode (get ?yesChild UUID)) (noNode (get ?noChild
           UUID)) (answer nil) )
150     (isassertsession eTS ?oid "ejbAskTryAgain" 0 "")
151     (ejbretract ?newAnimalSessionFact)
152     (ejbretract ?oldAnimalSessionFact)
153     (ejbretract ?questionSessionFact)
154    catch
155     (printout t "RULE: ANIMALEJB-replace-answer-node ERROR
           : " (call ?ERROR toString) crlf)
156     (if (neq ?transaction NULL) then
157       (call ?transaction rollback)
158     )
159   )
160   (commit)
161 )
```

C.3.2 Pipeline

Abbildung C.1 zeigt die Pipeline des Tier-Rate-Spiels. Es besteht ausschließlich aus Pipelets zum Bestimmen der SessionID, zur Synchronisation mit der Regelmaschine und Pipelets zum Lesen und Schreiben von den durch die Regelmaschine manipulierten „SessionFacts“[2]. Die Entscheidung, welche Templates angezeigt werden, wird anhand der Existenz von Parametern und „SessionsFacts“ getroffen.

[2]siehe Abschnitt 3.3.6

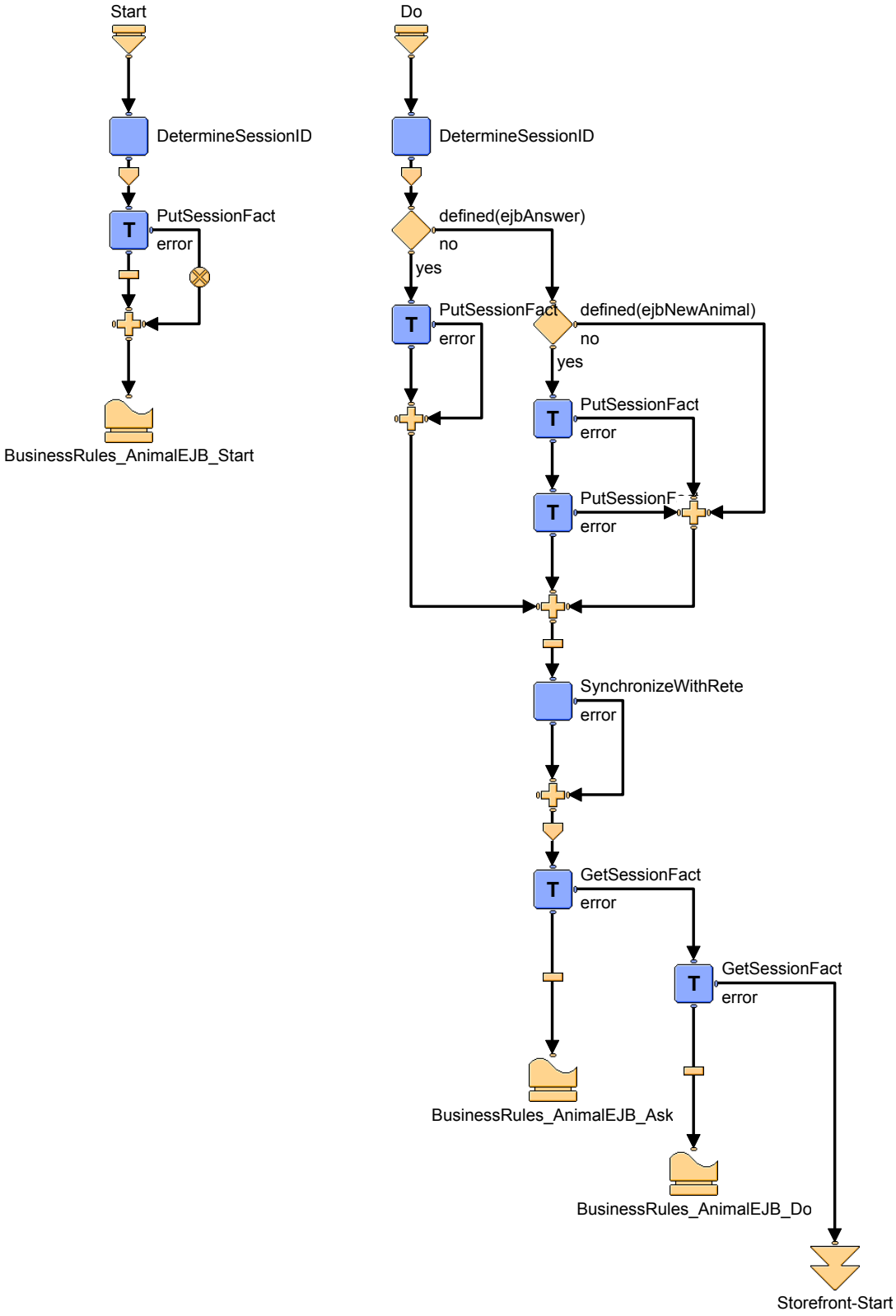

Abbildung C.1: Pipeline des Tier-Rate-Spiels

Abbildungsverzeichnis

Tabellenverzeichnis

Definitionsverzeichnis

Literaturverzeichnis

[BEA] BEA Systems: *BEA Weblogic Portal – Personalization and Interaction Management*. http://edocs.bea.com/wlp/docs40/pdf/p13ndev.pdf – BEA Systems, Inc.; 2315 North First Street; San Jose, CA 95131 U.S.A.; Telephone: +1.408.570.8000; Facsimile: +1.408.570.8901; www.bea.com: BEA Systems, Inc.

[BS84] Buchanan, Bruce G. ; Shortliffe, Edward H.: *Rule-Based Expert Systems - The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, 1984

[Bus97] Business Object Domain Task Force: Common Business Objects / Object Management Group. http://cgi.omg.org/cgi-bin/doc?bom/97-12-04.pdf, 1997. – OMG Document bom/97-12-04

[Bus00a] Business Rule Group: Defining Business Rules ∼ What are They Really? / the Business Rule Group. http://www.businessrulegroup.org/first_paper/apbrules.pdf, Juli 2000. – Final Report, revision 1.3

[Bus00b] Business Rule Group: Organizing Business Plans - The Standard Model for Business Rule Motivation / the Business Rule Group. 2000. – revision 1.0

[Cha98] Chan, Patrick: *The Java Developers ALMANAC 1999*. Reading, Massachusetts 01867 : Addison Wesley Longman, November 1998 (The Java series)

[Che76] Chen, Peter: The Entity-Relationsship Model - Toward a Unified View of Data. In: *ACM Transactions on Database Systems* 1 (1976), März, Nr. 1, S. 9 – 36

[Cov01] Cover, Robin: The XML Cover Pages - Business Rules Markup Language (BRML) / OASIS. http://www.oasis-open.org/cover/brml.html, Mai 2001. – Forschungsbericht

[Deu98] Deutsches Institut für Normung e.V. (DIN): *DIN EN ISO 9241*. Berlin : Beuth-Verlag GmbH, 1998

[DG99] Dorn, Jürgen ; Gottlob, Georg: Künstliche Intelligenz. In: Rechenberg, Peter (Hrsg.) ; Pomberger, Gustav (Hrsg.): *Informatik-Handbuch*. 2. München : Carl Hanser Verlag, 1999, Kapitel E7, S. 975 – 997

[DP00] Denninger, Stefan ; Peters, Ingo: *Enterprise JavaBeans*. Addison-Wesley, 2000

[FH] Friedman-Hill, Ernest J. *Jess*. `http://herzberg.ca.sandia.gov/jess/`

[FH01] Friedman-Hill, Ernest J.: *Jess, The Expert System Shell for the Java Platform*. 6.0a5. Livermore, CA: Distributed Computing Systems, Sandia National Laboratories, Mai 2001. – SAND98-8206 (revised)

[For82] Forgy, Charls L.: Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. In: *Artificial Intelligence* (1982), Nr. 19, S. 17 – 37

[Gen] Genesereth, Michael R.: Knowledge Interchange Format - draft proposed American National Standard (dpANS). `http://logic.stanford.edu/kif/dpans.html`, . – Forschungsbericht. NCITS.T2/98-004

[GHJV96] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. 4. Addison-Wesley, 1996

[Gia] Giarratano, Joseph C.: CLIPS User Guide. `http://www.ghgcorp.com/clips/download/documentation/usrguide.pdf`, . – Forschungsbericht

[GJSB00] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad: *The Java Language Specification*. Second Edition. `ftp://ftp.javasoft.com/docs/specs/langspec-2.0.pdf` : Addison Wesley Longman, Inc., Juni 2000

[Got97] Gottesdiener, Ellen: Business RULES Show Power, Promise. In: *Application Development Trends* 4 (1997), März, Nr. 3

[GR98] Giarratano, Joseph C. ; Riley, Gary: *Expert systems: principles and programming*. PWS Publishing Company, 1998

[Gro99a] Grosof, Benjamin N.: Compiling Prioritized Default Rules Into Ordinary Logic Programs / IBM Research Divison. 1999. – IBM Research Report, RC21472 (96900) 07 May 1999 Computer Science

[Gro99b] Grosof, Benjamin N.: DIPLOMAT: Compiling Prioritized Default Rules into Ordinary Logic Programs, for E-Commerce Applications (extended abstract of Intelligent Systems Demonstration) / IBM Research Divison. 1999. – IBM Research Report, RC21473 (96901) 07 May 1999 Computer Science

[HMW98] Hupe, Patrick ; Matthes, Florian ; Wegner, Holm: Ein bruchloser Übergang von der Prozeßmodellierung zu kooperativen Software-Architekturen / Technische Universität Hamburg-Harburg, Arbeitsbereich Softwaresysteme. `http://www.sts.tu-harburg.de/papers/1998/HMW98`, 1998. – Forschungsbericht

[IBM02] IBM Corporation: *WebSphere Personalization, Version 4.0.* `http://www-3.ibm.com/software/webservers/personalization/doc/v40/ws_personalization_v4e.pdf` – IBM Corporation; Software Group; Route 100; Somers, NY 10589; U.S.A.: IBM Corporation, März 2002

[ILO01a] ILOG: *ILOG JRules 3.1 – Business Action Language – User's Manual.* Gentilly, Frankreich: ILOG, Januar 2001

[ILO01b] ILOG: *ILOG JRules 3.1 – User's Manual.* Gentilly, Frankreich: ILOG, Januar 2001

[INTa] INTERSHOP Software Entwicklungs GmbH: *Design: Using Templates and ISML.* INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTb] INTERSHOP Software Entwicklungs GmbH: *Developer Guide: Programming with enfinity Cartridge API.* INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTc] INTERSHOP Software Entwicklungs GmbH: *Developer Guide: Programming with enfinity Remote XML Interface.* INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTd] INTERSHOP Software Entwicklungs GmbH: *enfinity 2 Feature Enhancements.* INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTe] INTERSHOP Software Entwicklungs GmbH: *Inside INTERSHOP enfinity.* INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTf] INTERSHOP Software Entwicklungs GmbH: *INTERSHOP enfinity: Business White Paper.* INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTg] INTERSHOP Software Entwicklungs GmbH: *INTERSHOP enfinity: Data Import/Export Formats & Configuration*. INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTh] INTERSHOP Software Entwicklungs GmbH: *INTERSHOP enfinity: Die e-commerce Lösung für Großunternehmen*. INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTi] INTERSHOP Software Entwicklungs GmbH: *INTERSHOP enfinity: Technical White Paper*. INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTj] INTERSHOP Software Entwicklungs GmbH: *Pipelines: Using the Visual Pipeline Manager*. INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTk] INTERSHOP Software Entwicklungs GmbH: *Sell Anywhere mit INTERSHOP*. INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTl] INTERSHOP Software Entwicklungs GmbH: *Store Administration: Using the enfinity Management Center*. INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTm] INTERSHOP Software Entwicklungs GmbH: *System Administration: Using the Server Management Console*. INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[INTn] INTERSHOP Software Entwicklungs GmbH: *Visual Pipeline Manager Tools: Pipeline Debugger*. INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG

[Int99] International Business Machines: IBM SanFransisco: Concepts & Facilities / International Business Machines. `http://www.ibm.com/software/ad/sanfrancisco/`, 1999. – Forschungsbericht

[INT01] INTERSHOP Software Entwicklungs GmbH: *INTERSHOP enfinity – Beehive Platform 2 API Specification*. INTERSHOP Tower, 07743 Jena, Deutschland: INTERSHOP Communications AG, Oktober 2001

[LKK93] Lockemann, Peter C. ; Krüger, Gerhard ; Krumm, Heiko: *Telekommunikation und Datenhaltung*. München, Wien : Carl Hanser Verlag, 1993

[Lup00] Lup, Low Wai: IntelliClean – The smartest way to clean data. / School of Computing, National University of Singapore. `http://www.comp.nus.edu.sg/~iclean/`, 2000. – Forschungsbericht

[Min74] Minsky, Marvin: A Framework for Representing Knowledge / MIT - AI Laboratory. `http://www.ai.mit.edu/people/minsky/papers/Frames/frames.html`, Juni 1974. – Memo 306

[Nau60] Naur, Peter: Revised Report on the Algorithmic Language ALGOL 60. In: *Communications of the ACM* 3 (1960), Mai, Nr. 5, S. 299 – 314

[NGR88] Nayak, Pandurang ; Gupta, Anoop ; Rosenbloom, Paul: Comparison of the Rete and Treat Production Matchers for Soar (A Summary) / Stanford University; University of Southern California. Stanford University, Stanford, CA 94305; University of Southern California, Marina del Rey, CA 90292, 1988. – Forschungsbericht

[Obj00] Object Management Group: OMG Unified Modelling Language Specification / Object Management Group. `http://www.omg.org/cgi-bin/doc?formal/00-03-01.pdf`, März 2000. – version 1.3

[OW93] Ottman, Thomas ; Widmayer, Peter: *Algorithmen und Datenstrukturen.* Zweite Auflage. Mannheim, Leipzig, Wien, Zürich : BI Wissenschaftsverlag, 1993 (Reihe Informatik)

[Plo99] Plotkin, David: Business Rules Everywhere. In: *Intelligent Enterprise Magazine* 2 (1999), März, Nr. 4

[Pup88] Puppe, Frank: *Einführung in Expertensysteme.* Springer Verlag, 1988

[RD00] Rahm, Erhard ; Do, Hong Hai: Data Cleaning: Problems and Current Approaches / Universität Leipzig; Microsoft Research. `http://www.research.microsoft.com/research/db/debull/A00dec/rahm.ps`, 2000. – Forschungsbericht

[RMH+97] Rational Software ; Microsoft ; Hewlett-Packard ; Oracle ; Sterling Software ; MCI Systemhouse ; Unisys ; ICON Computing ; IntelliCorp ; i-Logix ; IBM ; ObjecTime ; Platinum Technology ; Ptech Taskon ; Reich Technologies ; Softeam: Object Constraint Language Specification / Object Management Group. `http://cgi.omg.org/cgi-bin/doc?ad/97-08-08.pdf`, 1997. – OMG Document ad/97-08-08, Version 1.1, 1 September 1997

[Rom99] Roman, Ed: *Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition.* John Wiley & Sons, Inc, 1999

[Sch01] Schreiber, Hendrik: Verzaubertes Java - Codename Merlin: JDK 1.4. In: *c't – Magazin für Computer Technik* (2001), September, Nr. 20, S. 188 – 190

[Sel02] Selman, Daniel: JSR 94: Java Rule Engine API / Java Community Process Program. `http://jcp.org/jsr/detail/094.jsp`, Februar 2002. – Java Specification Request

[SH87] Schnupp, Peter ; Huu, Chau Thuy Nguyen: *Expertensystempraktikum*. Springer Verlag, 1987

[SH99] Stahlknecht, Peter ; Hasenkamp, Ulrich: *Einführung in die Wirtschaftsinformatik*. Springer Verlag, 1999

[Sim00] Simons, Patrick: *Extending and Implementing the Stable Model Semantics*. Espoo, Finland, Helsinky University of Technology Labaratory for Theoretical Computer Science, HUT-TCS-A58, April 2000

[SSW+00] Sangonas, Konstantinos ; Swift, Terrance ; Warren, David S. ; Freire, Juliana ; Rao, Prasad: *The XSB System - Version 2.2 - Volume 1: Programmer's Manual*. `http://www.cs.sunysb.edu/~sbprolog/xsb-page.html`: Stony Brook - State University of New York, Katholieke Universiteit Leuven, Universidade Nova de Lisboa, April 2000

[Sun] Sun Microsystems: *What's new in the Enterprise JavaBeans 2.0 Specification?* `http://java.sun.com/products/ejb/2.0.html`: Sun Microsystems

[Sun97] Sun Microsystems: *JavaBeans*. `ftp://ftp.javasoft.com/docs/beans/beans.101.pdf`: Sun Microsystems, Juli 1997

[Sun98] Sun Microsystems: *ENTERPRISE JAVABEANS TECHNOLOGY Server Component Model for the Java Platform*. `http://java.sun.com/products/ejb/white_paper.pdf`: Sun Microsystems, Dezember 1998

[Sun99a] Sun Microsystems: *Enterprise JavaBeans Specification, v1.1*. `http://java.sun.com/j2ee`: Sun Microsystems, Dezember 1999

[Sun99b] Sun Microsystems: *The Java 2 Enterprise Edition Specification, v1.2*. `http://java.sun.com/j2ee`: Sun Microsystems, Dezember 1999

[Sun99c] Sun Microsystems: *Java 2 SDK, Standard Edition Documentation*. `http://java.sun.com/products/jdk/1.2/docs/index.html`: Sun Microsystems, 1999

[Sun99d] Sun Microsystems: *Java Remote Method Invocation Specification*. ftp://ftp.java.sun.com/docs/j2se1.3/rmi-spec-1.3.pdf: Sun Microsystems, Dezember 1999

[Sun99e] Sun Microsystems: *Java Servlet Specification, v2.2*. http://java.sun.com/j2ee: Sun Microsystems, August 1999

[Sun99f] Sun Microsystems: *Java Transaction API*. http://java.sun.com/j2ee: Sun Microsystems, April 1999

[Sun99g] Sun Microsystems: *JavaServer Pages Specification*. http://java.sun.com/j2ee: Sun Microsystems, Dezember 1999

[Sun00a] Sun Microsystems: *The Java 2 Enterprise Edition Developer's Guide*. http://java.sun.com/j2ee: Sun Microsystems, Mai 2000

[Sun00b] Sun Microsystems: *Java 2 SDK, Enterprise Edition Documentation Bundle*. http://java.sun.com/j2ee/sdk_1.2.1/techdocs/index.html: Sun Microsystems, Mai 2000

[Sun01] Sun Microsystems: *Enterprise JavaBeans Specification, Version 2.0*. http://java.sun.com/j2ee: Sun Microsystems, August 2001

[Sun02a] Sun Microsystems: *Java 2 SDK, Standard Edition Documentation*. http://java.sun.com/products/jdk/1.4/docs/index.html: Sun Microsystems, 2002

[Sun02b] Sun Microsystems: *Java Message Service Specification*. ftp://ftp.java.sun.com/pub/jms/jder123/jms-1_1-fr-spec.pdf: Sun Microsystems, April 2002

[Sun02c] Sun Microsystems: *The Java Tutorial – A practical guide for programmers*. ftp://ftp.javasoft.com/docs/tutorial.zip: Sun Microsystems, 2002

[SW01] Schneider, Uwe ; Werner, Dieter: *Taschenbuch der Informatik*. 4, aktualisierte Auflage. München Wien : Fachbuchverlag Leipzig, 2001

[Ver] Verschiedene. *CLIPS*. http://www.ghgcorp.com/clips/CLIPS.html

[von93] von Foerster, Heinz: Epistemologie und Kybernetik: Rückblick und Ausblick. Ein Fragment. In: Weibel, Peter (Hrsg.): *KybernEthik*. Merve Verlag, 1993, S. 92 – 108

[von94] VON FOERSTER, Heinz: Betrifft: Erkenntnistheorien. In: SCHMIDT, Siegfried J. (Hrsg.): *Wissen und Gewissen – Versuch einer Brücke*. Suhrkamp Taschenbuch Wissenschaft. Suhrkamp Verlag, 1994, S. 364 – 370

[Wat85] WATZLAWICK, Paul: *Die erfundene Wirklichkeit*. Piper Verlag, 1985

[Wes99] WESKE, Matthias: Business-Objekte: Konzepte, Architekturen, Standards. In: *Wirtschaftsinformatik* (1999), Nr. 1, S. 4 – 11

[WR00] WARREN, David S. ; RAO, Prasad: *The XSB System - Version 2.2 - Volume 2: Library and Interfaces*. `http://www.cs.sunysb.edu/~sbprolog/xsb-page.html`: Stony Brook - State University of New York, Katholieke Universiteit Leuven, Universidade Nova de Lisboa, April 2000

Index