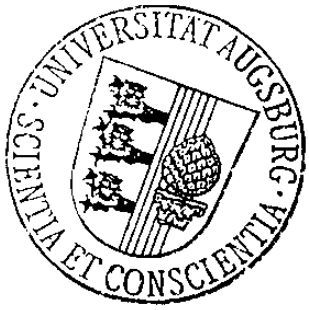

Produktiver Einsatz eines J2EE-Application Servers am Beispiel von IBM Websphere

Seminararbeit

im Rahmen des Projektseminars (Skiseminar) im WS 03/04

Themensteller: Prof. Dr. Rony G. Flatscher

Betreuer: Sebastian Klöckner

vorgelegt von:

Fabian Matthiessen
Fabian.Matthiessen@student.uni-augsburg.de

Ralph Sauer
Ralph.Sauer@student.uni-augsburg.de

Abgabetermin: 12. Dezember 2003

Inhaltsverzeichnis

Abkürzungsverzeichnis

BMP	Bean Managed Persistence
BSF	Bean Scripting Framework
CMP	Container Managed Pesistence
EAI	Enterprise Application Integration
FQDN	Fully Qualified Domain Name
IDE	Integrated Development Environment
JMS	Java Message Service
JSP	Java Server Page
JNDI	Java Naming and Directory Interface
JNI	Java Native Interface
JRE	Java Runtime Environment
JSP	Java Server Page
MdB	Message driven Bean
UML	Unified Modelling Language
W2K	Windows 2000
W2K3	Windows 2003
WSAD	WebSphere Application Developer

Abbildungsverzeichnis

0 Einleitung

Moderne Softwareentwicklung hat Dank der Hilfe von Tools, Architekturkonzepten und vor allem durch die Entwicklung neuer Programmiersprachen deutliche Schritte nach vorne gemacht. Viele neue Konzepte sind zusammen mit der Programmiersprache JAVA umgesetzt worden. JAVA als plattformunabhängige Lösung hat mit den Paketen aus J2EE (Java 2 Enterprise Edition) auch Lösungen für größere und verteilte Anwendungen erhalten.

Im ersten und umfangreichsten Teil dieser Seminararbeit wird man sich mit den verschiedenen Schritten auf dem Weg zu einer Java Enterprise Anwendung auseinandersetzen. Gestartet wird mit einem Überblick über die UML (Unified Modelling Language), mit deren Hilfe es möglich ist, (semi-) formal Software zu beschreiben. Durch verschiedene Diagramme und Konzepte können schon im Entwurf und Design genaue Angaben über das Aussehen und Verhalten der zu entwickelnden Software gemacht werden. Mit Hilfe von IBM Rational XDE können UML-Diagramme erstellt und schon ein Grundgerüst für die spätere Implementierung generiert werden, weshalb auf dieses Werkzeug ebenfalls eingegangen wird. Die Entwicklung selbst erfolgt im WebSphere Application Developer, der ebenfalls vorgestellt wird. Dabei werden jedoch vor Allem die Bereiche abgedeckt, die in [Lehn2003] nicht erwähnt wurden (Teamunterstützung und Funktionen für größere Projekte). Am Ende des Kapitels gehen die Autoren noch auf den Application Server an sich ein, zeigen Probleme beim Deployment von Applikationen auf und wie sich diese umgehen lassen.

Das zweite Kapitel beschäftigt sich mit der Verifizierung der Diplomarbeit zum Thema „WebSphere“ von Frau Lehnert. Hierbei soll vor Allem geprüft werden in wieweit die gezeigten Schritte nachvollziehbar und die Beispiele ohne größere Probleme umsetzbar sind.

Der letzet Teil beleuchtet einige ausgewählte Themen im Umfeld von WebSphere bzw. J2EE. So wird zum einen der Bezahlstandard PayCircle vorgestellt, der auf einem J2EE-Server basiert, zum anderen wird gezeigt, wie die Skriptsprache Rexx bzw. Object Rexx in das Bean-Scripting Framework des WebSphere Application Servers integriert werden kann. Zuletzt wird mit den Message driven Beans eine spezielle Form von Enterprise-Java-Beans vorgestellt.

1 J2EE – Entwicklung von der Wiege bis zur Bahre

1.1 Überblick (Was ist WS, J2EE, UML)

Die Nutzung von J2EE und UML ermöglicht es unter Heranziehung von Entwicklungstools ein Softwareprojekt vom Requirements Engineering über das Design des Softwareprodukts, der automatischen Codegenerierung bis hin zur Auslieferung und Wartung zu betreuen, was die Entwicklungsarbeit klarer, konsequenter und zum Teil einfacher gestaltet.

1.1.1 UML; Eine Sprache zur Systembeschreibung

UML, die Unified Modelling Language dient der Beschreibung von Systemen und Abläufen, der strukturierten Darstellung von Zusammenhängen zwischen Systemkomponenten sowie den Interaktionen des Systems mit den Benutzern. Nachfolgend soll die Basisnotation von UML in der Version 1.5 sowie die wichtigsten Diagrammarten erläutert und durch einfache Beispiele anschaulich dargestellt werden. Auch die Historie von UML sowie deren Entwickler wird kurz vorgestellt werden.

1.1.2 J2EE; Java für Große

J2EE, Java 2 Enterprise Edition ist die konsequente Weiterentwicklung der Java Standard Edition. Mit Hilfe der Java Standard Edition werden lokale Applikationen entwickelt. Sie laufen immer auf dem Client Rechner ab, sind für verteilte Systeme aber schlecht nutzbar. Beim der Programmierung zum Beispiel eines Internetshops, werden Komponenten benötigt, die auf dem Serversystem ablaufen und auf Anfragen von (Web-)Clients reagieren. Für diesen Zweck, und noch viel mehr, gibt es J2EE. Die Entwicklung von Applikationen mit J2EE, sowie Tools zum Design – hier IBM-Rational XDE – und zur Programmierung – hier WebSphere Application Developer – werden genauer betrachtet und mit prägnanten Beispielen verdeutlicht.

1.1.3 IBM WebSphere®, mehr als ein Applikationsserver

WebSphere, eine Produktfamilie der Firma IBM ist mehr als nur ein J2EE Applikationsserver. Vielmehr ist der Begriff WebSphere als Rahmen für verschiedenste Applikationen zu sehen, die für das e-business on demand zur Verfügung stehen. Es gibt den Applikationsserver, eine Entwicklungsumgebung, Portalsoftware, ubiquitäre

Komponenten, Spracherkennung sowie Komponenten für Business-Integration (Planung, Steuerung und Kontrolle von Geschäftsprozessen). Es sollen an dieser Stelle die grundlegenden Produkte WSAD und WebSphere Application Server vorgestellt werden.

1.2 UML; Geschichte und Notation

Wie auch in der klassischen Architektur war es irgendwann notwendig geworden vom mehr oder weniger begabten „Basteln“ zu einem strukturierten, vorausplanenden Denken und Handeln beim Erstellen von Softwareprojekten zu gelangen. Es genügte durch immer komplexer werdende Projekte nicht mehr, dass sich einfach ein paar Programmierer ins stille Kämmerchen begaben und dort eine Software schrieben, die, mit viel Glück, auch den Anforderungen des Kunden entsprach. Auf der NATO-Tagung 1967 wurde erstmals der Begriff „Software Engineering“ geprägt. Das Ziel war nun vorgegeben: Ablösung der „Kunst“ der Programmierung durch „ingenieurmäßiges“ Vorgehen [Sche2002 Kap1 S. 11].

Wie auch in anderen Disziplinen sind nun fünf (angepasste) Bereiche eines Projektes abzudecken: Analyse, Design, Implementierung, Test sowie Einsatz & Wartung [Sche2002 Kap2 S. 2].

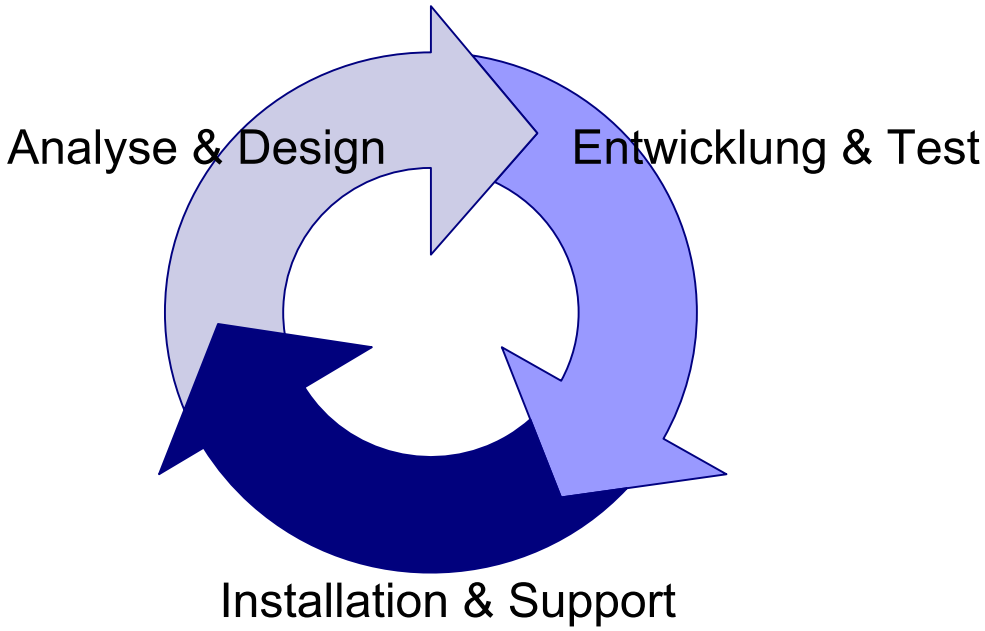

Abb. 1.1: Lifecycle einer Applikation

Durch den Einsatz der objektorientierten Sprachen im größeren Stil ab ca. 1995 war es nun nötig geworden, die bis dahin etwa 50 verschiedenen objektorientierten Sprachdialekte und Methodiken zu vereinheitlichen.

1.2.1 Die Erfinder, erste Schritte und aktueller Stand

Ivar Jacobson, James Rumbaugh und Grady Booch, die drei maßgeblichen Erfinder von UML hatten in den Jahren 1991/1992 bereits selbst Techniken zu Analyse, Design und Modellierung entwickelt und begannen 1994/1995 bei der Firma Rational zusammenzuarbeiten und waren schnell als die „3 Amigos“ bekannt. 1996 wurde bei der Object Management Group (OMG) eine Gruppe gebildet (unter anderem mit Firmen wie IBM, Rational und HP), was dann im selben Jahr zum ersten Entwurf von UML führte. Im darauf folgenden Jahr wurde UML in der Version 1.0 veröffentlicht. [Sche2002 Kap5 S. 33f].

Momentan ist die Version 1.5 von UML aktuell, die Version 2.0 nähert sich mit großen Schritten der Fertigstellung [OmgMod2003] .

1.2.2 Was ist UML?

> *UML ist die standardisierte Modellierungssprache zur Visualisierung, Spezifikation, Konstruktion und Dokumentation von Software(-lastigen) Systemen, die nach objektorientierten Prinzipien entworfen werden. [Sche2002 Kap5 S. 30]*

Folgende Aspekte zeichnen UML aus:

- Die Standardisierung der Sprache erlaubt eine Kommunikation der am Projekt beteiligten Parteien, ohne erst die Bedeutung der Modelle/Diagramme erklären zu müssen.
- UML ist eine semi-formale Sprache mit festgelegter graphischer Syntax, wobei die Semantik teilweise nicht eindeutig festgelegt ist.
- Es existieren direkte Verknüpfungen zu Programmiersprachen wie Java, C++ oder Visual Basic.
- Es gibt direkte Verbindungen zu relationalen wie auch objektorientierten Datenbanken.

- UML ist gedacht sowohl für die Entwicklung von neuen Projekten, als auch für das Reverse Engineering (Analyse bereits vorhandener Software).

1.2.3 Grundbegriffe der Notation

Zuerst soll festgelegt werden, was Software-Entwicklung mit UML bedeutet, nämlich eine Kombination von Objektorientierung, graphischer Modellierung mit der Standardsprache UML und einem evolutionären Entwicklungsmodell. [Sche2002 Kap5 S. 2]

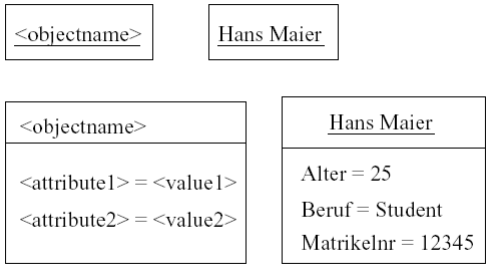

Abb. 1.2 Darstellung von Objekten

Im Rahmen dieser Arbeit soll Java als objektorientierte Sprache zu Grunde gelegt werden.

Wenn nun eine Beschreibungssprache für eine objektorientierte Sprache existiert, ist es die Intention, die Objekte dieser Sprache, deren Zusammenspiel, Verhalten, Lebensdauer und Zugehörigkeit zu beschreiben (und vieles mehr).

Dies führt zuerst zur Darstellung von Objekten in UML, wobei ein Objekt (als konkrete Instanz) als ein Rechteck mit unterstrichenem Namen dargestellt wird. Optional kann das Objekt auch Attribute (Werte) enthalten; siehe Abb. 1.1

Wie in objektorientierten Sprachen üblich, sind Objekte konkrete Instanzen von Klassen, die als Sammlung gleichartiger Objekte angesehen werden können. In UML werden Klassen als Rechtecke dargestellt, allerdings wird der Klassenname nicht unterstrichen. In zwei Teilrechtecken unter dem Klassennamen können (müssen aber nicht) Attribute und Operationen angegeben werden, wobei diese auch typisiert werden können; siehe Abb. 1.2

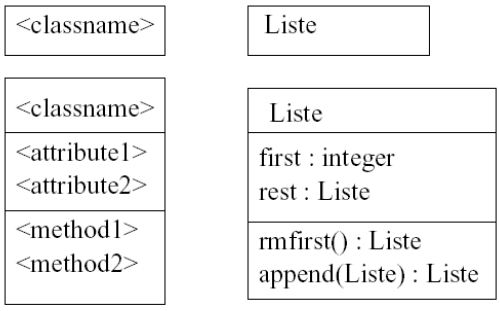

Abb. 1.3: Modellierung von Klassen in UML

Verständlicherweise können nicht nur Attribute von Klassen typisiert werden, sondern auch die Instanzen einer Klasse (die konkreten Objekte). Sie können somit nach dem Objektnamen eine Klassenangabe besitzen, wobei auch anonyme Klassen erlaubt sind; siehe Abb. 1.3

In allen weiterführenden Diagrammarten sind nun verschiedene Objekte miteinander verbunden, durch einfache Linien und Pfeile, denen je nach Diagrammart unterschiedliche Bedeutungen zukommen.

Mit diesem Grundgerüst können die verschiedenen Arten von Diagrammen betrachtet werden, die in der UML zur Beschreibung verschiedener Sachverhalte angeboten werden.

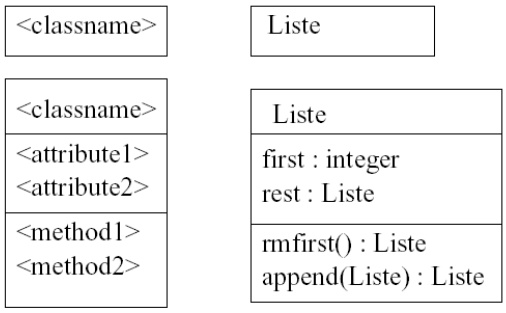

Abb. 1.4 Modellierung von Instanzen

Alle Beispiele aus [Sche2002 Kap5 S. 10ff].

1.2.4 Die grundlegenden Diagramme

Die möglichen UML-Diagramme lassen sich in drei Kategorien aufteilen [OmgUML2003]:

- Strukturdiagramme wie Klassendiagramme, Objektdiagramme, Komponentendiagrame und Deploymentdiagramme.
- Verhaltensdiagramme wie das Use Case Diagramm (das bei der Anforderungsanalyse in einem Use Case verwendet wird); Sequenzdiagramme, Aktivitätsdiagramme, Kollaborationsdiagramme, und Statechart Diagramme.
- Modellmanagementdiagramme, die Packages, Subsysteme und Modelle enthalten.

Diese Arbeit beschränkt sich auf drei der wichtigsten Diagramme:

- Das Use Case Diagramm, das zusammen mit einer textuellen Ablaufbeschreibung die Interaktion des Benutzers mit dem System beschreibt.
- Das Klassendiagramm, das für die Implementierung von großer Wichtigkeit ist.
- Das Sequenzdiagramm, das eine genaue zeitliche Abfolge der Aktionen der beteiligten Teile beschreibt.

Mit diesen drei Diagrammen lassen sich das Verhalten des Systems nach Außen, die Interaktion der beteiligten Systemteile und die dazu benötigten Klassen relativ komplett und konkret beschreiben. Damit liegt die zu entwickelnde Software als (in Teilen)

genau definiertes Projekt und nicht nur in einer ungenauen Beschreibung vor, die unterschiedlich interpretiert werden kann.

1.2.4.1 Das Use Case Diagramm

Bevor auf das Use Case Diagramm eingegangen wird, soll zuerst einmal der Use Case selbst erklärt und informell definiert werden.

> *Der Use Case wird in der Anforderungsanalyse erstellt und ist eine Beschreibung, wie Anwender ein System benutzen, um eine bestimmte, genau definierte Aufgabe zu erledigen, die ein definiertes Ergebnis (das Ziel) hat. [Sche2002 Kap6 S. 3]*

Der wesentliche Punkt bei einem Use Case ist, dass er das System aus der Sicht des Anwenders sowie dessen Aktionen mit dem System beschreibt. Für verschiedene Aktionen werden verschiedene Use Cases benötigt (z.B. macht es einen Unterschied, ob bei einem Bezahlvorgang mit Kreditkarte oder bar bezahlt wird). Es ist wichtig zu erkennen, dass der Use Case nicht nur zur Anforderungsanalyse benutzt wird, sondern darüber hinaus auch als Grundlage für die spätere Entwicklung genutzt wird.

Der prinzipielle Ablauf beim Erstellen eines Use Cases sieht wie folgt aus:

- Identifikation der Akteure (am System beteiligte Einheiten)
- Erstellen einer Grobbeschreibung (z.B. „Einkaufen“)
- Erstellen einer Feinbeschreibung wichtiger Use Cases („Einkaufen eines Buches bei Amazon, Bezahlen mit Kreditkarte“)

In diesem Rahmen soll auf eine tiefer gehende Beschreibung der Use Cases an sich verzichtet werden, da der hier wichtige Teil, das Diagramm mit obiger Beschreibung verständlich erklärt werden kann.

Im Use Case Diagramm (Kommunikationsmodell) werden die Akteure (Kunde, Kassierer, Kreditkartensystem) als Strichmännchen dargestellt. Das System, mit dem die Akteure interagieren ist als Kasten dargestellt, in dem die einzelnen Use Cases durch Ovale repräsentiert werden. Einfache Interaktionen sind normale Linien, initiale Aktionen sind Linien mit einem Pfeil an deren Ende. Use Cases können innerhalb des Systems untereinander verbunden sein, entweder mit einem «includes», das eine Teil-Ganzes Beziehung darstellt (Das Bezahlen des Einkaufes kann durch Barbezahlung oder Kreditkartenbezahlung erfolgen) oder durch ein «extends», das eine Generalisierungsbeziehung darstellt. In dem nachfolgenden Diagramm könnte als

Generalisierung zum Beispiel ein Use Case „Bezahlen“ eingefügt werde, der dann mit «extends» die beiden spezielleren Use Cases enthält.

Das nachfolgende Use Case Diagramm (Abb 1.4) beschreibt eine Supermarktkasse, die vom Filialleiter gestartet wird, an der sich die Kassierer anmelden, an der der Kunde seinen Einkauf bezahlt und ggf. das Kreditkartenprüfsystem hinzugezogen wird. Alternativ kann der Kunde auch bereits gekaufte Ware zurückgeben.

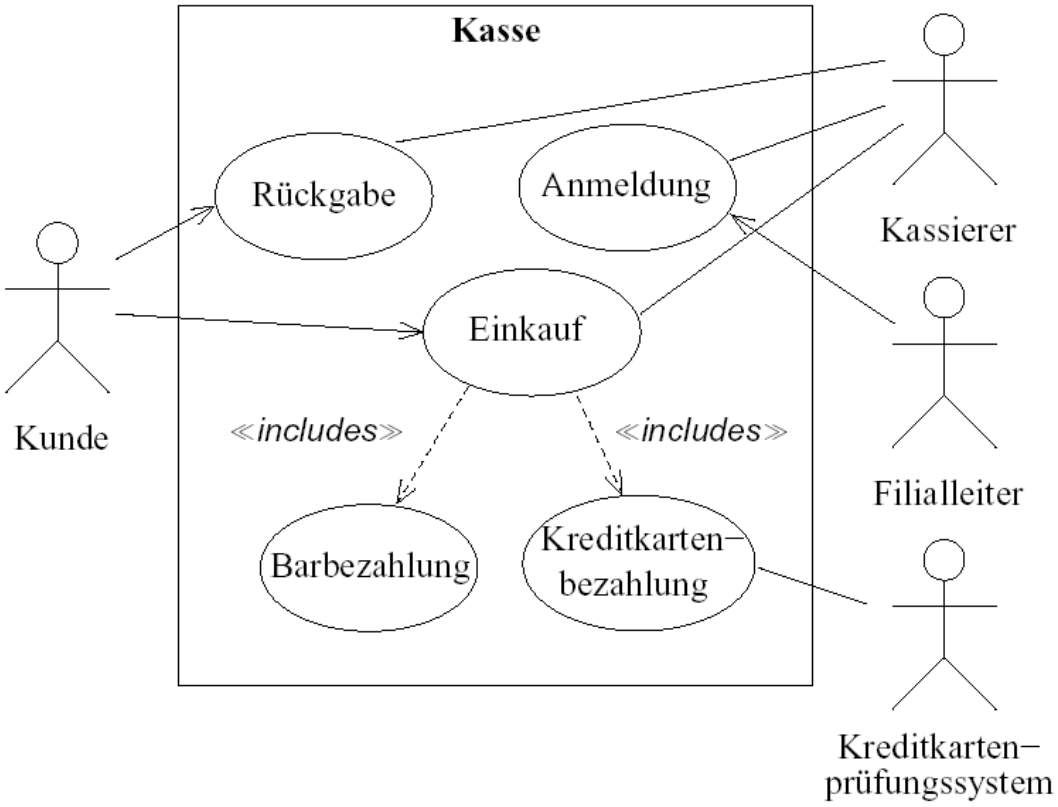

Abb. 1.5: Beispiel für ein Use-Case-Diagramm

1.2.4.2 Das Sequenzdiagramm

Das nächste Diagramm, das betrachtet werden soll, ist das Sequenzdiagramm. Aufbauend auf die Use Cases wird hier das System als sogenannte „Black Box“ betrachtet. Wichtig ist nun, wie mit dem System kommuniziert wird, also eine Konkretisierung der Use Cases dahingehend, dass nun schon Systemoperationen festgelegt werden können, die später in der Implementierung berücksichtigt werden. Jeder Akteur, der mit dem System kommuniziert, erhält eine vertikale, gestrichelte Linie (sog. swim lane), ebenso das System. An jeder Stelle im Use Case, an der ein Akteur dem System etwas mitteilt (Systemoperation), wird eine waagerechte Linie zwischen

den swim lanes des Akteures und des Systems gezeichnet. Diese wird beschriftet mit dem Namen und den Attributen der Systemoperation.

Nachfolgendes Beispiel zeigt, wie ein Kunde an der Kasse seine Artikel vorlegt, dann der Kasse mitteilt, dass er nun alles angegeben hat und dann seinen Einkauf bezahlt.

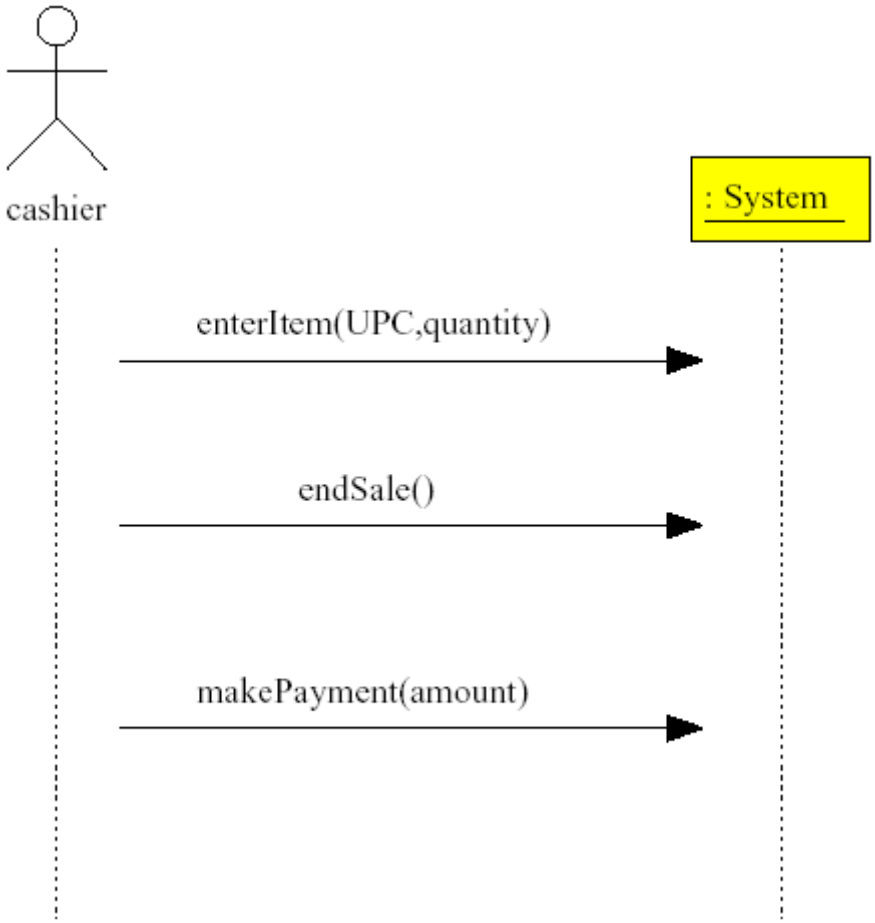

Abb. 1.6: Ein Squenzdiagramm zur Beschreibung eines Bezahlvorgangs

1.2.4.3 Das Klassendiagramm

Als drittes Diagramm wird nun ein Klassendiagramm betrachtet. Klassendiagramme zeigen eine statische Sicht eines System(-ausschnitt)s. Sie enthalten die beteiligten Klassen, deren Attribute und Operationen, Assoziationen zwischen den Klassen sowie Typhierarchien, eventuelle Constraints (einschränkende Bedingungen) und wichtige Kommentare.

Nachfolgendes Diagramm (Abb. 1.5) zeigt ein Klassendiagramm angelehnt an das oben gezeigte Use Case Diagramm, das einen Supermarkt abbildet. Die Klasse Store

repräsentiert das Geschäft an sich. Im Geschäft arbeiten Angestellte (in einem Package gezeigt) und das Geschäft hat eine bis mehrere Kassen (POST). Das Personal (sowohl Manager als auch Angestellte) bedienen die Kassen, wobei ihnen unterschiedliche Systemoperationen zur Verfügung stehen.

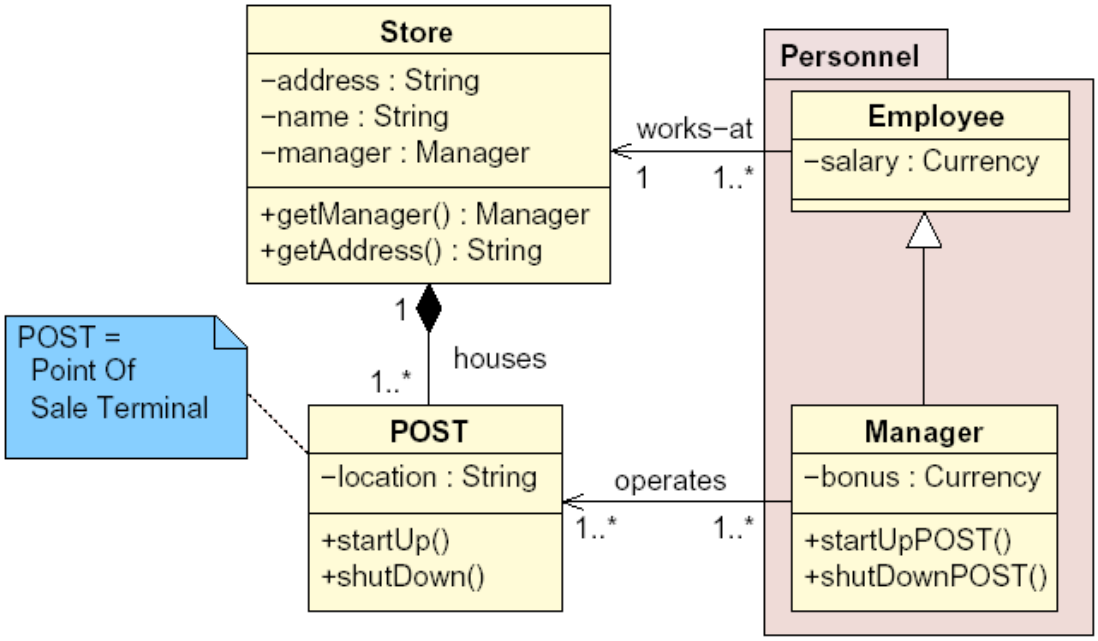

Abb. 1.7: Klassendiagramm für unser Kassenbeispiel

Die gezeigten Diagramme können zwar auch von Hand erzeugt werden, oder in einem reinen Zeichenprogramm, das ggf. UML unterstützt, sinnvoller ist es jedoch, die Modellierung in eine Entwicklungsumgebung zu integrieren, die dann anhand der Diagramme eine Dokumentation sowie Grundgerüste des Programmcodes erzeugen kann. Auf eines dieser Tools, IBM Rational XDE soll im nächsten Abschnitt eingegangen werden.

1.3 Modellierung mit UML und XDE

Die Firma Rational (dies war und ist teilweise der Arbeitsplatz der UML-Erfinder) hat als Tool zur Modellierung mit UML (unter anderem) das Produkt XDE (Extended Development Environment) entwickelt.

1.3.1 Einordnung von XDE

Um Applikationen von Analyse & Design über die Implementierung & Test bis zum Release toolgestützt zu entwickeln, unterstützt Rational XDE den ersten Abschnitt des Lifecycles einer Anwendung.

In XDE werden die Diagramme entwickelt, die die spätere Grundlage der Implementierung bilden. Im Gegensatz zu einigen Freeware-Tools zur UML-Modellierung ist XDE kostenpflichtig (ca. 3500 US-Dollar). Dafür bietet XDE eine integrierte Anbindung an den WebSphere Application Developer und in einer anderen Version auch an Visual Studio von Microsoft.

XDE ist eine – durch den Funktionsumfang bedingt – sehr komplexe Anwendung, auf die hier nicht im Detail eingegangen werden soll, da dies den Rahmen dieser Arbeit sprengen würde.

1.3.2 Die Vorteile der UML-Entwicklung mit XDE

Die weiter oben bereits vorgestellten Diagrammarten sollen nun in ähnlichen Beispielen in XDE entwickelt werden, um dann ein Codegerüst für die (Java) Applikation zu erhalten. Bewusst haben die Autoren auf eine komplexe J2EE-Anwendung verzichtet, um die Beispiele einfach zu halten.

Nachdem man in XDE ein neues Projekt angelegt hat (quasi als Rahmen) kann damit begonnen werden, z.B. ein Use-Case Diagramm zu erzeugen, was in nachstehender Grafik zu sehen ist. Dieser Screenshot zeigt aber auch, dass die Oberfläche durchaus verwirrend sein kann.

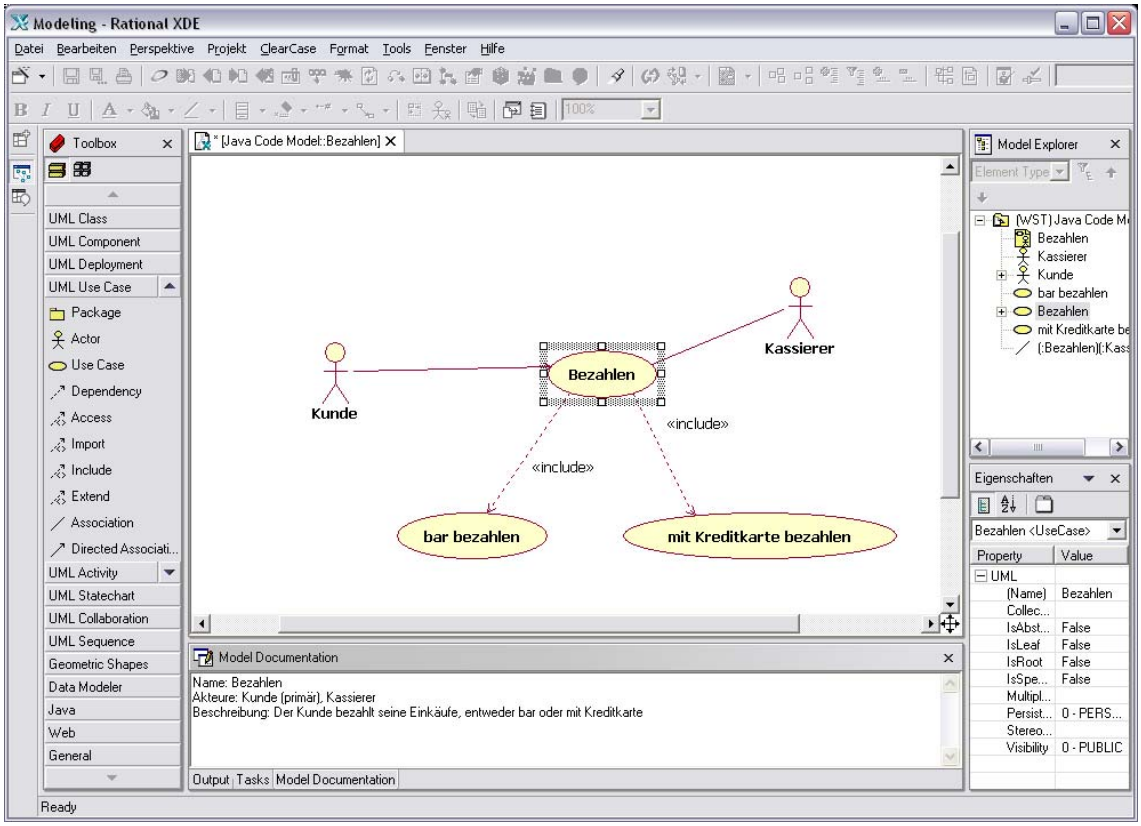

Abb. 1.8: Use-Case-Diagramm in Rational XDE

Die linke Spalte enthält die möglichen Objekte, die dem Diagramm hinzugefügt werden können, die rechte Seite die Baumstruktur des Projektes und des Diagramms. Darunter erkennbar sind die Eigenschaften des jeweils ausgewählten Objektes, in der Mitte das erzeugte Diagramm und die textuelle Grobbeschreibung im unteren Bereich.

Nachdem das Use Case Diagramm erstellt wurde, kann darauf aufbauend ein Sequenzdiagramm erstellt werden, wobei dieselben Objekte (die bereits angelegt wurden) verwendet werden können.

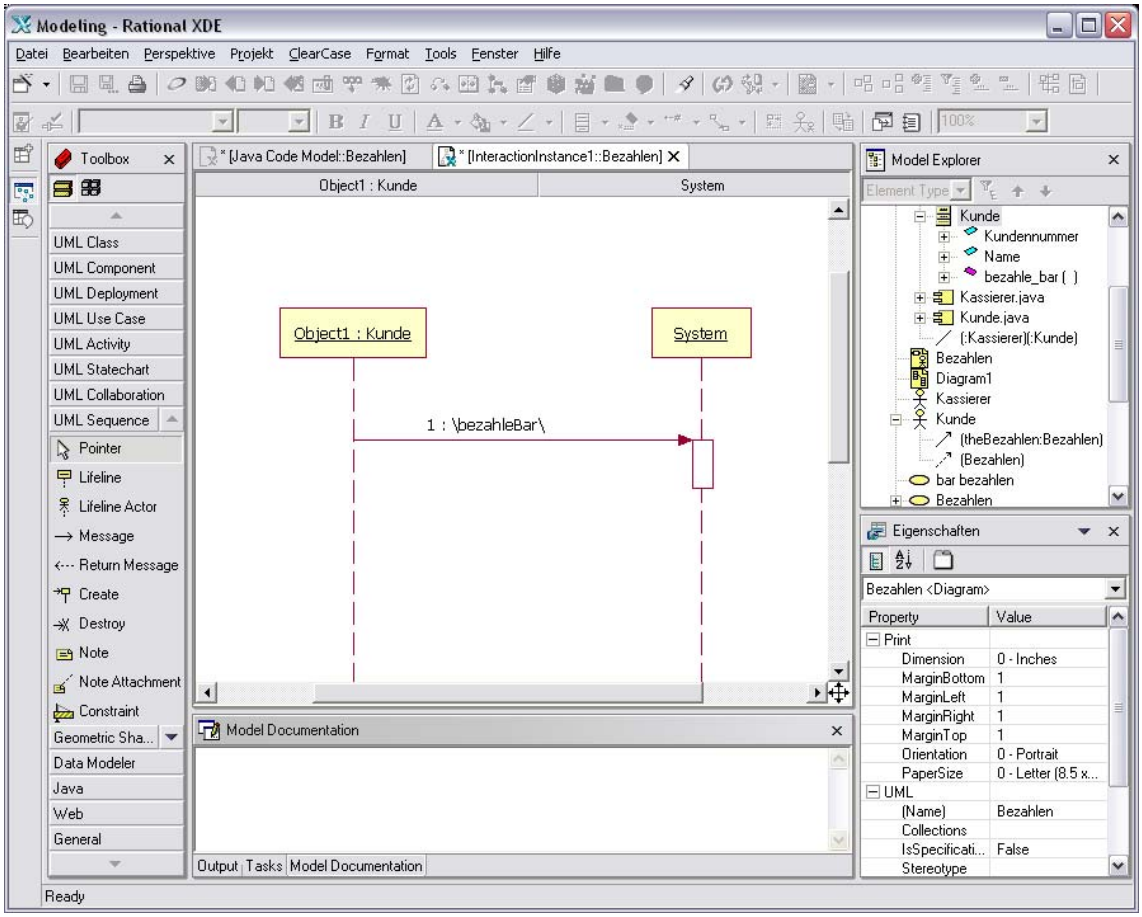

Abb. 1.9: Sequenzdiagramm in Rational XDE

Deutlich erkennbar sind die beiden Objekte mit den dazugehörigen Swim-lanes.

Nachdem die Objekte zum großen Teil identifiziert sind und Systemoperationen feststehen, kann mit der Erstellung eines Klassendiagramms begonnen werden.

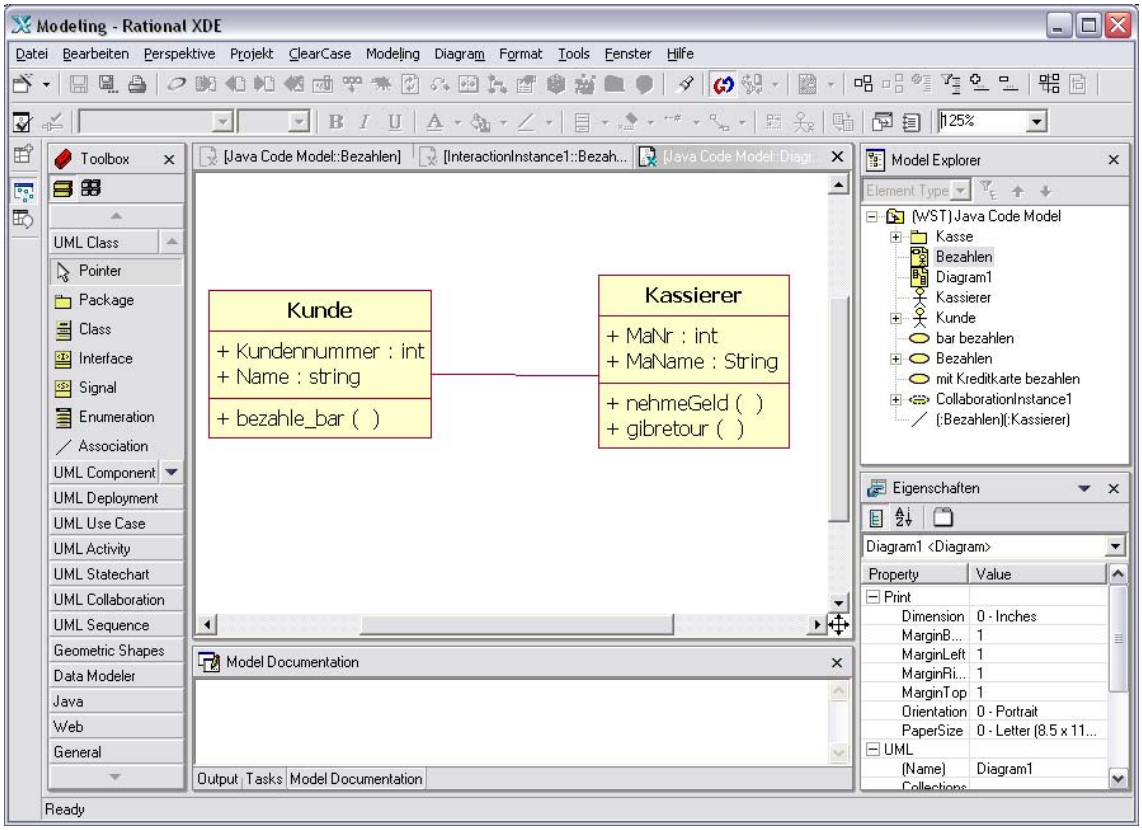

Abb. 1.10: Klassendiagramm in Rational XDE

Abb. 1.10 zeigt die zwei Klassen, eine für den Kunden, eine zweite für den Kassierer. Beide haben eine Assoziation, die im weiteren Verlauf noch näher spezifiziert werden muss. Auch haben beide Attribute und Operationen, die hier durch das + als 'public' gekennzeichnet sind.

Der größte Vorteil bei der Modellierung mit XDE ist, dass sich anhand des Klassendiagramms ein Codegerüst erzeugen lässt. Dies kann dann bearbeitet werden, entweder direkt in XDE oder in anderen Programmierumgebungen.

Ein weiterer Vorteil ist die automatische Konsistenzabsicherung zwischen den einzelnen Komponenten durch XDE, zum einen innerhalb der Diagramme und zum anderen aber auch zwischen Diagrammen und Quelltext. Sobald z.B. in einem Klassendiagramm Klassen, Methoden oder Operationen verändert werden, wird das Codegerüst automatisch angepasst. Aber auch in der anderen Richtung nimmt XDE dem Entwickler viel Arbeit ab. Werden z.B. während der Implementierung Methoden im Quelltext ergänzt, so werden auch automatisch die dazugehörigen Diagramme angepasst.

Nachstehend nun ein Screenshot des automatisch erzeugten Codes für die Klasse Kassierer:

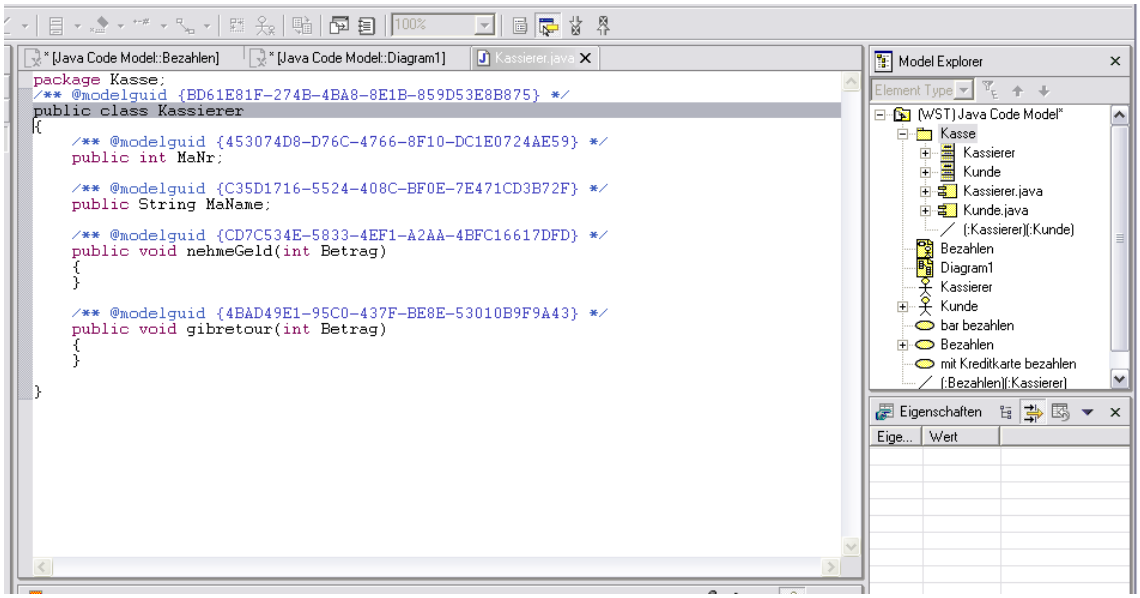

Abb. 1.11: Das Quellcodegerüst wird automatisch von XDE generiert

Ein weiterer wichtiger Punkt bei der Entwicklung von Software ist die Dokumentation der geleisteten Arbeit. Es genügt nicht, nur die fertige Software auszuliefern, der Kunde wünscht sich neben einem Handbuch auch konkrete Dokumente zur Spezifikation. Die Programmierer müssen sich über die Arbeit der Kollegen informieren können, ebenso wie sie sich einen Überblick über das Gesamtprojekt verschaffen müssen, um die von Ihnen entwickelten Teile einordnen zu können. XDE bietet hierzu verschiedene Methoden an. Zum einen kann man die erzeugten Diagramme separat weiter verwenden, XDE bietet aber auch an, automatisiert eine (z.B. html-basierte) Dokumentation anzulegen. Diese kann automatisch jeden Tage aktualisiert auf einen Webserver gespielt werden, wo dann sowohl der Kunde als auch die anderen Entwickler, Projektleiter und Manager sich einen Überblick über den aktuellen Stand der Analyse und des Designs verschaffen können. Diese Methode vermeidet erheblichen Aufwand, der sonst zu betreiben wäre, um allen beteiligten Parteien diesen Einblick zu gewähren.

1.3.3 Von XDE zu WSAD

Mit den bisher betrachteten Werkzeugen und Methodiken sind die Analyse und das Design abgeschlossen. Nun steht die Implementierung des Projektes an. Hier kommt der zweite Baustein aus der IBM-Familie zum Tragen, der WebSphere Studio Application Developer (WSAD). XDE bietet eine hochintegrierte Schnittstelle zum WSAD an. Die kompletten Projektstrukturen, Codegerüste und vieles mehr können direkt an den WSAD übergeben werden, bzw. XDE integriert sich wahlweise auch direkt in den

WSAD. Somit erfolgt mit den beiden Programmen XDE und WSAD der Brückenschlag zwischen Analyse, Design und der Implementierung/Test.

1.4 J2EE-Programmierung mit WebSphere Application Developer 5.0

Der WebSphere Application Developer (WSAD) stellt eine integrierte Entwicklungsumgebung (IDE ≈ integrated development environment) für Java Enterprise Anwendungen zur Verfügung. Mit Hilfe seiner Werkzeuge lassen sich Enterprise-Anwendungen relativ schnell und unkompliziert erstellen. So können z.B. Entity-Beans automatisch generiert, JSPs grafisch bearbeitet und die ganze Applikation auf Fehler und Verstöße gegen Standards hin geprüft werden.

Wird der Lifecycle einer Anwendung betrachtet, unterstützt WSAD die Phasen der Implementierung und des Tests. Es sollen nun im Folgenden einige ausgewählte Features des WebSphere Application Developers vorgestellt werden. Für eine grundlegende Einführung in die Funktionen und verschiedenen Perspektiven des WSAD wird auf [Lehn2003] verwiesen.

1.4.1 Ausgewählte Funktionen von WSAD

Aus der Fülle von Funktionen und Hilfen, die der Application Developer anbietet, werden an dieser Stelle einige herausgegriffen, die den Autoren während der Entwicklungszeit als besonders interessant erschienen sind. Die Auswahl wurde absichtlich so getroffen, dass die hier gegebene Information, die generelle Einführung druch [Lehn2003], vervollständigt. Auf eine komplette Behandlung der grundlegenden Funktionen wird aus diesen Gründen hier verzichtet. Weiterführende Informationen können in [WNA+2003] nachgelesen werden.

1.4.1.1 Assistent zum Erstellen einer EJB

Eine der grundlegenden Funktionen, mit denen vermutlich jeder Entwickler einer J2EE-Anwendung früher oder später in Kontakt kommen wird, ist der Assistent zum Erstellen von neuen EJBs. Weniger bekannt ist vielleicht, dass er auch verwendet werden kann, um bestehende Java-Dateien in das Projekt als Bean aufzunehmen. Sollten bereits Java-Dateien von einem anderen Projekt existieren und sollen diese in das aktuelle Projekt integriert werden, lässt sich das ebenfalls über diesen Assistent durchführen.

Um den Assistenten zu starten, wir die J2EE-Hierarchie-Perspektive benötigt. Mit einem Rechtsklick auf ein EJB-Projekt Neu→Enterprise Bean wird der Assistent aktiv.

Im ersten Schritt wird das Enterprise-Projekt ausgewählt, in dem die Bean gespeichert werden soll. Es wird von den IBM-Entwicklern empfohlen nicht mehr als ein Enterprise-Projekt pro Workspace zu verwalten, so dass im Regelfall hier keine große Auswahl vorhanden ist.

Im nächsten Schritt können die essenziellen Eigenschaften der Bean festgelegt werden. Der Typ der Bean legt ihre spätere Aufgabe innerhalb der Anwendung fest.

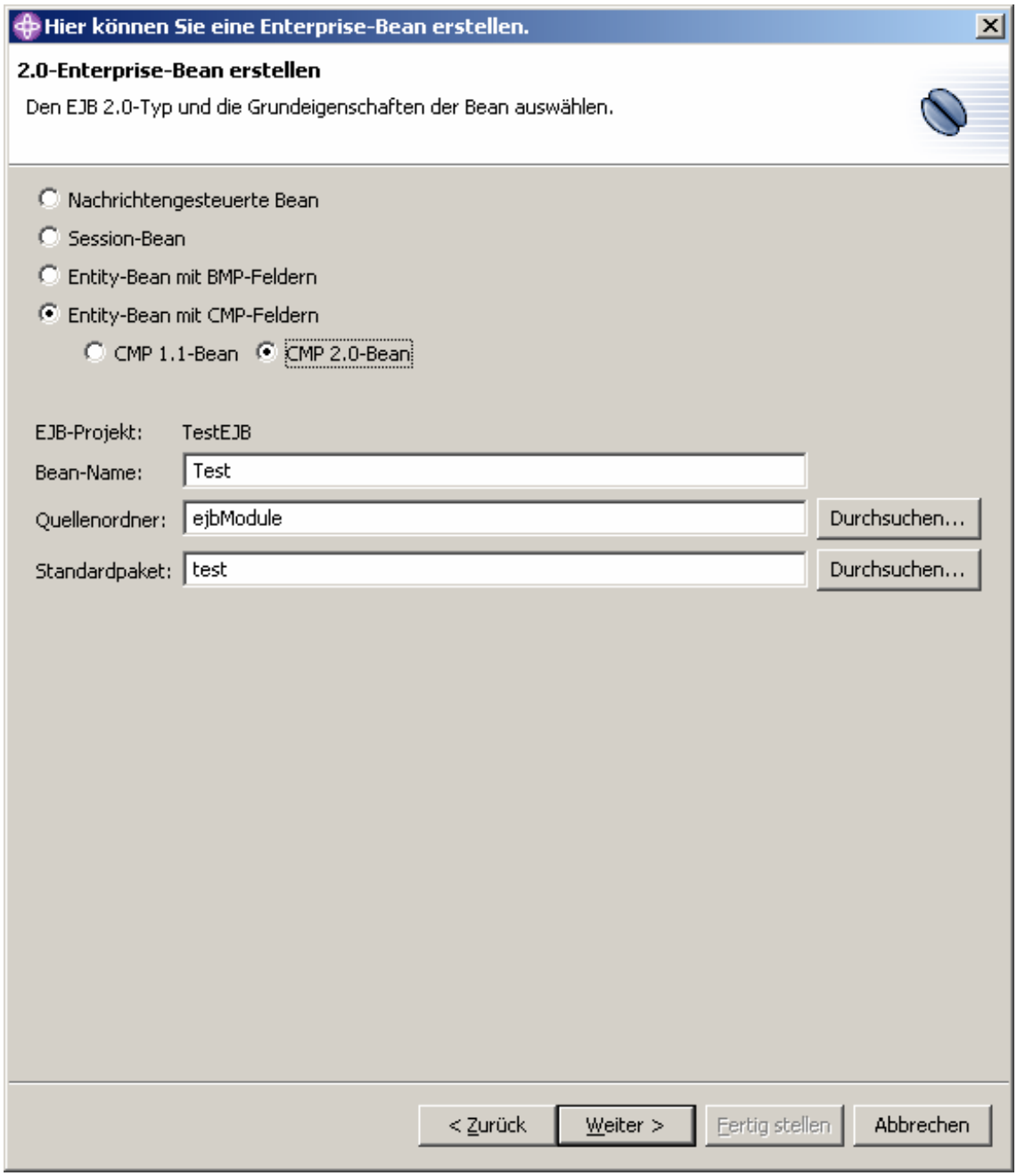

Abb. 1.12: Auswahl für den Typ der zu erstellenden Bean

In J2EE wurden folgende verschiedene Typen von Enterprise-Beans spezifiziert:

- Message driven Beans
  Diese Art Bean wird in Kapitel 3.3 noch genauer behandelt. Sie arbeiten als Empfänger von Nachrichtendiensten und werden aktiv sobald eine Nachricht für sie eintrifft. Auf diese Weise lassen sich Events in relativ lose gekoppelten Systemen realisieren.

- Session Beans
  Session Beans bilden in den meisten Fällen die Anlaufstelle für externe Applikationen bzw. Servlets und steuern den eigentlichen Programmablauf. Es existieren zwei Ausprägungen, stateless und statefull, wobei im letzteren Fall jede Client-Applikation eine eigene Session-Bean zugeordnet bekommt, die sich den Status der Kommunikation für die Dauer der Verbindung merkt.

- Entity Beans mit Bean Managed Persistence (BMP)
  Entity Beans dienen der persistenten Speicherung von Daten. Das besondere an der Bean managed Persistence ist, dass sich der Entwickler um die Speicherung seiner Beans selbst kümmern muss. Dies ermöglicht ihm maximale Gestaltungsfreiheit, bedeutet aber im Gegensatz zur Container managed Persistence einen erheblichen Mehraufwand.

- Entity Beans mit Container Managed Persistence (CMP)
  Wie die andere Sorte Entity Beans können auch diese Daten persistent speichern, allerdings wird die Arbeit der Speicherung bei diesem Typ vom EJB-Container übernommen. Der Programmierer kann sich also voll und ganz auf die Entwicklung seiner Anwendung konzentrieren und muss sich über die Speicherung oder Transaktionen keine Gedanken machen.

Der gewünschte Bean-Typ wird ausgewählt. Je nachdem welcher Typ Bean gewählt wurde, unterscheiden sich die nächsten Seiten des Assistenten etwas voneinander.

Bei der Message driven Bean muss festgelegt werden, ob die Bean für eine Warteschlange oder für ein bestimmtes Topic konfiguriert werden soll. Außerdem kann hier der Name des Listenerports festlegt werden, der später auch noch im Server zu konfigurieren ist.

Bei der Erstellung von Session Beans wird als nächstes festgelegt, ob sie sich ihren Status merken soll, oder nicht. Mit Hilfe der „Sichten“ kann festgelegt werden, welche Methoden der Bean nach außen sichtbar sein sollen. 'Ferner Client' steht dabei für Applikationen, die nicht in derselben Laufzeitumgebung laufen, also auch für

Anwendungen auf demselben Rechner, sofern sie nicht direkt in WebSphere installiert sind.

Die Erstellung von Entity Beans mit BMP läuft im Wesentlichen gleich ab, wie die Erstellung von Session Beans.

Bei Entity Beans mit CMP müssen zusätzlich die Attribute eingeben werden, die in der Bean verwaltet werden sollen.

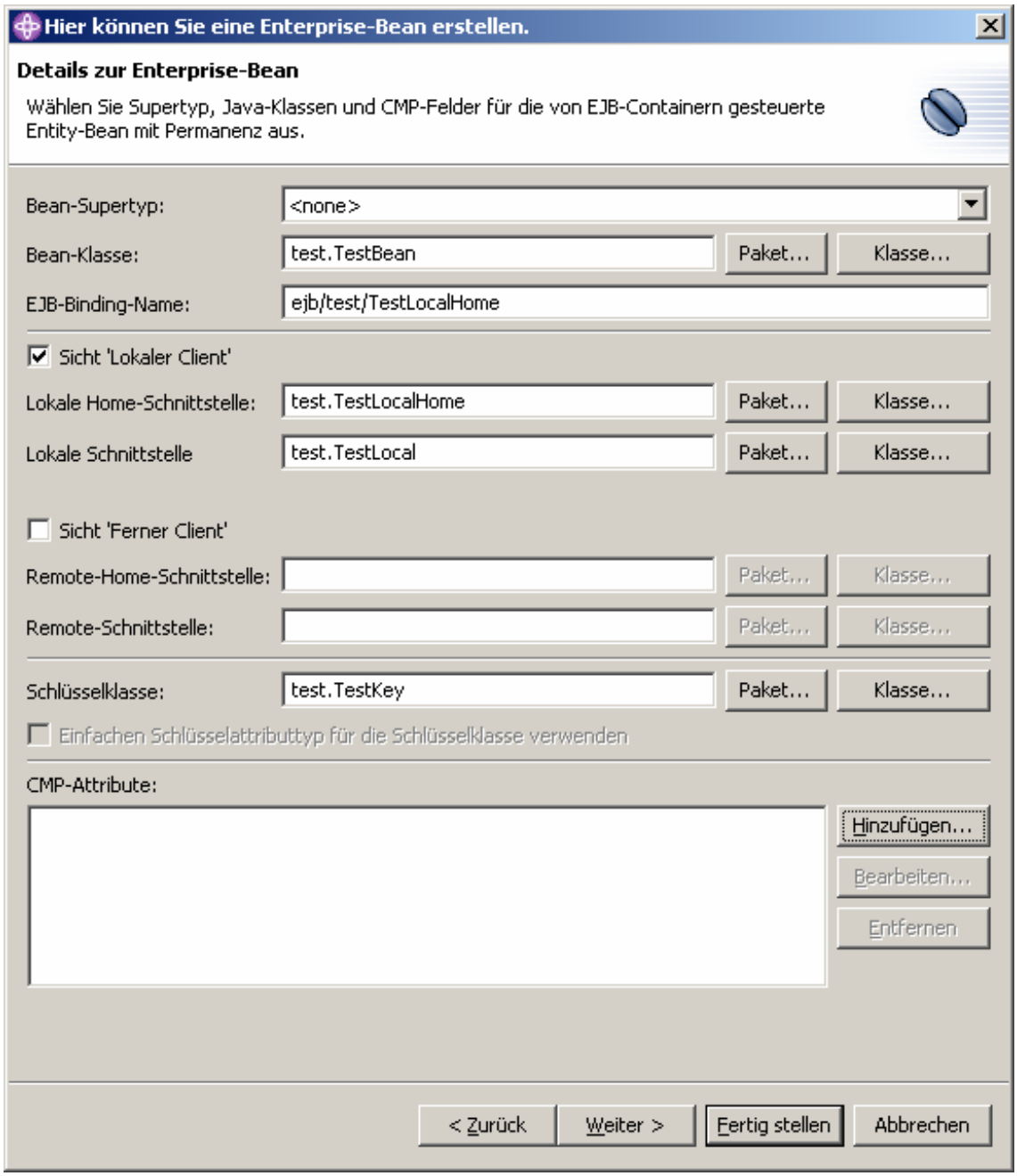

Abb. 1.13: Assistent für die Erstellung von Beans mit CMP

Mit Hilfe des Hinzufügen-Buttons kann festgelegt werden, welche Attribute die Bean haben soll und zusätzlich, ob es möglich sein soll, diese von „außen“ zu ändern.

Auf der nächsten Seite wird bestimmt, welchen Interfaces die Bean genügen soll, bzw. von welchen sie evtl. Eigenschaften und Methoden vererbt bekommen soll. Auf diese Weise kann redundanter Code vermieden werden, indem gemeinsame Attribute und Methoden von zwei Klassen in eine gemeinsame Super-Klasse ausgelagert werden, und beide dann von ihr erben.

Nach Abschluss des Assistenten sehen Sie, wie sich Ihre neu estellte EJB in die Hierarchie eingliedert. In der Hierarchie-Ansicht können EJBs am leichtesten bearbeitet werden. Um Einstellungen zu verändern, die den Deployment-Deskriptor betreffen (dazu später mehr) doppelklicken Sie einfach auf die Bean-Darstellung. Die einzelnen Klassen der Bean können durch einen Klick auf das + vor der Bean zum Vorschein gebracht werden. Sie können die Klasse dann durch einen Doppelklick öffnen, um die eigentliche Logik hineinzuprogrammieren.

1.4.1.2 Refactoring

Bei der Aufarbeitung von älterer Software oder bei einem länger andauernden Entwicklungsprozess, ergibt sich meist das Problem, dass die Strukturen mit der Zeit sehr starr und unflexibel werden. Der Refactoring-Ansatz versucht dieses Problem zu beheben:

> *Refaktorisierung ist eine Technik der Softwareentwicklung, bei der man den Programmcode in kleinen Schritten umgestaltet und aufräumt, ohne dabei sein von außen sichtbares Verhalten zu ändern. [ScVi2003]*

Ziel der Refaktorisierung ist es, die Lesbarkeit eines Programms zu verbessern und die Strukturen flexibler zu gestalten. Mit der immer weiteren Verbreitung von agilen Entwicklungsprozessen, wie z.B. „Extreme Programming“, steigt die Anforderung an den Programmierer, seinen Code schnell und unkompliziert an die neuen Anforderungen anpassen zu können. Mit jeder Iteration des Entwicklungsprozesses sollen neue Funktionen hinzugefügt und bestehende verändert werden, was sehr schnell zu unflexiblen und nur schwer überschaubaren Strukturen führen kann.

Das Refactoring soll dieser Code-Entropie entgegenwirken. Beim Aufräumen strukturiert der Entwickler Klassen neu, entfernt unter anderem Redundanzen und benennt Methoden und Bezeichner um. Die Gefahr dabei ist jedoch, dass sich auf diese Weise in bereits fertigen Code, der schon getestet und überprüft wurde, neue Programmierfehler einschleichen können.

Vor der Überarbeitung von Code, sollten sich die Entwickler zuerst Gedanken machen, wie der entstehende Code auszusehen hat. Sie müssen sich auf gemeinsame Standards für die Namensgebung und den Stil der Formatierungen einigen. Empfohlen sei für diejenigen, die sich diese Überlegung sparen möchten, Suns Java Code Convention (http://java.sun.com/docs/codeconv).

Eine zusätzliche Überlegung empfiehlt sich dennoch: Viele Programmierer deklarieren am Anfang der Methode sämtliche verwendeten Variablen. Das hat nicht nur negativen Einfluss auf die Lesbarkeit des Codes, sondern vergrößert auch den Gültigkeitsbereich und verleitet dazu, Variablen mehrfach für unterschiedliche Zwecke zu verwenden. Die Variablen sollten also stets direkt vor ihrer Verwendung deklariert werden. Das erleichtert das spätere Refactoring.

Eine der nützlichsten Möglichkeiten des Refactoring ist das Extrahieren von Methoden, dabei wird ein Teilstück einer Methode in eine neue Methode verschoben. Nach [Fowl2000] ist es angebracht, diese Technik für eine Quellcodesequenz innerhalb einer längeren Methode anzuwenden, sobald zu deren Verständnis ein Kommentar nötig erscheinen würde. Durch das Extrahieren von Methoden aus solchen längeren Codestücken wird meist redundanter (≈doppelt vorhandener) Code sichtbar. In WebSphere Application Developer gelingt das Extrahieren von Methoden problemlos.

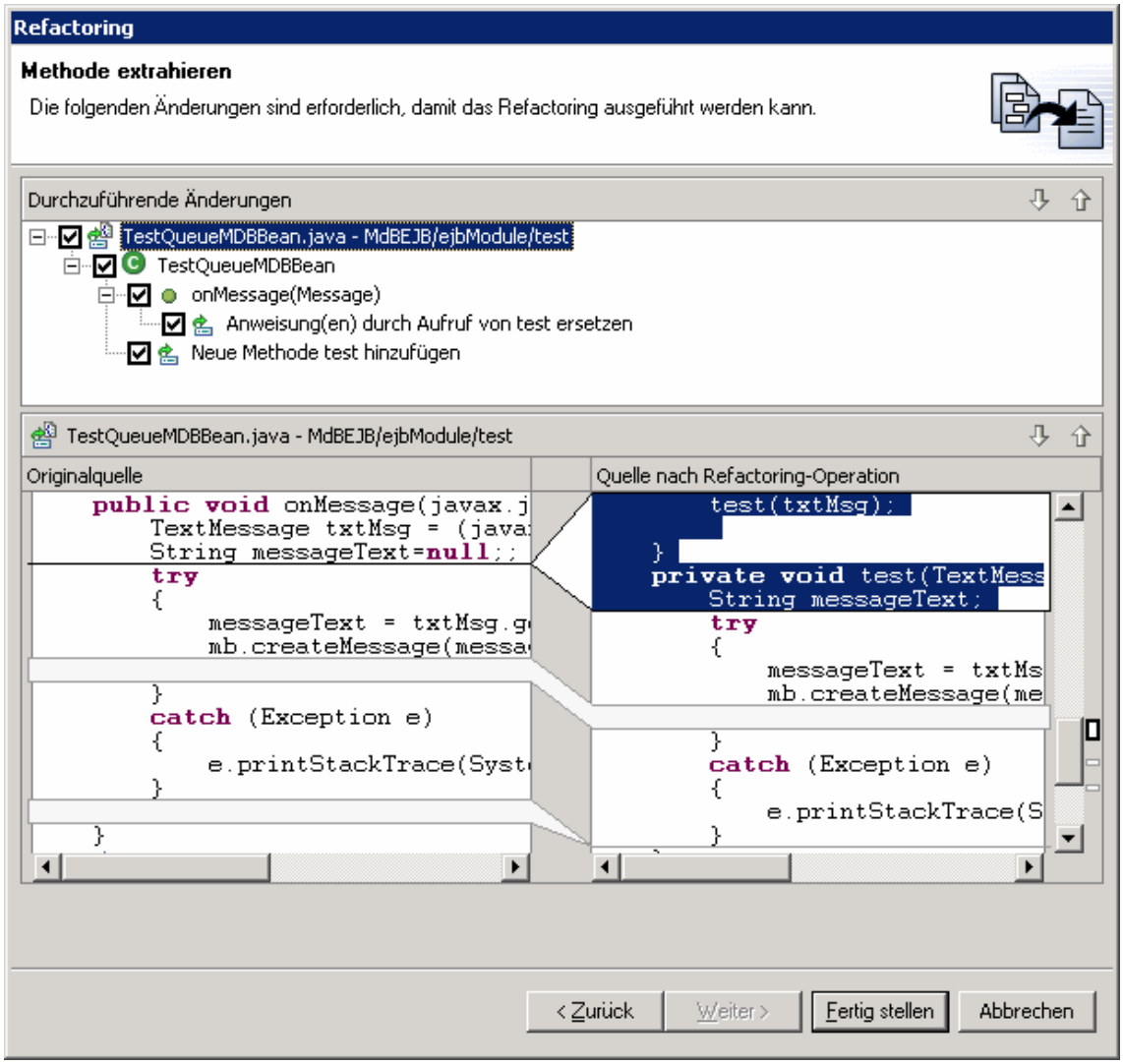

Abb. 1.14: Extrahieren einer Methode

Um ein Stück Quellcode in eine neue Methode zu extrahieren, wird der entsprechende Text markiert, die Auswahl mit der rechten Maustaste angeklickt und „Refaktor→Methode extrahieren“ gewählt. Sie werden daraufhin nach dem Namen für die neue Methode gefragt und können angeben, ob in diesem Teilstück auftretende Exceptions in die Signatur der neuen Methode aufgenommen werden sollen. Der nächste Dialog zeigt wie sich die Refaktorisierung auf die Datei auswirkt. Es werden die neu eingefügten Stellen markiert und gezeigt, wo die alten Passagen wieder zu finden sind. Mit einem Klick auf „Fertig stellen“ werden die Änderungen in die Datei übernommen, werden aber erst mit dem normalen Abspeichern der Datei endgültig festgeschrieben.

Eine weitere Möglichkeit des Refactoring, zur Verbessung der Lesbarkeit des Quellcodes, ist die Umbenennung von Methoden. Dabei erlaubt es WSAD auch, Methoden zu verändern, die von anderen Klassen benutzt werden.

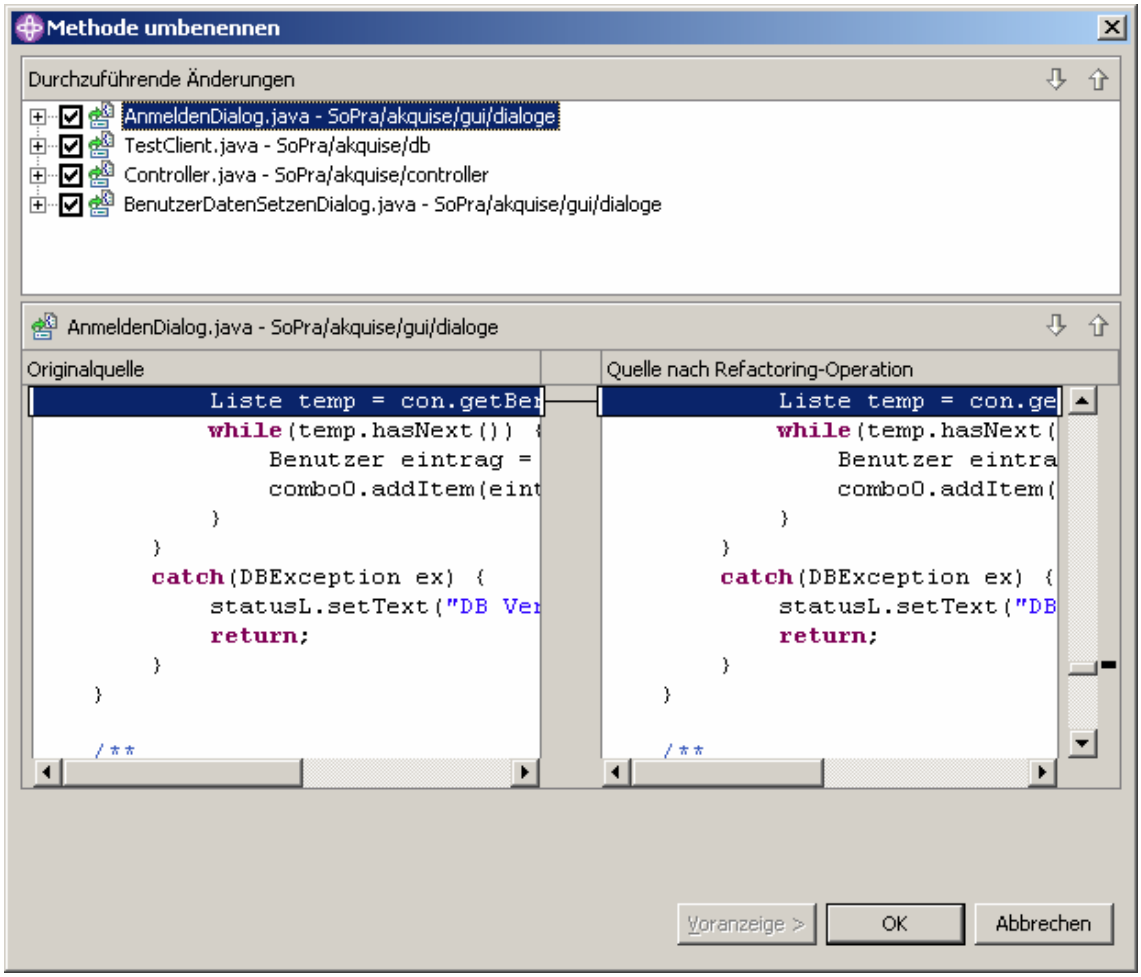

Abb. 1.15: Umbenennen einer Public-Methode

Um eine Methode umzubenennen, kann ihr Namen direkt in der Quelltextansicht, oder in der Gliederungsansicht die komplette Methode markiert werden. Rechtsklick, dann auf „Refaktor→Umbenennen“, schon wird ein Assistent aktiv und fragt wie die Methode nach dem Umbenennen heißen soll. Mit dem Klick auf Weiter werden sämtliche Fundstellen im Arbeitsbereich angezeigt und können komplett automatisch geändert, oder von Hand noch einmal überprüft werden.

Die Umbenennen-Funktion steht nicht nur für Methoden, sondern auch für ganz normale Variablen zur Verfügung.

Der WSAD bietet noch eine ganze Reihe von weiteren Funktionen für das Refactoring von Quellcode. Es lohnt sich, sich diese, gerade bei größeren Projekten, einmal genauer anzusehen, da meistens eine enorme Arbeitserleichterung erreicht werden kann, wenn stupide Schritte, wie das Schreiben von Gettern und Settern oder Ähnliches von einem intelligenten Tool übernommen werden können.

1.4.1.3 Anbindung an ein CVS Repository

Die Funktion zur Anbindung des WSAD an ein CVS Repository kann gerade für Studenten, die gemeinsam ein Projekt verwirklichen möchten, eine große Hilfestellung leisten.

Die Bezeichnung Concurrent Versions System (CVS) steht für ein Softwareprogramm zur Verwaltung von Dateien. Meist werden mit seiner Hilfe Quelltexte von Programmen verwaltet, da CVS hierbei seinen vollen Funktionsumfang zur Geltung bringen kann. CVS verwaltet die Quelltexte in einem zentralen Repository (≈ Speicher), so dass auch mehrere Programmierer gleichzeitig arbeiten können. Ein solches Repository ist gerade für Programmierer-Teams, die weit voneinander entfernt arbeiten essenziell wichtig. Daher ist CVS vor allem in der Open-Source-Welt sehr verbreitet.

Um von der Anbindung an ein CVS-Reposiory profitieren zu können, muss als erstes ein Server dafür eingerichtet werden. Wichtig dabei ist, dass der Server ständig über das Internet erreichbar ist und für ihn ein Fully Qualified Domain Name (FQDN) registriert wurde. Da nur wenige Privatpersonen eine feste IP-Adresse besitzen, bietet es sich an einen Dienst wie http://www.dyndns.org heranzuziehen, um den Domain Namen jeweils auf die aktuelle IP-Adresse zu setzen. Die Konfiguration wird auf der Webseite recht detailliert beschrieben.

Als nächstes muss ein CVS-Server aufgesetzt werden. Unter Linux/Unix ist CVS meist standardmäßig installiert, oder lässt sich recht einfach über die Paketverwaltung der Distribution nachinstallieren. Um den CVS-Server zu aktivieren, bedarf es folgenden Eintrags in der inetd.conf:

```
cvspserver stream tcp nowait root /usr/sbin/tcpd /usr/bin/cvs
-f --allow-root=/home/cvsroot pserver
```

Abschließend muss das Repository noch initialisiert warden. Dazu muss das Verzeichnis /home/cvsroot angelegt und

```
cvs -d /home/cvsroot init
```

ausgeführt werden. Der Server steht nun im Internet zur Verfügung. Die Verwaltung der Benutzer/Passwörter geschieht direkt über das Betriebssystem. Zugriffsrechte am Repository können ebenfalls mit den Standard Unix-Kommandos (chmod/chown) festgelegt werden.

Unter Windows gestaltet sich die Installation eines solchen CVS-Servers etwas komplizierter, es soll daher an dieser Stelle darauf verzichtet werden. Weitere Anleitungen und Downloads unter http://www.cvsnt.org.

Eine weitere Möglichkeit, ohne größeren Aufwand einen CVS-Server gestellt zu bekommen ist die Anmeldung des Projekts bei Sourceforge.net. Dieser Internetdienst stellt Entwicklern Server zur weltweiten Präsentation ihrer Projekte zur Verfügung, unter anderem auch CVS-Server. Bedingung ist allerdings, dass das Programm OpenSource entwickelt wird.

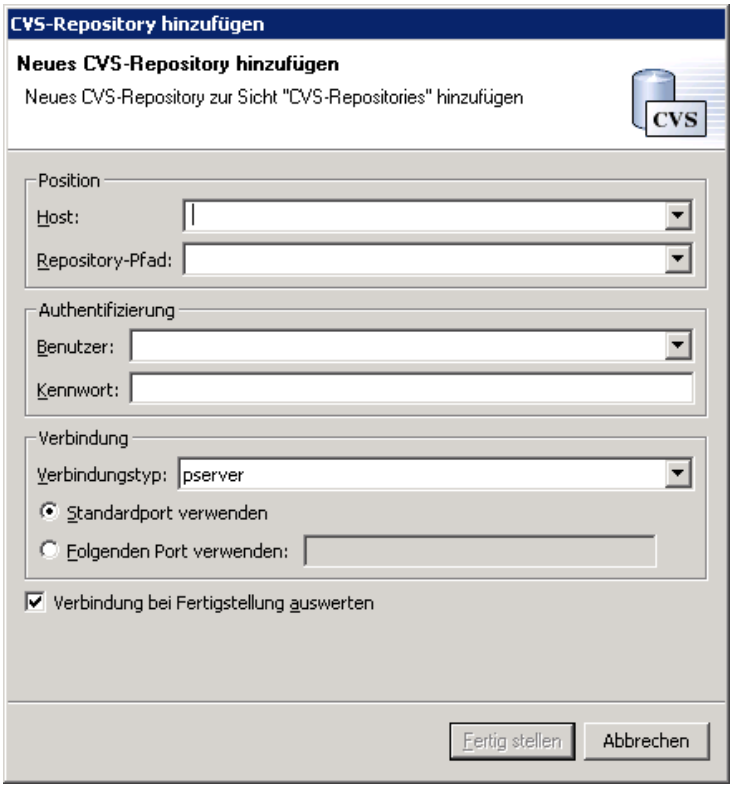

Abb. 1.16: Dialog zum Hinzufügen eines neuen CVS-

Um den WebSphere Application Developer in Verbindung mit CVS zu nutzen, muss die Verbindung zum Server in der CVS-Ansicht erstellt werden. Dafür wird die Perspektive „CVS Repository - Durchsuchen“ benötigt. Über „Neu→Repositoryposition“ kann diese Verbindung erstellt werden. Unter Host muss der für den CVS-Server registrierte Domain-Name, bei Repository-Pfad der entsprechende Ordner des Dateisystems angegeben werden, in unserem Fall /home/cvsroot. Sollte ein Fehler ausgegeben werden, muss überprüft werden, ob der Dienst (inetd) gestartet wurde und ob evtl. eine Firewall den Zugriff auf den Port (TCP Port 2401) aus dem Internet verweigert. Ein weiterer Fehler könnte bei dyndns.org liegen. Mit Hilfe von nslookup [meinHost].dyndns.org kann überprüft werden, welche IP-Adresse diesem Namen gerade zugeordnet ist. Sollte das nicht die aktuelle Adresse des Servers sein, muss eine Aktualisierung vorgenommen werden.

Nachdem die Verbindung eingerichtet ist, können die Projekte auf dem Server publiziert werden. Dazu wird in die J2EE-Perspektive (J2EE-Navigator) gewechselt. Mit einem Rechtsklick auf das Projekt→Team→„Projekt gemeinsam benutzen“ kann das Projekt in den CVS-Server geschrieben werden. Hierzu kann die soeben erstellte CVS-Verbindung genutzt werden. Bei „Modulnamen eingeben“ sollte nicht der Standard befolgt werden, da sonst für jedes der Projekte ein eigenes Modul benötigt wird. Um dieses Problem zu umgehen, kann dem Projektnamen ein „[modul]/“ vorangestellt

werden. Auf diese Weise kann die komplette Anwendung in einem gemeinsamen Modul untergebracht werden. Im unteren Bereich der Arbeitsfläche wird nun ein Synchronisieren-Fenster angezeigt, in dem das gewählte Projekt angezeigt wird. Mit einem Rechtsklick auf die Projektwurzel und „Zur Projektsteuerung hinzufügen“ werden die gewählten Elemente markiert, damit sie bei der nächsten Synchronisation hoch geladen werden können. Über „Festschreiben“ können die Dateien auch sofort auf dem Server verfügbar gemacht werden. Diese Prozedur sollte nun mit jedem einzelnen Projekt wiederholt werden, so ist garantiert, dass alle Teammitglieder den kompletten Stand der Entwicklungen verfolgen und sich in allen Bereichen beteiligen können.

Sind die Daten auf dem Server festgeschrieben, können andere Clients ihrerseits das Repository herunter laden. In der CVS-Perspektive kann das entsprechende Modul gewählt werden. Hier sollten alle Projekte als Ordner angezeigt werden.

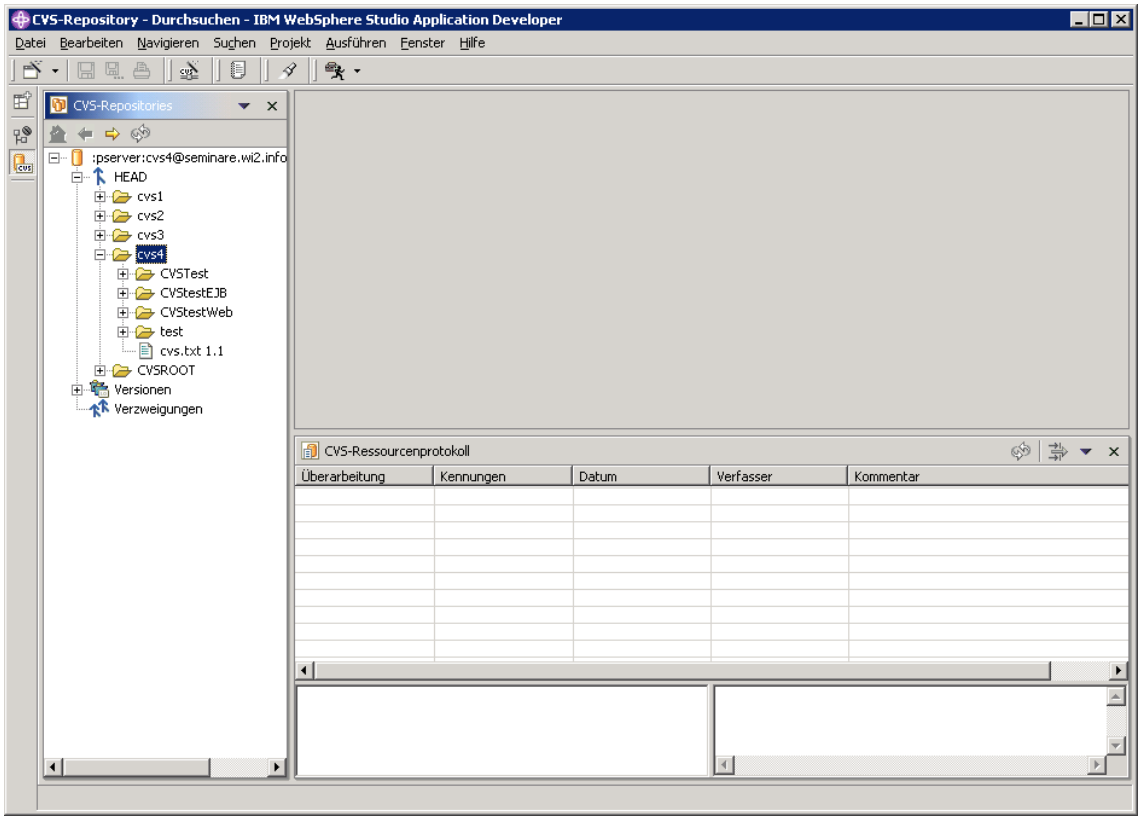

Abb. 1.17: Die CVS-Perspektive des WSAD

Über einen Rechtsklick auf den Projektordner→„Als Projekt entnehmen“ wird der Ordner vom CVS-Server herunter geladen und die Versionskontrolle für das entsprechende Projekt aktiviert.

Änderungen an den Quelltexten, oder den Konfigurationen können ab jetzt manuell in das CVS eingespielt werden. Dies geschieht mit Hilfe der Funktion

„Team→Festschreiben“. Über Synchronisieren besteht eine noch feinere Kontrolle darüber, welche Dateien in das Repository geschrieben werden wollen. Änderungen, die andere am Projekt durchgeführt haben, können mit Hilfe von „Team→Aktualisieren“ auf den lokalen Rechner synchronisiert werden. Treten hierbei Konflikte auf, werden diese automatisch in einem Fenster angezeigt und der Entwickler hat die Möglichkeit die Dateien manuell zusammenzuführen.

CVS hat weit mehr zu bieten als diese wenigen hier vorgestellten Features. So können Codezweige (Branches) ausgegliedert werden, um z.B. eine neue Architektur zu implementieren, während die anderen Entwickler ganz normal weiterarbeiten können. Erst wenn die neue Architektur dann fertig gestellt und getestet ist, werden die von den anderen Programmierern inzwischen eingespielten Veränderungen auch auf den neuen Zweig angewendet (Merge).

CVS bietet für Teams fast alle Funktionen, die benötigt werden, um rund um die Uhr in verschiedensten Teilen der Erde an Quelltexten zu arbeiten. Dank des zentralen Servers sind die Daten jederzeit in einem konsistenten Zustand.

1.4.1.4 Integrierte Testumgebung

Für die Entwicklung einer Applikation sind das Testen und das Debugging (Finden von Fehlern während dem Ablauf einer Applikation) sehr wichtig. Anwendungen, die auf J2EE basieren, lassen sich aber auch nur in einem J2EE-kompatiblen Application Server betreiben. Ein solcher Server ist aber eher für den produktiven Einsatz, als für die Entwicklung gedacht und ist deshalb meist sehr unhandlich und kompliziert zu konfigurieren.

Um es Entwicklern auf einfache Weise zu ermöglichen ihre Anwendungen zu testen, wurde in den WSAD ein im Funktionsumfang reduzierter, aber voll funktionsfähiger WebSphere Application Server eingebaut. Der Server wird bei Bedarf gestartet und verhält sich völlig analog dem originalen WebSphere. Die Prozedur des Deployment wurde allerdings gleich in den Aufruf des Servers integriert, so dass direkt nach dem Start die Anwendung fertig konfiguriert zur Verfügung steht. Wie der WebSphere Application Server selbst muss jedoch auch die Testumgebung vor dem ersten Start konfiguriert werden und muss die entsprechenden Ressourcen bereitstellen.

Die Anlage einer solchen Testumgebung erfolgt über die J2EE-Perspektive druch einem Rechtsklick auf Server „Neu→Server und Serverkonfiguration“. Es startet ein Assistent, der nach dem Namen und einer Portnummer fragt, die für den http-Transport verwendet werden soll. Dieser Zahl kommt besondere Bedeutung zu, da kein anderer Dienst diese

Portnummer bereits verwenden darf. Sollte also auf dem Entwicklungsrechner auch ein voller WebSphere Application Server installiert sein, kommt es bei Verwendung des Standard-Ports zu einem Konflikt und die Testumgebung wird nicht starten.

Die weiteren Einstellungen geschehen über einen speziellen Editor für Serverkonfigurationen. Durch einen Doppelklick auf den neu angelegten Server können seine Einstellungen bearbeitet werden. Die Registerkarte „Konfiguration“ enthält die ersten wichtigen Optionen: Hier kann festgelegt werden, welche Anwendungen beim Start der Umgebung zur Verfügung stehen sollen. So lassen sich der Universelle Testclient und die Verwaltungskonsole für WebSphere aktivieren. Letzteres ist vor allem wichtig, wenn die Einstellungen von der Testumgebung auf den Produktivserver übertragen werden sollen, und die Optionen, die WSAD, hinter dem Editor versteckt, noch unbekannt sind. Außerdem stehen mit der Verwaltungskonsole ungleich mehr Konfigurationsoptionen zur Verfügung, so dass eine Einstellung, die für die Applikation benötigt wird, evtl. nur hier verfügbar ist.

In den „Umgebungsoptionen“ lassen sich, ähnlich dem Classpath in Java, zusätzliche benötigte Bibliotheken einbinden. Bei Verwendung z.B. des Bean Skripting Framework für Rexx (siehe 3.2 Seite 40) innerhalb einer Testumgebung, müssen hier die entsprechenden JAR-Dateien eingebunden werden.

Auf der Registerkarte „Web“ lassen sich die MIME-Typen konfigurieren. Diese Zuordnung zwischen einer Dateiendung und einem bestimmten Textstring, etwa „text/html“ für HTML-Dateien legt das Verhalten fest, mit dem Browser auf eine Datei dieses Typs reagieren. So wird z.B. für Dokumente des Typs „video/mpeg“ ein geeigneter Movie-Player gestartet. Im Normalfall müssen an der vorgegebenen Tabelle keine Anpassungen vorgenommen werden.

Ebenso wie im „echten“ Application Server müssen Ressourcen, auf welche die Enterprise-Anwendung zugreifen können soll, erst auf dem Server konfiguriert werden. Für die Konfiguration der Datenquellen dient die nächste Registerkarte. Mit Cloudscape und DB2 stehen hier die von IBM präferierten Datenbanken schon konfiguriert zur Verfügung, andere lassen sich aber, wie im echten Server auch ohne größere Probleme einbinden. Für die Datenbanken müssen, um eine Ressource zur Verfügung zu stellen, auch Datenquellen definiert werden. Für diese können wiederum weitere Eigenschaften festgelegt werden. Besonders wichtig ist hierbei der Name der Datenquelle (DatabaseName), der festlegt auf welche Datenbank innerhalb des Systems mit Hilfe dieser Verbindung zugegriffen werden soll.

Auf der Registerkarte „Ports“ müssen für jeden Dienst, den die Testumgebung zur Verfügung stellen soll, die TCP-Ports konfguriert werden, die benutzt werden sollen. Die unter „Zelleneinstellung“ vergebenen Hostaliasnamen legen lediglich fest, auf welche Namen (z.B. meinPC.uni-augsburg.de) der integrierte Webserver reagieren soll. Soll die Verwaltungskonsole gestartet werden, muss in dem entsprechenden Feld noch der Port dafür konfiguriert werden. Der Abschnitt Servereinstellungen enthält nun die Portzuordnungen der eigentlichen Instanz. Wird auf dem lokalen Rechner auch noch ein WAS betrieben, müssen die Einstellungen entsprechend angepasst werden, damit diese nicht mit dem Server in Konflikt stehen.

Die nächste betrachtete Registerkarte ist „Sicherheit“. Hier kann eingestellt werden, dass die Umgebung im Kontext eines anderen Benutzers ausgeführt wird. Dadurch kann die Sicherheit erhöht werden, da die Umgebung nicht alle Rechte des Entwicklers benötigt. Mit Hilfe der JAAS-Einträge lassen sich Kennwörter für bestimmte Anwendungen zur Verfügung stellen. So müssen z.B. Passwörter, die zum Zugriff auf eine Datenbank benötigt werden, hier konfiguriert werden.

Für die meisten Zwecke dürfte dieser kurze Einstieg in die Konfiguration der Testumgebung ausreichend sein. Für weitergehende Hinweise zur Konfiguration, siehe [Sadt2003].

1.4.2 Entwicklung einer Applikation

Zur Entwicklung einer Applikation gibt es einige „best-practices“, über die bereits einige Bücher geschrieben wurden. An dieser Stelle soll nur kurz auf die Reihenfolge in der Implementierung eingegangen werden. Weitergehende (sehr aufschlussreiche) Informationen siehe [WNA+2003].

Es sollte als erstes das Klassendiagramm der Anwendung umgesetzt werden, also der Daten-Hintergrund für die EJBs in Form von Entity-Beans erstellt werden. Anschließend können die Relationen zwischen den Beans erstellt werden. Damit ist bereits das Grundgerüst der Anwendung erstellt und es kann versucht werden, mit Hilfe des Universellen Testclients einige Beans zu erzeugen. Nachdem dies geschehen ist, kann mit der Implementierung der eigentlichen Programmlogik begonnen werden.

Erst nachdem die Anwendungslogik soweit feststeht und nicht mehr verändert werden wird, macht es Sinn die Implementierung des GUI zu beginnen. Bei J2EE-Anwendungen wird das in den meisten Fällen eine Weboberfläche mit JSPs und/oder Servlets sein.

Zuletzt sollte festgelegt werden, wie externe Clients über Java Anwendungen oder Webservices angebunden werden.

Natürlich kann von dieser Reihenfolge abgewichen werden, wenn die Anwendung gut spezifiziert ist. Insbesondere, wenn ein Team in die Entwicklung involviert ist, macht es Sinn, vorab Prototypen der GUI zu erstellen und die Schnittstellen in einem frühen Stadium der Entwicklung festzulegen.

1.4.3 Vorbereitung des Deployments

Bevor eine Anwendung auf einem Application-Server laufen kann, muss zuerst sie selbst und alle mit ihr verbundenen Dateien, wie Grafiken für die Web-Anzeige, zusätzlich benötigte Bibliotheken und sonstige Hilfsdateien zusammen in ein Enterprise Java Archiv (EAR) gepackt werden. Der Aufbau dieses Archivs wurde standardisiert, damit der Inhalt von allen auf J2EE-basierenden Umgebungen gleich interpretiert wird. Das Format des Archivs selbst ist ZIP, es kann also mit einem herkömmlichen Packprogramm erstellt oder verändert werden.

Innerhalb einer EAR-Datei liegen weitere Java-Archive, die je nach Inhalt anders aufgebaut sind. Es gibt:

- Web-Achive
  Diese enthalten alle Dateien, die in den Webcontainer des Application Servers deployed werden sollen. Hier finden sich also neben JSPs auch die Klassen für Servlets und evtl. Grafikdateien oder statische HTML-Seiten.

- EJB-Archive
  Wie aus dem Namen des Archivs hervorgeht, werden hier alle Klassendateien abgespeichert, die in den EJB-Container des Servers wandern sollen.

- Client-Archive
  Client-Anwendungen, die auf den Server zugreifen sollen, können in den Application Client Container deployed werden.

Jedes dieser Archive enthält einen speziell aufgebauten Deployment-Descriptor, der auch die genaue Funktion der Datei kennzeichnet. In dieser Datei ist vermerkt, wie der Application Server mit der Anwendung verfahren soll. So wird hier z.B. festgelegt, welche Datenbank für die Persistenz von Entity-Beans verwendet werden soll, oder wie das Basis-Verzeichnis für die Web-Anwendungen lautet. Ein weiterer Standard innerhalb jedes Java-Archivs ist die Datei MANIFEST.MF, die sich im

Unterverzeichnis META-INF befinden muss. Hier können zusätzliche Attribute der Anwendung festgelegt werden, wie der Name oder die Signatur, im Falle einer versiegelten Anwendung.

Der Aufbau dieses Archivs tritt auch im WSAD zu Tage. Hier müssen für jede der einzelnen Java Archive eigene Projekte erzeugt werden. Der Typ des Projekts im WSAD legt den späteren Typ des Java Archivs fest. Die meisten der Strukturen werden von WSAD selbst verwaltet, nur selten sind Eingriffe direkt am Quelltext der Deployment-Deskriptoren nötig. Sollte diese XML-Dateien bearbeit werden müssen, so stellt WSAD auf der letzten Registerkarte („Quelle“) des Deployment-Deskriptors den Quelltext der Datei zum Bearbeiten dar.

1.5 WebSphere Application Server

Der Application Server ist das Herzstück der WebSphere-Welt. Er stellt für die installierten Anwendungen eine Laufzeitumgebung zur Verfügung. So muss sich ein Entwickler nicht darum kümmern, wie die Daten abgespeichert werden müssen, oder wie die Kommunikation zwischen den Clients und der Serveranwendung funktioniert. Stattdessen kann er sich auf die Entwicklung der eigenen Applikation konzentrieren. Dank des J2EE-Standards können Anwendungen, die für einen J2EE-konformen Server geschrieben wurden, rein theoretisch ohne Anpassung oder Neukompilierung auf einem anderen Application-Server eingesetzt werden. Abb. 1.18 zeigt den grundsätzlichen Aufbau eines solchen Servers. Dank der verschiedenen Schnittstellen zu Backend-Systemen können auch andere Applikationen als Dienstgeber eingebunden werden.

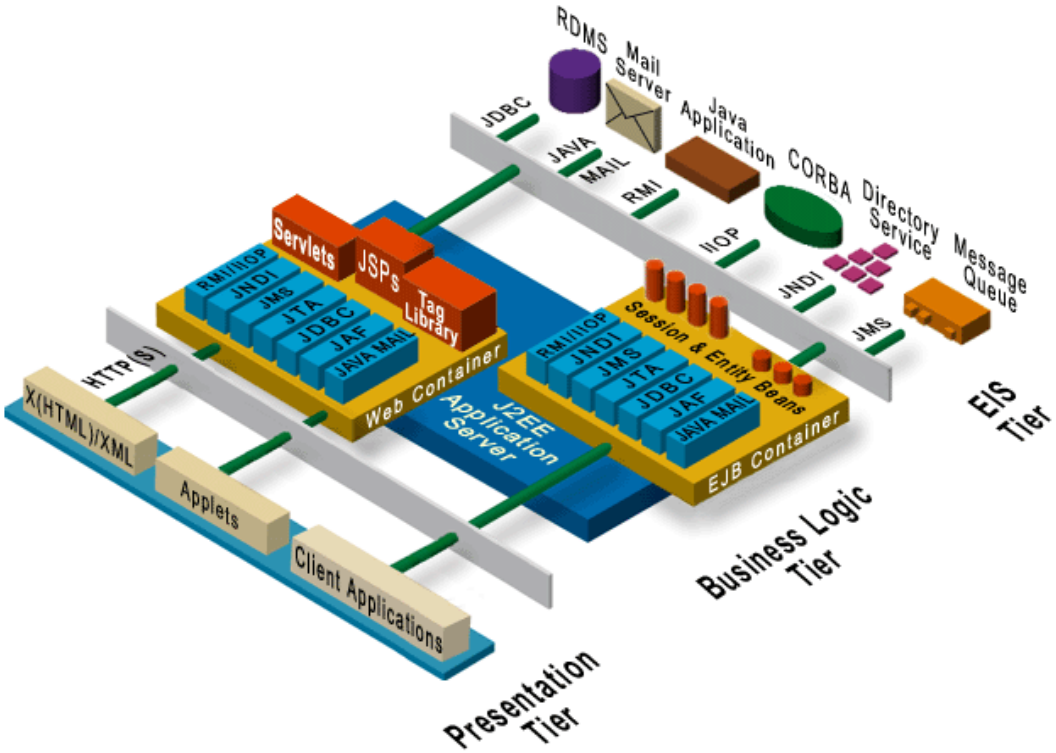

Abb. 1.18: Generelle Architektur eines J2EE konformen Application Servers (Quelle: [Info2002])

J2EE-Server werden aufgrund dieses vielfältigen Schnittstellenangebots häufig für die Enterprise Application Integration (EAI) eingesetzt. Dabei werden die in den Unternehmen vorhandenen Altsysteme als Dienstgeber in den Application-Server eingebunden, um darauf basierend eine neue IT-Infrastruktur zu schaffen.

1.5.1 Aufgaben des Application-Servers

Application Server bieten vielfältige Dienste für ihre Anwendungen an. Die wichtigesten sind:

- Transaktionssicherheit
  Der Entwickler muss sich nicht um die Transaktionssicherheit seiner Komponenten kümmern, da diese Aufgabe vom Container für ihn erledigt wird.

- Sicherheit
  Applikationen laufen innerhalb einer gesicherten Umgebung ab. Anmeldungen und Zugriffsrechte werden vom Application Server verwaltet und stehen jeder Anwendung zur Prüfung der Autorisierung zur Verfügung.

- Persistenzmanagement
  Wie bereits erwähnt, werden Entity-Beans vom Container verwaltet, ohne dass sich der Entwickler darum kümmern muss.

- Trennung der Applikationen von den Infrastrukturdiensten
  Anwendungen müssen nicht länger speziell für eine Infrastruktur angepasst werden. Generelle Schnittstellen erlauben es die im Hintergrund verwendeten Anwendungen über Schnittstellen anzusprechen und dementsprechend bei Bedarf auszutauschen.

Die Unterstützung des Komponentenbasierten Ansatzes mit Hilfe der Enterprise-Java-Beans ermöglicht weiterhin ein hohes Maß an Widerverwendbarkeit.

Es wurde nun schon häufiger der Begriff des „Containers“ im Zusammenhang mit J2EE verwendet. Eine der Hauptaufgaben des Application Servers besteht darin, solche Container zur Verfügung zu stellen. Die J2EE-Spezifikation ([SunJ2EE2003]) sieht vier verschiedene Sorten von Containern vor: application client container, applet container, web component container und enterprise bean container. Jeder wird für einen bestimmten Typ Anwendung verwendet. Abb. 1.19 zeigt die Container des Java Application Servers mit ihren gängigen Schnittstellen:

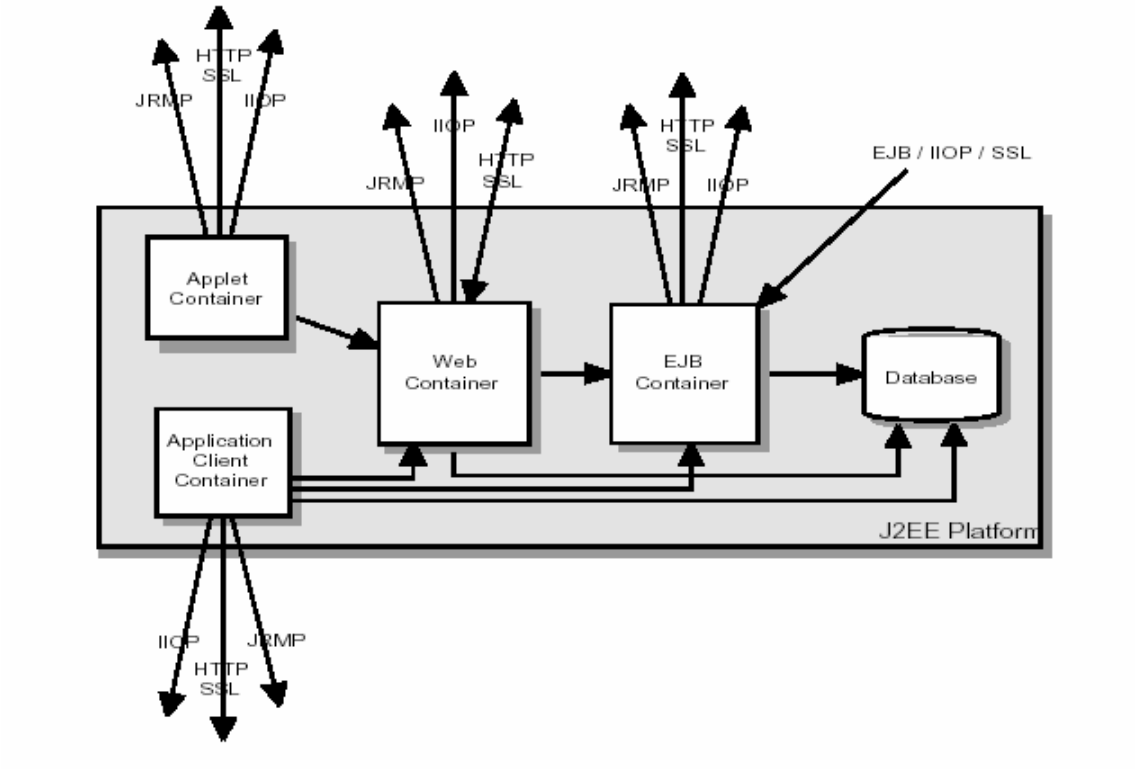

Abb. 1.19: Die Container eines J2EE Application Servers (Quelle: [SunJ2EE2003])

Ein weiterer wichtiger Dienst des Application Servers, der schon häufiger in diesem Dokument verwendet wurde, ist das JNDI (Java Naming and Directory Interface). Jede Anwendung und jede Ressource wird im JNDI vermerkt, so dass von überall auf sie zugegriffen werden kann.

1.5.2 Deployment einer Applikation

Das Deployment einer Anwendung soll nun anhand des JMS-Beispiels aus 3.3 nachvollzogen werden. Um eine Anwendung zu installieren wird die Adminstrationskonsole geöffnet. Dort ist im linken Menü die Option „Anwendungen→Enterprise Anwendung installieren“ verfügbar.

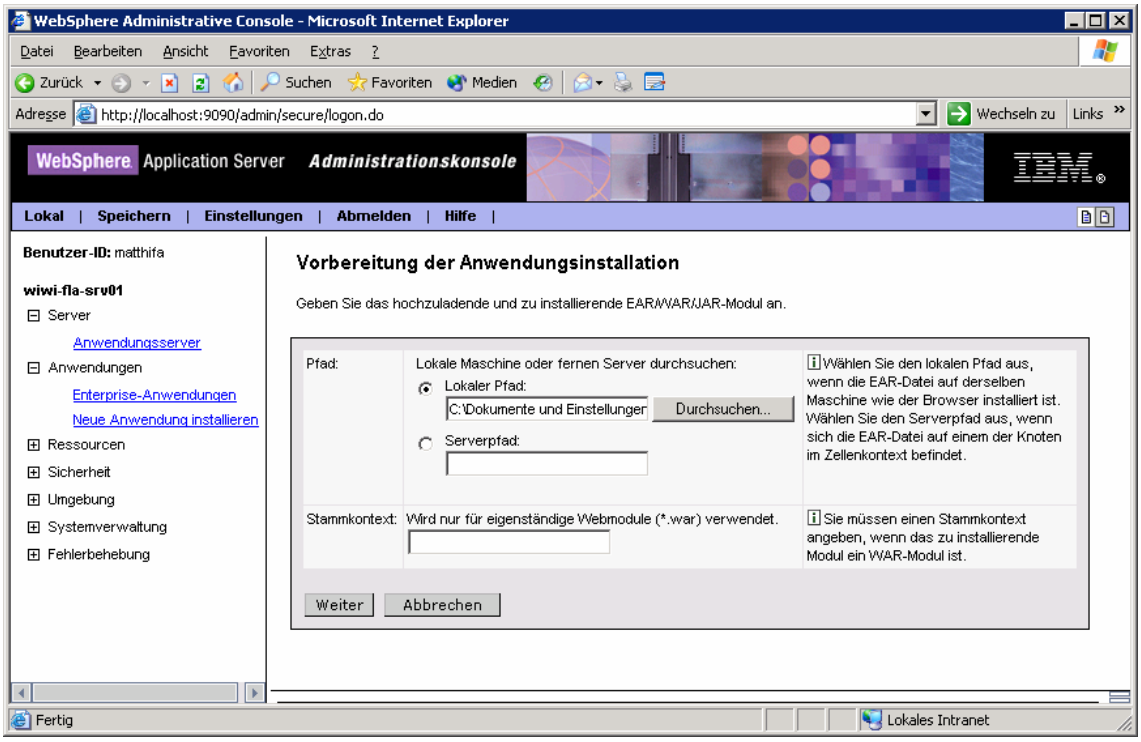

Abb. 1.20: Installation einer neuen Anwendung auf dem WAS

Die nächste Konfigurationsseite enthält generelle Einstellungen, die die gesamte Anwendung betreffen. Im Regelfall müssen hier jedoch keine Anpassungen vorgenommen werden.

Die weiteren Schritte lassen sich in beliebiger Reihenfolge durchlaufen und variieren in ihrer Anzahl, je nachdem, welche Informationen für das Deployment der Anwendung benötigt werden. Sind im WSAD die Werte schon konfiguriert, werden normalerweise wenige weitere Angaben benötigt.

Zu einer dieser Angaben gehört die Datenquelle, auf die zur Speicherung zurückgegriffen werden soll. Mit Hilfe von „Mehrere Zuordnungen anwenden“ kann eine der konfigurierten Datenquellen, die bereits auf dem Server besteht, ausgewählt werden und dann ggf. für alle Entity Beans auf einmal festgelegt werden.

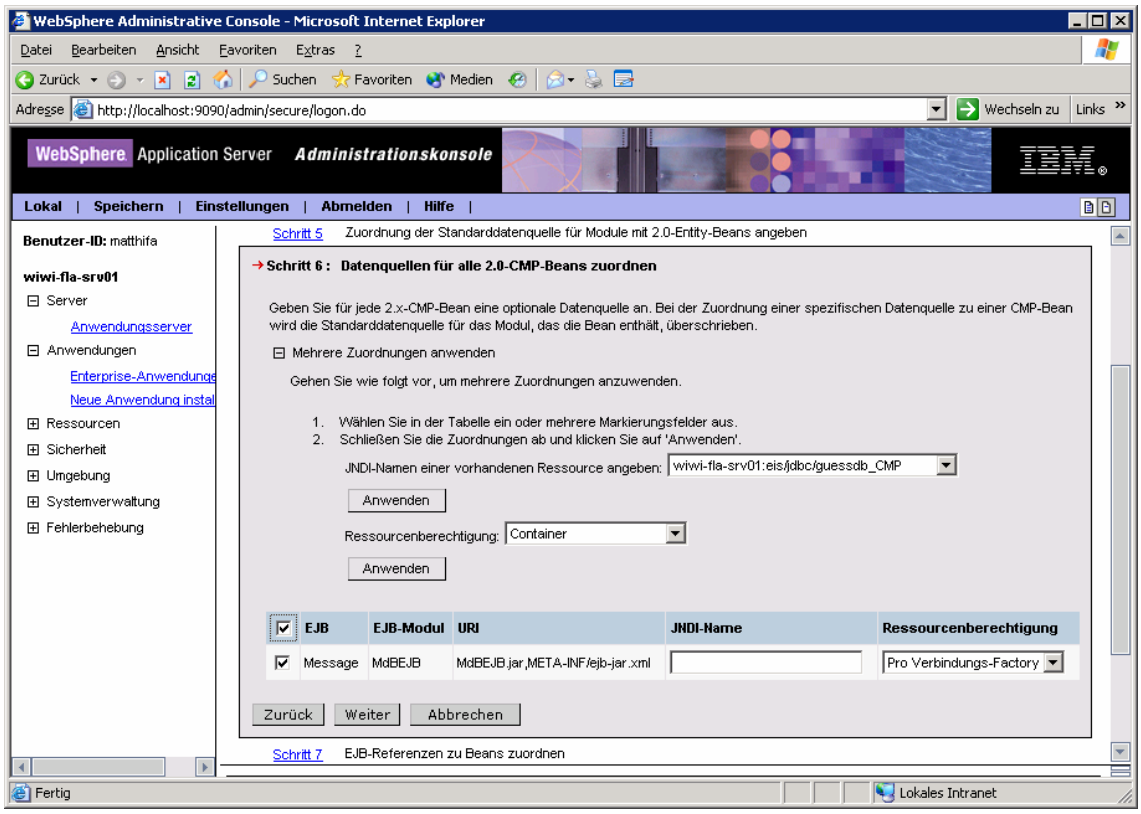

Abb. 1.21: Auswahl der Datenquelle für die Entity Beans mit CMP

Damit die während der Entwicklung verwendeten JNDI-Namen für die Beans später nicht zu Namenskonflikten führen, kann während des Deployment den Name jeder Bean gezielt festgelegt werden. Allerdings muss dann auch die Referenz, die auf diese Beans verweist angepasst werden.

Nach Abschluss des Deployments muss der die neue Konfiguration für die Anwendung erst auf dem Server gespeichert werden. Erst nachdem das geschehen ist, kann die Anwendung gestartet werden. Es sollte vor allem beim Start auf die Fehler oder Warnungen, die evtl. in der Konsole angezeigt werden, geachtet werden. Konfigurationsfehler, die relativ leicht passieren, sind nicht zur Verfügung stehende Ressourcen. So muss die Datenbankverbindung genau mit dem für die EJB konfigurierten JNDI-Namen erstellt werden, da sonst der Container die Beans nicht erstellen kann.

2 Verifizierung der Diplomarbeit von Petra Lehner

Die bereits öfter erwähnte Diplomarbeit von Frau Lehner war der eigentliche Anstoß der Autoren für die Wahl des Themas dieser Arbeit. Die Autoren hatten die Gelegenheit (und das Privileg) die Arbeit von Frau Lehner vor deren offiziellen Abgabe für ihre Zwecke heranziehen zu können. Es war nicht das Ziel, diese Arbeit zu bewerten. Hauptsächlich ging es darum, nachzuvollziehen, ob die Schritte, die Frau Lehner beschreibt, leicht verständlich sind und ob die dort vorgestellten Beispiele einfach nachzuvollziehbar sind. In den nachfolgenden Kapiteln geben die Autoren eine Zusammenfassung, soweit sie ihnen wichtig für das Verständnis bei der Arbeit mit den dort besprochenen Programmteilen und Beispielen erscheint.

2.1 Allgemeines

Frau Lehner hat die Installation auf dem Betriebssystem Windos XP Home beschrieben, was den Autoren als nicht ideal erscheint, da der echte Einsatz sehr wahrscheinlich auf Server-Betriebssystemen stattfinden wird. Ein kurzer Verweis auf Unterschiede bei Serverbetriebssystemen wäre hilfreich (vor allem im Bereich Benutzerverwaltung / Anbindung an Domänen / Dateirechten wegen Benutzerzugriffen) Die Autoren haben die Anleitung mit Windows 2000 Professional, Windows 2000 Server, Windows XP Professional und Windows 2003 Server getestet, wobei nur die Installation auf allen Systemen vollzogen wurde. Die hauptsächliche Testarbeit fand auf einem Windows 2003 Server statt.

In der Diplomarbeit wurde darauf verzichtet, den kompletten Life-Cycle einer J2EE-Applikation auf IBM-Produkten abzubilden. Hier wäre eine Idee, die von IBM erworbene Firma Rational zu erwähnen. Diese deckt mit dem Produkt XDE, das eine Integration in den Application-Developer besitzt, die ersten beiden Bereiche Design und Process Engeneering in der Grafik auf Seite 38 der Diplomarbeit ([Lehn2003]) ab.

2.2 Installation der Applikationen

Auf einem W2K3-Server ließ sich die der Diplomarbeit beiliegende Version der WebSphere Application Servers nicht installieren, erst bei einer speziell dafür erhältlichen Version war dies möglich, wobei hier der IBM-HTTP-Server nicht in der Installation enthalten ist.

Die Installation auf W2K und Windows XP führte zu keinen weiteren Problemen.

Generell war es aber auf allen Systemen so, dass die httpd.conf – die Konfigurationsdatei für den IBM http Server – nicht mit den notwendigen Einträgen zum Laden des Moduls für das WebSphere PlugIn bei der Installation erzeugt wurde. Erst nach weiterer Recherche haben die Autoren die entsprechenden Einträge gefunden und manuell hinzugefügt. Sollte ein Automatismus zur Erzeugung des Eintrages existieren, wäre ein Verweis darauf sinnvoll, ansonsten die Angabe der manuellen Prozedur.

Die Installation des WSAD und der DB2-Datenbank ließen sich ohne Probleme nachvollziehen.

2.3 Testen der Beispiele

Die Beispiele innerhalb der Diplomarbeit ließen sich ohne größere Probleme nachvollziehen. So war die Erstellung der MyVisitor-Anwendung durchaus sehr lehrreich, allerdings gestaltet sich das Abtippen der Quelltexte bisweilen etwas mühsam.

Leider wurde im letzten Beispiel nicht die Verwendung der eigens eingerichteten Relation zwischen den EJBs gezeigt. Das wäre auf der High-Score-Tabelle problemlos möglich gewesen.

Insgesamt verdeutlichen die Beispiele die gezeigten Sachverhalte sehr gut, nur manchmal ist es wünschenswert, die einzelnen Optionen würden in ihrer Funktionsweise genauer beschrieben. Für einen ersten Einstieg ist das aber wohl zu umfangreich und auch zu unübersichtlich.

3 Erweiterungen

Im folgenden Kapitel werden anhand einiger Aufgabenstellungen typische Probleme und Lösungsmöglichkeiten innerhalb von WebSphere dargestellt. Im ersten Beispiel widmen sich die Autoren einer Portierung einer Enterprise-Anwendung von Suns J2EE Referenzimplementierung auf WebSphere. Was theoretisch ohne Probleme möglich sein sollte, bereitet in der Praxis durchaus einiges Kopfzerbrechen. Folgend wird kurz das Bean Skripting Framework vorgestellt und gezeigt, wie eine weitere (Programmier-) Sprache integriert werden kann. Zuletzt wird noch kurz in die Thematik des Java Messaging eingeführt.

3.1 Paycircle

Aufgabenstellung: „Portieren Sie die Referenzimplementierung des PayCircle-Standards von der J2EE von Sun auf WebSphere."

3.1.1 Was ist PayCircle?

PayCircle ist eine non-profit Organisation, die unabhängig von den Banken und Kreditkartenanbietern ein System zum elektronischen bezahlen entwickeln will.

Mitglieder von PayCircle sind unter anderem HP, Siemens, Oracle, CSG-Systems, Sun, die Universität Augsburg sowie viele weitere.

In erster Linie wird aber kein fertiges Produkt entwickelt, sondern „nur" der Standard für das Verfahren anhand fest definierter Szenarien festgelegt.

Der Aufbau eines PayCircle Systems besteht im Wesentlichen aus drei Teilen. Zum einen gibt es den PayCircle Server, der z.B. bei einem TelCo-Betreiber installiert ist. Kunden und Händler haben dort Konten, die Buchungen werden hier vorgenommen, ebenso wie die Verwaltung (Rechungserstellung…). Als zweite Komponente gibt es den PayCircle-Client, der auf den Servern von Händlern oder Dienstanbietern läuft, dort z.B. eingebettet in einen Spieleserver, um die Bezahlfunktionen abzudecken. Der dritte Teil ist nun die Applikation auf dem (mobilen) Endgerät des Verbrauchers. Wenn nun z.B. auf einem Java-fähigen Mobiltelefon ein kostenpflichtiges Spiel ausgeführt werden soll, entsteht folgender, grob dargestellter Ablauf:

- Das Spiel auf dem Endgerät fragt Username und PW für den PayCircle Account ab.

- Mit einem http-POST werden diese Parameter, ergänzt um Angaben zum Spiel, an ein Servlet des Spieleanbieters weitergeleitet und das Spiel geht in „Warteposition“

- Das Servlet des Anbieters verpackt die Angaben in eine SOAP-Nachricht und sendet diese an den PayCircle-Server weiter.

- Der Server entpackt diese Nachricht, prüft die Zahlung und gibt sein o.k (oder verweigert es), verpackt seine Antwort wieder in eine SOAP-Nachricht, die dann an das Servlet beim Anbieter gesandt wird.

- Das Servlet beim Anbieter nun registriert das Durchführen der Zahlung und sendet per http-RESPONSE sein o.k. an die mobile Applikation, woraufhin das Spiel gestartet wird.

Deutlich erkennbar ist die verteilte Struktur der Anwendungen.

3.1.2 PayCircle – Technische Aspekte

Die Referenzimplementierung des PayCircle Servers steht aus bestimmten rechtlichen Gründen nur als Binärdatei zur Verfügung. Sie ist als EAR verpackt und wurde auf der J2EE-Referenzimplementierung von Sun entwickelt und getestet. Der Paymentserver-Core stellt die notwendigen Schnittstellen bereit, die für die Durchführung einer Transaktion nach diesem Standard notwendig sind. Realisiert wurde dies mit Hilfe von Entity-Beans mit Container-Managed-Persistence. Aufbauend auf diesem Core wurde ein Satz Webservices entwickelt um auch in verteilten Systemen auf die Dienste des Payment-Core zugreifen zu können. Diese wurden mit Hilfe von Axis realisiert.

Apache Axis ist eine Implementierung des SOAP-Standards des W3C, die vom Jakarta Projekt vorangetrieben wird. SOAP-Nachrichten werden über HTTP übertragen und können demnach vom Web-Container eines Application Servers empfangen und verarbeitet werden. Axis steht unter http://ws.apache.org/axis/ zum Download bereit.

3.1.3 Probleme bei der Umsetzung

Leider ließ sich der Paymentserver nicht so einfach installieren, wie das von der theoretischen Seite her möglich sein sollte. Schon beim Deployment wurden einige Fehler angezeigt, dass der vorhandene Bytecode einigen benötigten Schnittstellen nicht gerecht würde. Als Ursache für dieses Problem stellte sich schließlich heraus, dass IBM die Unterscheidung zwischen dem primitiven Datentyp int und java.lang.Integer

strenger sieht als Sun, denn auf der Referenzimplementierung von J2EE ließ sich die Anwendung ohne Fehler installieren.

Nachdem von Siemens ein Update zur Verfügung gestellt worden war, in dem sämtliche Vorkommnisse von int durch Integer ersetzt worden waren, ließ sich die Anwendung tatsächlich ohne weitere Warnungen deployen. Die Ausführung der Applikation (bzw. des für die Verwaltung zuständigen Servlets) schlägt nun jedoch mit einer NullPointException fehl. Wird die Anwendung in der Testumgebung des WSAD platziert, um den Fehler zu untersuchen, lassen sich einige der Funktionen des PaymentCore ausführen, andere dagegen schlagen mit einer NotSerializableException fehl. Der Fehler, der vermutlich hinter dieser Meldung steckt, wird vom WSAD als Warnung angezeigt: java.util.Collection muss zur Laufzeit serialisierbar sein.

Leider wurde bis zum heutigen Tag noch keine überarbeitete Version des Paymentservers zur Verfügung gestellt, so dass PayCircle bis jetzt nicht auf WebSphere zu betreiben ist. Immerhin konnten aber die Webservices und Axis ohne größere Probleme installiert werden, so dass die Anwendung voll funktionsfähig sein wird, sobald eine funktionsfähige Version des PaymentCore zur Verfügung steht.

3.2 BeanScriptingFramework für Rexx

Aufgabenstellung: „Integrieren Sie die Scriptsprache REXX bzw. Object REXX in das Bean Scripting Framework des WebSphere Application Servers.“

3.2.1 Funktionsweise des BSF

Das Bean Scripting Framework (BSF) ist eine Architektur um die Zusammenarbeit von Java Applikationen und Applets auf der einen und Scriptsprachen auf der anderen Seite zu ermöglichen. Es wurde 2001 von IBM entwickelt ([IbmBSF2003]). Skriptsprachen wie Netscape Rhino (Javascript), VBScript, Perl, TCL, Python, NetRexx und Rexx werden häufig verwendet, um die Funktionen einer bestehenden Anwendung zu erweitern, oder um mehrere Komponenten verschiedener Anwendungen zu einem gemeinsamen Informationssystem zusammenzufassen.

Es gibt zurzeit einige direkt in Java implementierte Scriptsprachen wie Jacl, JPython und NetRexx. Diese Sprachen können direkt in Java eingebettet werden und haben die Möglichkeit mit Java Objekten ohne weitere Umwege zu interagieren. Mit der Verwendung einer dieser Sprachen wird die Anwendung jedoch fest an diese eine Sprache gebunden.

In der Java-Welt gibt es im Moment noch keine fest definierte Architektur, die es erlauben würde Java Applikationen auf einfach Art und Weise in Skriptsprachen zu verwenden – Das BSF ist genau so eine Architektur. Es erlaubt einer Anwendung von verschiedenen Scriptsprachen verwendet zu werden, die das BSF unterstützen, ohne sich auf eine Sprache festzulegen.

Das BSF erlaubt beide Richtungen des Scripting: Zum einen kann die Java Seite diverse Skripte in einer beliebigen (vom BSF unterstützten) Sprache ausführen, zum anderen können diese Sprachen Java Applikationen ausführen, Objekte erzeugen und schließlich mit ihnen interagieren.

Auf einem Application Server erlaubt das BSF die Verwendung der unterstützten Scriptsprachen innerhalb von Java Server Pages (JSP).

Aktuell wird das BSF vom Jakarta Projekt weiterentwickelt und steht unter [ApacBSF2003] zum Download bereit.

3.2.2 Integration von Rexx in das Bean Scripting Framework

2001 wurde das Paket „BSF4Rexx“ (Bean Scripting Framework for Rexx) auf dem Internationalen Rexx Symposium der Öffentlichkeit vorgestellt. Es war aus einer Seminararbeit von Peter Kalender, einem Studenten an der Universität Essen, hervorgegangen und ist dann von Herrn Prof. Flatscher weiterentwickelt worden.

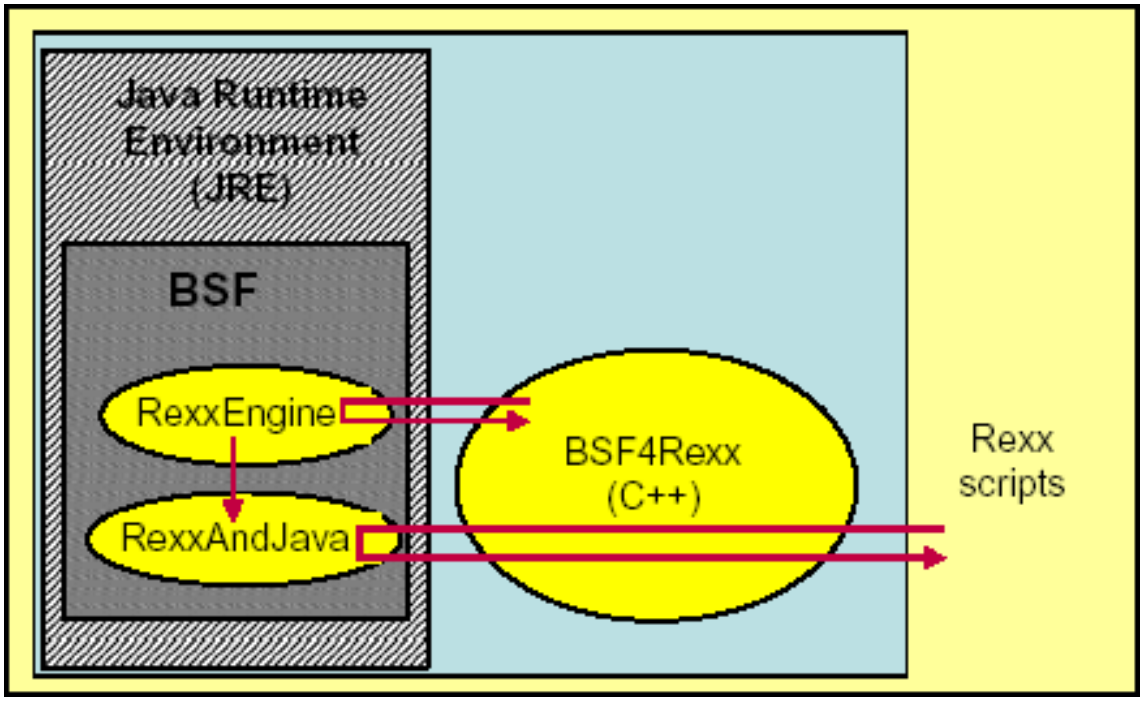

Abb. 3.1: Architektur des BSF4Rexx-Pakets

Das Paket besteht nach [Flat2003] aus drei wesentlichen Bestandteilen:

- BSF4Rexx.jar
  Diese Datei enthält den Java-seitigen Code, der zur Unterstützung des Bean Scripting Frameworks benötigt wird. Sie stellt außerdem das Interface zur Verfügung, das für die Kommunikation mit Java benötigt wird.
  Diese Datei muss zur „CLASSPATH"-Umgebungsvariablen der lokalen Java-Installation hinzugefügt werden.

- BSF4Rexx.dll bzw. BSF4Rexx.so
  Diese Datei stellt die Brücke zwischen Rexx und Java dar und verwendet zur Kommunikation das Java Native Interface (JNI). Das Programm wurde in C++ geschrieben und muss für jede Plattform kompiliert werden. Damit die Datei vom System gefunden wird, muss sie in ein Bibliotheksverzeichnis des Betriebssystems (Alle Verzeichnisse im Suchpfad unter Windows, unter Linux z.B./usr/lib oder ein anderer Bibliothekspfad) kopiert werden

- BSF.cls
  Diese Rexx-Klassendatei stellt Java Objekte innerhalb der Scriptsprache als riesige Objektbibliothek dar.

3.2.3 Installation von BSF4Rexx im WebSphere Application Server

Die Installation des Paketes gestaltet sich relativ einfach, da sie in der Anleitung zum BSF4Rexx recht detailliert beschrieben wird. Vorraussetzungen für die erfolgreiche Installation des Paketes sind eine funktionsfähige Installation eines Java Runtime Environment (JRE) und ein installierter Rexx-Interpreter, der durch das Paket unterstützt wird. Zur Verifizierung dieser Anforderungen kann an der Kommandozeile/Shell folgende Kommandos ausgeführt werden

```
java –version

rexx –v
```

Die Dateien aus dem Zip-Archiv sollten in ein temporäres Verzeichnis entpackt werden. Die Datei bsf4rexx.jar muss in das Bibliotheksverzeichnis des Application Servers kopiert werden. Java Archive, die in diesem Verzeichnis (Appserver/lib/) liegen, werden automatisch beim Start von WebSphere in einen internen Classpath geschrieben, mit dem der Server dann arbeitet. Da bei diesem Vorgang die genaue Reihenfolge aber nicht vorgegeben werden kann, muss zusätzlich noch die originale „BSF.jar"-Datei angepasst werden. Die Liste der unterstützten Scriptsprachen, die in der

Datei „Languages.properties“ steht, muss dementsprechend um einen Eintrag für Rexx erweitert werden.

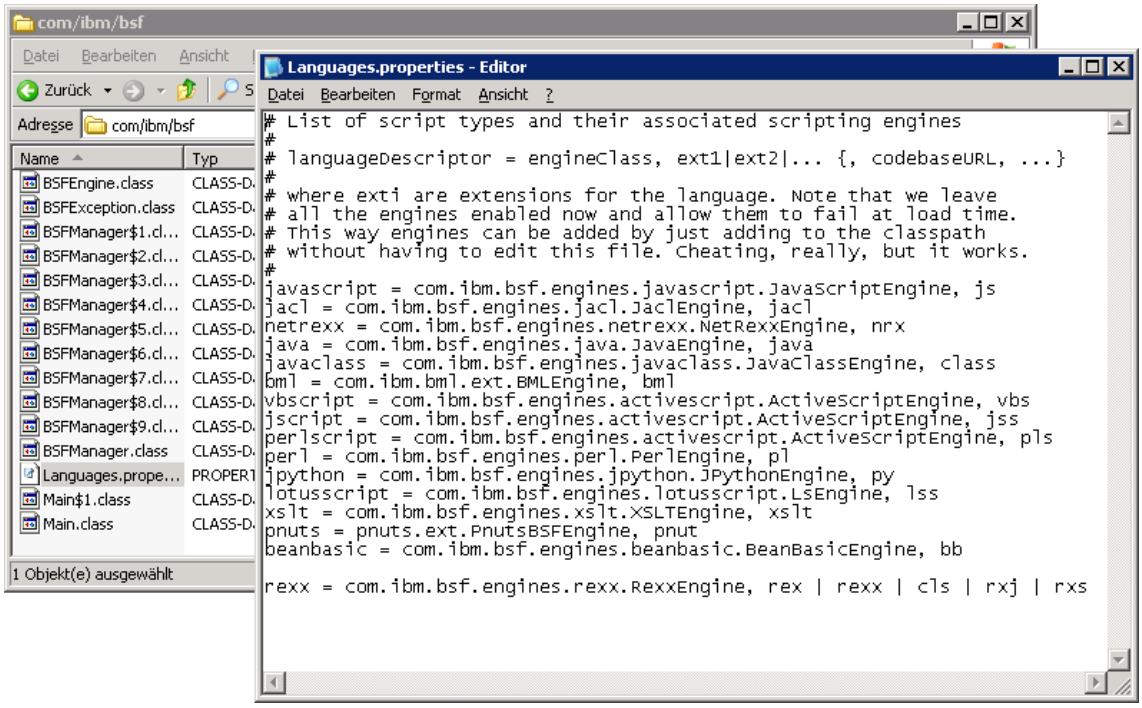

Abb. 3.2: Einträge in der Datei Languages.properties

In der Datei bsf4rexx.jar befindet sich eine entsprechend angepasste Datei, die ursprüngliche Version muss also nur noch aus dem von IBM mitgelieferten Archiv entfernt werden, damit keine Konflikte entstehen können.

Beide Dateien sollten jetzt noch in den CLASSPATH der lokalen Systems eingetragen werden, da die von BSF4Rexx gestartete Java Instanz nichts von dem Klassenpfad weiß, den WebSphere automatisch beim Starten ermittelt hat. Sie könnte dann nicht auf die beiden benötigten Archive zugreifen.

Die für Windows kompilierte Binärdatei bsf4rexx.dll kann ebenso wie die BSF.cls, rexj.cmd und rexja.cmd in das Hauptverzeichnis von Rexx kopiert werden, da sich dieses bereits im Systempfad befindet.

Um die vollständige Installation und Konfiguration zu überprüfen, können die Kommandos

```
rexxj testVersion.rex
rexx testVersion.rex
```

verwendet werden. Der Output sollte folgendermaßen aussehen:

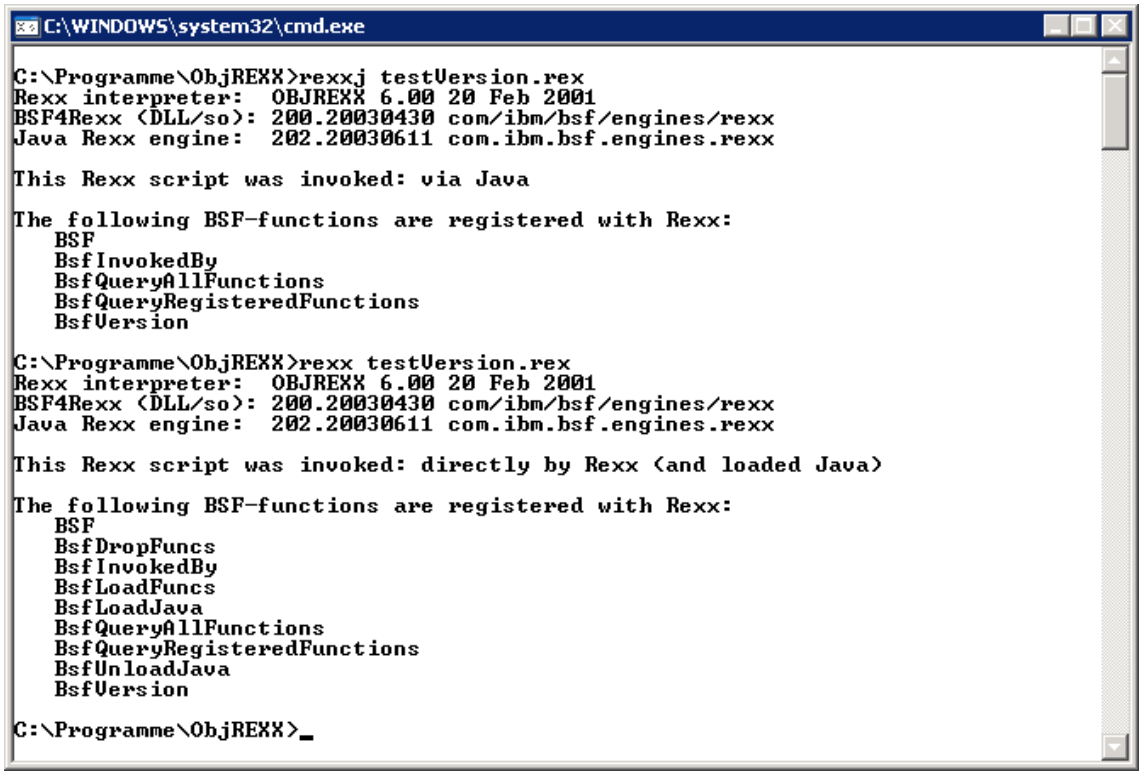

Abb. 3.3: Ausgabe des TestSkripts, aufgerufen von Java oder von Rexx direkt

3.2.4 Beispiele

Um die Integration von WebSphere und Rexx zu demonstrieren, hier zwei kleine Java-Programme, die als Servlets im Application Server laufen und jeweils Rexx verwenden, um die Ausgabe auf einer HTML-Seite zu generieren.

Das erste Beispiel verwendet Rexx um das einfach Beispiel von 100 Würfen mit einer Münze zu simulieren.

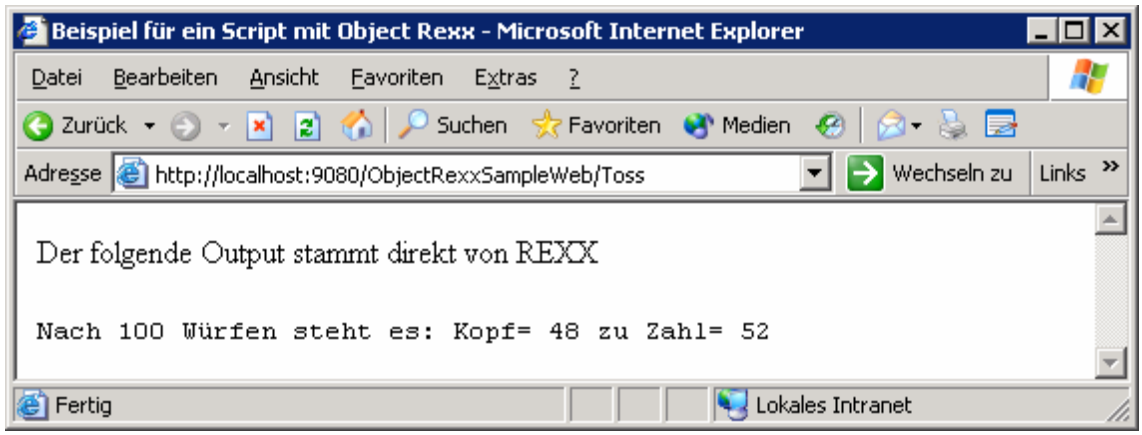

Abb. 3.4: Ausgabe des Toss-Servlets

Das zweite Beispiel verwendet die Scripting Engine von Rexx um ein beliebiges Script, das über ein Eingabefeld an den Server gesendet werden kann, auszuführen. Dieses Servlet ist höchst sicherheitskritisch und sollte daher ausschließlich zu Testzwecken und niemals ungesichert auf dem Server ausgeführt werden.

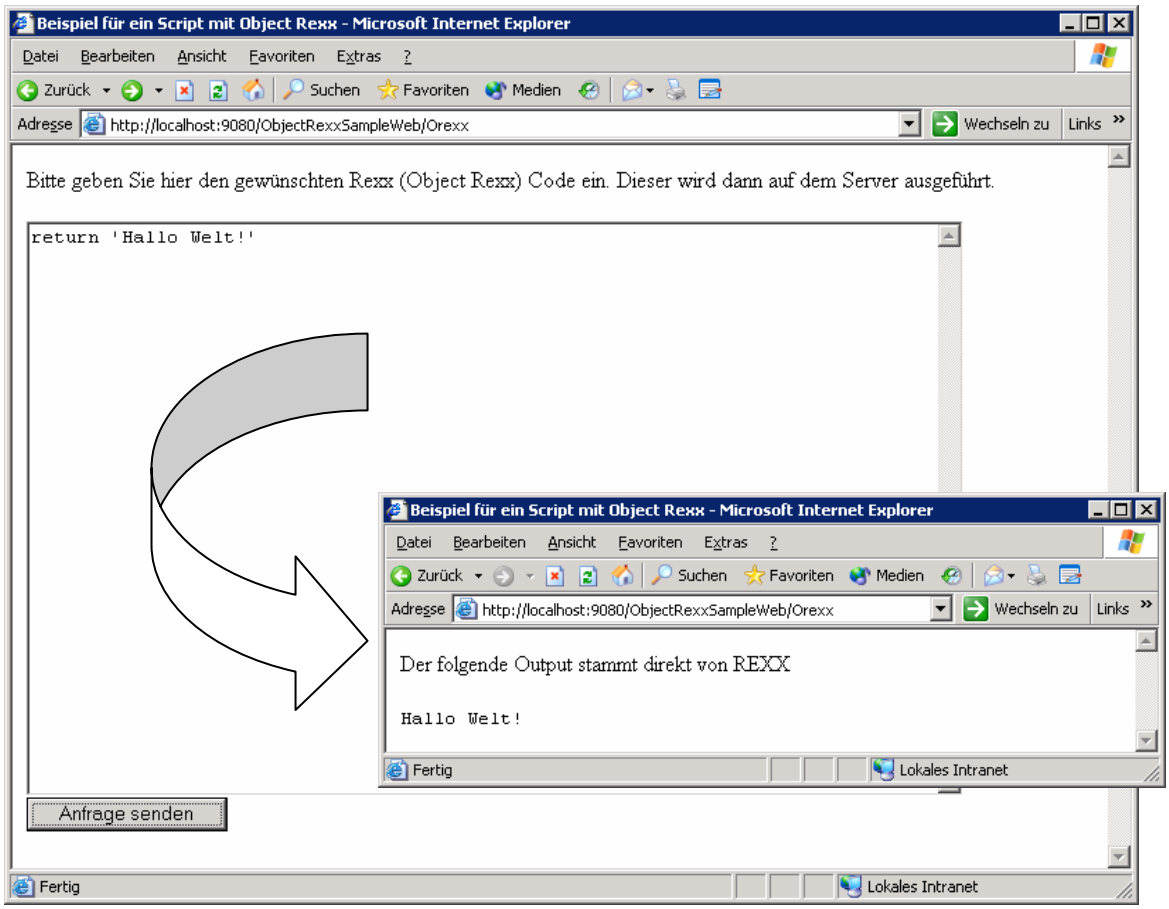

Abb. 3.5: Eingabe und Ausgabe des ORexx-Servlets

Es ist auch möglich Java Server Pages (JSP) mit Hilfe von Object Rexx zu erstellen. Dazu muss jedoch das BSF4rexx angepasst werden. In der Datei RexxEngine.java muss die Funktion `undeclareBean (BSFDeclaredBean bean)` ergänzt werden, da sonst bei der Ausführung eine Exception geworfen wird. Eine angepasste Version des BSF4Rexx wird in Kürze verfügbar werden.

3.3 Message driven Beans

Aufgabenstellung: „Ermöglichen Sie die Kommunikation innerhalb von WebSphere mit Hilfe von JMS. Erstellen Sie zur Demonstration eine Beispielapplikation, die Message driven Beans verwendet“

3.3.1 Synchrone vs. Asynchrone Kommunikation

Bisher war in Java Enterprise Anwendungen nur synchrone Kommunikation möglich (siehe Abb. 3.6). Diese Art Nachrichten zu senden hat einige spezifische Vor- und Nachteile: Auf der einen Seite ist die synchrone Kommunikation darauf angewiesen ihren Partner genau zu kennen, dadurch ist es auch nur möglich Nachrichten an einzelne Empfänger zu schicken. Auf der anderen Seite ist es nur so möglich von den aufgrufenen Methoden ein Feedback zu erhalten, sei es ein bestimmtes Ergebnis oder eine Ausnahmebedingung. Nach [IhHe2002] erfolgt die synchrone Kommunikation im Wesentlichen in diesen vier Schritten:

1. Der Sender ruft eine Methode beim Empfänger auf.
2. Der Empfänger erhält den Aufruf und führt die entsprechende Methode aus.
3. Der Sender bleibt während der Ausführung der Methode blockiert. Wenn die Methode beim Empfänger ausgeführt worden ist, wird eine Quittung, ggf. mit einem Rückgabewert, an den Sender zurückgeschickt.
4. Der Sender erhält die Quittung seines Methodenaufrufs und kann mit den folgenden Verarbeitungsschritten fortfahren.

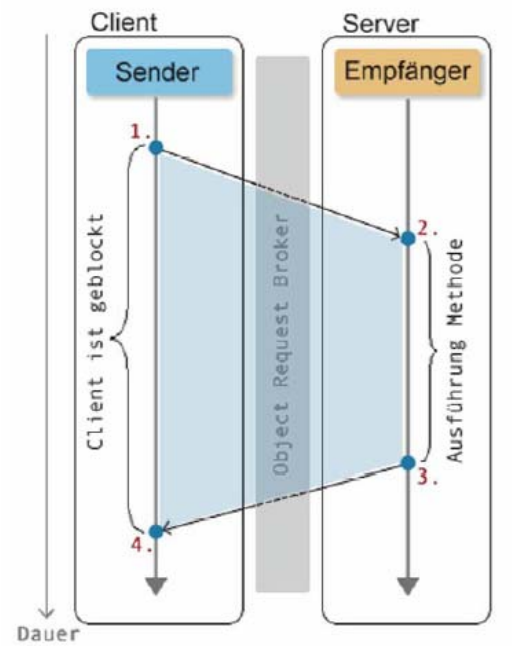

Abb. 3.6: Synchrone Kommunikation mit Hilfe eines ORBs

Im Gegensatz dazu sind bei asynchroner Kommunikation die Objekte sehr viel loser miteinander gekoppelt. Auch hier lässt sich die Kommunikation im Wesentlichen in vier Schritte einteilen:

1. Der Sender erzeugt eine Nachricht und übermittelt diese an den Message Broker.

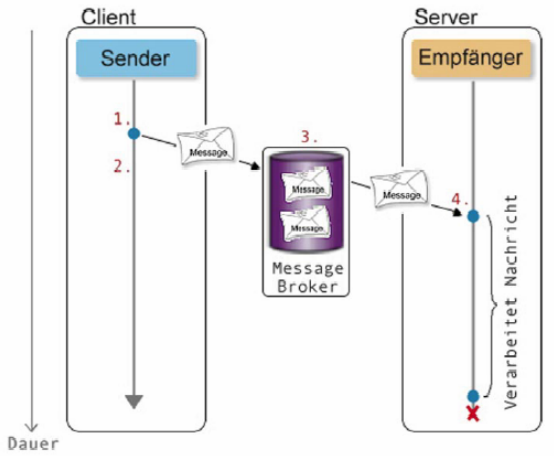

Abb. 3.7: Asynchrone Kommunikation durch einen Message-Broker

2. Im Gegensatz zur synchronen Kommunikation ist der Sender aber nicht blockiert, sondern kann mit der Verarbeitung fortfahren.

3. Nachdem der Message Broker die Nachricht erhalten hat, ermittelt er die Empfänger und leitet die Nachricht an diese weiter.

4. Der Empfänger erhält die Nachricht und verarbeitet sie. Er hat keine Möglichkeit mit dem Sender in direkten Kontakt zu treten oder ihn über das Ergebnis der Verarbeitung zu informieren.

3.3.2 Konzept des Java Message Service (JMS)

Die Aufgabe von JMS ist es, Java einen einheitlichen Zugriff auf diese Nachrichten basierte Middleware (message oriented middleware), über eine standardisierte Schnittstelle, zu ermöglichen. Es ist also ebenso wie bei JDBC der Zugriff auf Systeme unterschiedlicher Hersteller und Bauarten möglich. Sollten Sie weiteres Interesse an JMS haben, finden Sie detailierte Informationen unter [SunJMS2003].

3.3.3 Message driven Beans (MdB)

Message driven Beans wurden in der J2EE Spezifikation 1.3 zu den EJB-Typen hinzugefügt und ermöglichen es Java-Enterprise-Anwendungen auf solche asynchronen Kommunikationsaufrufe direkt innerhalb des EJB-Containers zu reagieren. MdBs sind reine JMS-Nachrichtenempfänger. Im Gegensatz zu den anderen EJBs haben sie kein Remote- oder Home-Interface, sondern werden nur aktiv, sobald ihnen vom EJB-Container eine für sie bestimmte Nachricht zugestellt wird. Die wichtigste Methode dieses Typs Bean ist `public void onMessage(javax.jms.Message msg)`, sie wird vom Container aufgerufen sobald eine Nachricht an das Ziel gesendet wird, mit dem diese MdB konfiguriert wurde.

Für den Einsatz von Message driven Beans muss der Container allerdings zuerst, wie bei Entity Beans mit „container managed persistence“, konfiguriert werden. In IBM Websphere erfolgt diese Konfiguration mit Hilfe der Administrationskonsole.

Zuerst muss der JMS-Server gestartet werden. Hier wurde der in Websphere integrierten JMS-Server benutzt. Dessen Konfiguration verbirgt sich in den Einstellungen zum Anwendungsserver unter Serverkomponenten. Standardmäßig ist der Server deaktiviert. Er muss auf den Status gestartet gesetzt werden und es muss ein Warteschlangenname konfiguriert werden.

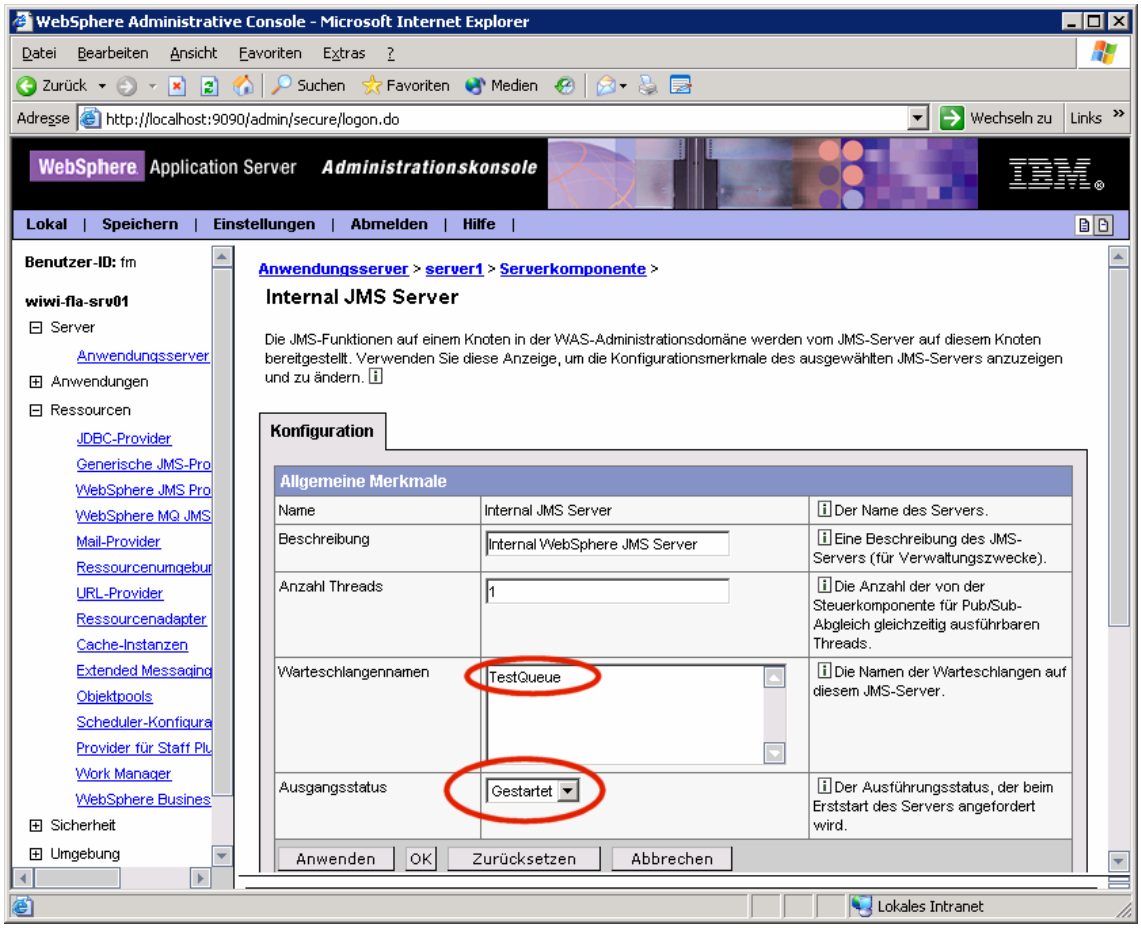

Abb. 3.8: Konfiguration des WebSphere JMS-Servers

Nun können im Bereich „Ressourcen→WebSphere JMS Provider“ eine Queue Connection Factory und eine JMS-Queue eingerichtet werden. Es können die Standardeinstellungen verwendet werden. Dann müssen allerdings die JNDI-Namen beim Deployment der Anwendung angepasst werden.

Zuletzt muss für den Anwendungsserver ein ListenerPort konfiguriert werden. (Unter „Anwendungsserver→Message Listener Service→Listener-Ports“) Dieser ist dafür zuständig die Queue zu überwachen und eintreffende Nachrichten an die Bean weiterzuleiten.

Da nun alle Vorraussetzungen erfüllt sind, kann die Anwendung deployed werden. Wesentlich ist hierbei vor allem der Eintrag des entsprechenden Listener Ports und der Queue Connection Factory, sowie der Queue.

4 Schlussteil

Die Autoren hoffen, einen guten Überblick über die Techniken verschafft zu haben, mit deren Hilfe Enterprise Java Anwendungen geschrieben werden. Es sollte nun auch für Einsteiger möglich sein mit Hilfe von WebSphere komplexere Anwendungen zu erstellen und mit Hilfe der UML zu dokumentieren. Die Beispiele zum Schluss sollten einen ersten Einblick ermöglichen, welche Möglichkeiten in einem solchen Java-basierten Application Server stecken.

Anhang

A Servlets mit Rexx

A.1 Quellcode Toss.java

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import com.ibm.bsf.*;

/**
 * Dieses Servlet verwendet die Rexx ScriptingEngine um den hundertmaligen Wurf
 * mit einer Münze zu simulieren. Zum Schluss wird das Ergebnis auf einer HTML-Seite
 * angezeigt.
 * @author     Fabian Matthiessen
 */
public class Toss extends HttpServlet {
   private BSFEngine rxEngine;

   public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
      resp.setContentType("text/html");
      PrintWriter out = resp.getWriter();
      out.println("<html><head><title>Beispiel f&uuml;r ein Script mit Object "+
         "Rexx</title></head>");
      out.println("<body><p>Der folgende Output stammt direkt von REXX</p>");
      Object result = null;
      try{

         String rexxCode =
            "headcount=0\n" +
            "tailcount=0\n" +
            "/* Eine Münze 100 mal werfen und Resutat ausgeben.*/\n" +
            "do i=1 to 100\n" +
            "  call cointoss /* Die Münze werfen*/\n" +
            "  if result='HEADS' then headcount=headcount+1 /* Zähler anheben*/\n" +
            "  else tailcount=tailcount+1\n" +
            "end\n" +
            "/* Ergebnisse zurückliefern*/\n" +
            "return 'Nach 100 W&uuml;rfen steht es: Kopf=' headcount ' zu Zahl='"+
            "  tailcount\n\n" +
            "cointoss: procedure /* Eine richtige REXX-Prozedur*/\n" +
            "i=random(1,2) /* Zufallszahl aus 1 oder 2 wählen */\n" +
            "if i=1 then return 'HEADS' /* Kopf */\n" +
            "return 'TAILS'\n";
         result=rxEngine.eval ("rexx", 0, 0, rexxCode);
      }catch(BSFException bsfe){
         result="<h2>Exception: "+bsfe.getMessage()+"</h2>";
      }
      out.println("<code>"+result+"</code>");
      out.println("</body></html>");
   }

   public void init() throws ServletException {
      super.init();
      BSFManager manager = new BSFManager ();
      try {
         rxEngine = manager.loadScriptingEngine("rexx");
      } catch (BSFException bsfe) {
         bsfe.printStackTrace();
      }
   }
}
```

A.2 Quellcode Orexx.java

```
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.ibm.bsf.*;
/**
 * Dieses Servlet verarbeitet einen beliebigen eingegeebenen
 * Rexx oder Object Rexx Quellcode und schreibt das zurückgegebene
 * Ergebnis auf eine Seite
 * @author     Fabian Matthiessen
 */
public class Orexx extends HttpServlet {
   BSFEngine rxEngine;

   /**
   * Bei der Initialisierung des Servlets wird versucht die ScriptingEngine für Rexx
   * zu laden. Dieser zeitaufwändige Aufruf muss dann nur ein einziges Mal
   * geschehen.
   */
   public void init() throws ServletException {
      super.init();
      BSFManager manager = new BSFManager ();
      try {
         rxEngine = manager.loadScriptingEngine("rexx");
      } catch (BSFException bsfe) {
         bsfe.printStackTrace();
      }
   }
   /**
   * Diese Methode verarbeitet die GET-Anfragen die vom Browser initial
   * gesendet werden. Es stellt eine Eingabemaske zum bearbeiten des Quell-
   * textes zur Verfügung.
   */
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
      resp.setContentType("text/html");
      PrintWriter out = resp.getWriter();
      out.println("<html><head><title>Beispiel f&uuml;r ein Script mit "+
         "Object Rexx</title></head>");
      out.println("<body><p>Bitte geben Sie hier den gewünschten Rexx (Object "+
         "Rexx) Code ein. Dieser wird dann auf dem Server ausgeführt.</p>");
      out.println("<form method='post'><textarea name='rexxcode' rows='25' "+
         "cols='80'>return 'Hallo Welt!'</textarea><br/><input type='submit'>"+
         "</form>");
      out.println("</body></html>");
   }
   /**
   * Diese Methode verarbeitet den Anfangs eingegebenen Quelltext und führt ihn
   * auf dem Server aus. Die Rückgabewerte werden anschließend in eine HTML-Seite
   * verpackt
   */
   public void doPost(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {
      resp.setContentType("text/html");
      PrintWriter out = resp.getWriter();
      out.println("<html><head><title>Beispiel f&uuml;r ein Script mit Object "+
         "Rexx</title></head>");
      out.println("<body><p>Der folgende Output stammt direkt von REXX</p>");
      Object result = null;
      String rexxCode = req.getParameter("rexxcode");
      try{
         result=rxEngine.eval ("rexx", 0, 0, rexxCode);
         out.println("<code>"+result+"</code>");
      }catch(BSFException bsfe){
         result="<h2>Exception: "+bsfe.getMessage()+"</h2>";
      }
      out.println("</body></html>");
   }
}
```

A.3 Quellcode RexxJSP.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<%@ page language="rexx"%>
<TITLE>RexxJSP.jsp</TITLE>
</HEAD>
<BODY>
<P>
<%=
headcount=0
tailcount=0
do i=1 to 100
        call cointoss
        if result='HEADS' then headcount=headcount+1
        else tailcount=tailcount+1
end

return 'Nach 100 W&uuml;rfen steht es: Kopf=' headcount ' zu Zahl='tailcount

cointoss: procedure
i=random(1,2)
if i=1 then return 'HEADS'
return 'TAILS'
%></P>
</BODY>
</HTML>
```

A.4 Quellcode JavaJSP.jsp

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<%@ page language="java"%>
<TITLE>JavaJSP.jsp</TITLE>
</HEAD>
<BODY>
<P><%
int headcount=0;
int tailcount=0;
for(int i=1;i<=100;i++){
        if(cointoss()) headcount++;
        else tailcount++;
}
out.println("Nach 100 W&uuml;rfen steht es: Kopf="+ headcount +" zu Zahl="+tailcount);
%>
<%!
private boolean cointoss()
{
        if(Math.random()>0.5) return true;
        else return false;
}
%>
</P>
</BODY>
</HTML>
```

B Message driven Bean

B.1 MessageBean.java

```
package test;
/**
 * Bean implementation class for Enterprise Bean: Message
 */
public abstract class MessageBean implements javax.ejb.EntityBean {
   private javax.ejb.EntityContext myEntityCtx;

   public void setEntityContext(javax.ejb.EntityContext ctx) {
      myEntityCtx = ctx;
   }

   public javax.ejb.EntityContext getEntityContext() {
      return myEntityCtx;
   }

   public void unsetEntityContext() {
      myEntityCtx = null;
   }

   public java.lang.Long ejbCreate(java.lang.Long id) throws javax.ejb.CreateException {
      setId(id);
      return null;
   }

   public void ejbPostCreate(java.lang.Long id)
      throws javax.ejb.CreateException {
   }

   public void ejbActivate() {
   }

   public void ejbLoad() {
   }

   public void ejbPassivate() {
   }

   public void ejbRemove() throws javax.ejb.RemoveException {
   }

   public void ejbStore() {
   }
   /**
    * Get accessor for persistent attribute: id
    */
   public abstract java.lang.Long getId();
   /**
    * Set accessor for persistent attribute: id
    */
   public abstract void setId(java.lang.Long newId);
   /**
    * Get accessor for persistent attribute: time
    */
   public abstract java.sql.Date getTime();
   /**
    * Set accessor for persistent attribute: time
    */
   public abstract void setTime(java.sql.Date newTime);
   /**
    * Get accessor for persistent attribute: message
    */
   public abstract java.lang.String getMessage();
   /**
    * Set accessor for persistent attribute: message
    */
   public abstract void setMessage(java.lang.String newMessage);
}
```

B.2 TestQueueMDBBean.java

```
package test;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.jms.*;
import javax.naming.*;
/**
 * Bean implementation class for Enterprise Bean: TestQueueMDB
 */
public class TestQueueMDBBean
   implements javax.ejb.MessageDrivenBean, javax.jms.MessageListener {

   private javax.ejb.MessageDrivenContext fMessageDrivenCtx;
   private MessageBoard mb;

   /**
    * getMessageDrivenContext
    */
   public javax.ejb.MessageDrivenContext getMessageDrivenContext() {
      return fMessageDrivenCtx;
   }
   /**
    * setMessageDrivenContext
    */
   public void setMessageDrivenContext(javax.ejb.MessageDrivenContext ctx) {
      fMessageDrivenCtx = ctx;
   }
   /**
    * ejbCreate
    */
   public void ejbCreate() {
      InitialContext ctx;
      try {
         ctx = new InitialContext();
            mb = ((MessageBoardHome) ctx.
               lookup("ejb/test/MessageBoardHome")).create();
      } catch (Exception e) {
         e.printStackTrace();
      }
   }

   /**
    * onMessage
    */
   public void onMessage(javax.jms.Message msg) {
      TextMessage txtMsg = (javax.jms.TextMessage)msg;
      String messageText=null;;
      try
      {
         messageText = txtMsg.getText();
         mb.createMessage(messageText);
      }
      catch (Exception e)
      {
         e.printStackTrace(System.out);
      }

   }
   /**
    * ejbRemove
    */
   public void ejbRemove() {
   }
}
```

B.3 MessageBoardBean.java

```
package test;

import java.sql.Date;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import javax.naming.*;
/**
 * Bean implementation class for Enterprise Bean: MessageBoard
 */
public class MessageBoardBean implements javax.ejb.SessionBean {
   private javax.ejb.SessionContext mySessionCtx;
   /**
    * getSessionContext
    */
   public javax.ejb.SessionContext getSessionContext() {
      return mySessionCtx;
   }
   public void setSessionContext(javax.ejb.SessionContext ctx) {
      mySessionCtx = ctx;
   }
   public void ejbCreate() throws javax.ejb.CreateException {
   }
   public void ejbActivate() {
   }
   public void ejbPassivate() {
   }
   public void ejbRemove() {
   }

   private MessageLocalHome mlh;

   /**
    * Stellt eine Verbindung zum Home-Interface von {@link:test.Message} her.
    */
   private MessageLocalHome getMessageLocalHome() throws NamingException{
      InitialContext ctx = new InitialContext();
      if(mlh==null){
         mlh = (MessageLocalHome) ctx.lookup("local:ejb/ejb/test/MessageLocalHome");
      }
      return mlh;
   }

   /**
    * Erstellt ein neues Message-Objekt
    */
   public void createMessage(String message) throws CreateException, NamingException{
      MessageLocal ml = getMessageLocalHome().create(
         new Long(System.currentTimeMillis()));
      ml.setMessage(message);
      ml.setTime(new Date(System.currentTimeMillis()));
   }

   public String findAllMessages() throws NamingException, FinderException{
      StringBuffer htmlBoard = new StringBuffer();
      try{
         java.util.Iterator IMessages = getMessageLocalHome().findAll().iterator();
         MessageLocal current;
         htmlBoard.append("<table><tr><th>Datum</th><th>Nachricht</th></tr>\n");
         while(IMessages.hasNext()){
               current = (MessageLocal) IMessages.next();
               htmlBoard.append("<tr><td>"+current.getTime()+"</td>\n");
               htmlBoard.append("<td>"+current.getMessage()+"</td></tr>\n");
         }
      }catch(Exception e){
         htmlBoard.append("<h3>Fehler beim holen der Nachrichten</h3>");
         e.printStackTrace();
      }
      return new String(htmlBoard);
   }

}
```

B.4 JMSTestServlet.java

```
package test;

import java.io.IOException;

import javax.jms.*;
import javax.naming.*;
import javax.servlet.*;
import javax.servlet.http.*;

/**
 * @version    1.0
 * @author     Fabian Matthiessen
 */
public class JMSTestServlet extends HttpServlet {
   /**
   * @see javax.servlet.http.HttpServlet#void (javax.servlet.http.HttpServletRequest,
         javax.servlet.http.HttpServletResponse)
   */
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {

      try
      {
         ServletOutputStream out = resp.getOutputStream();
         out.println(Layout.getHTMLhead("JMS - Nachricht senden"));
         out.println("<body>"+Layout.getHTMLtitle("JMS - Nachricht senden"));
         String inMessage=req.getParameter("message");
         if(inMessage==null){
            out.println("<form method='get'>");
            out.println("<p>Bitte eine Nachricht eingeben:</p>"+
               "<input type='text' name='message'/><input type='submit'/></form>");
         }else{
            InitialContext context = new InitialContext();
            QueueConnectionFactory qConnectionFactory =
               (QueueConnectionFactory)context.lookup("java:comp/env/jms/test/QCF");
            Queue queue = (Queue)context.lookup("java:comp/env/jms/test/Queue");
            QueueConnection qConnection = qConnectionFactory.createQueueConnection();
            QueueSession qSession =
               qConnection.createQueueSession(false,Session.AUTO_ACKNOWLEDGE);
            QueueSender sender = qSession.createSender(queue);
            TextMessage message = qSession.createTextMessage();

            message.setText(inMessage);
            sender.send(message);
            out.println("<p>Nachricht wurde versendet:</p>");
            out.println("<code>"+inMessage+"</code><br/>");
            out.println("<a href='MessageBoardServlet'>Zum MessageBoard</a>");
            sender.close();
            qSession.close();
            qConnection.close();
         }
         out.println("</HTML></BODY>");
      }
      catch(NamingException ne)
      {
         ne.printStackTrace(System.out);
      }
      catch(JMSException e)
      {
         e.printStackTrace(System.out);
         Exception linked = e.getLinkedException();
         linked.printStackTrace(System.out);
      }
   }

}
```

B.5 MessageBoardServlet.java

```
package test;

import java.io.*;

import javax.naming.*;
import javax.servlet.ServletException;

import javax.servlet.http.*;

/**
 * @version    1.0
 * @author     Fabian Matthiessen
 */
public class MessageBoardServlet extends HttpServlet {
   private MessageBoard mb;


   /**
   * @see javax.servlet.http.HttpServlet#void (javax.servlet.http.HttpServletRequest,
javax.servlet.http.HttpServletResponse)
   */
   public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws ServletException, IOException {
      resp.setContentType("text/html");
      PrintWriter out = resp.getWriter();
      out.println(Layout.getHTMLhead("JMS - Messageboard"));
      out.println("<body>"+Layout.getHTMLtitle("JMS - Messageboard"));
      InitialContext ctx;
      try {
         ctx = new InitialContext();
         MessageBoard mb =
            ((MessageBoardHome)ctx.lookup("ejb/test/MessageBoardHome")).create();
         out.println(mb.findAllMessages());
      } catch (Exception e) {
         e.printStackTrace();
         out.println("Error:"+e.getMessage());
      }
      out.println("</body>");

   }

}
```

C Literaturverzeichnis

[ApacBSF2003] *Apache* Jakarta BSF - Bean Scripting Framework. http://jakarta.apache.org/bsf/, Abruf am 2003-11-11.

[Flat2003] *Flatscher, R. G.* The Augsburg Version of the BSF4Rexx. http://wi.wu-wien.ac.at/rgf/rexx/orx14/orx14_bsf4rexx-av.pdf, Abruf am 2003-11-10.

[Fowl2000] *Fowler, M.*: Refactoring, Wie Sie das Design vorhandener Software verbessern. Addison-Wesley, 2000.

[IbmBSF2003] *IBM* bsf-Project. http://www-124.ibm.com/developerworks/projects/bsf, Abruf am 2003-11-10.

[IhHe2002] *Ihns, O.; Heldt, S. M.* Message driven Beans - Einsatz und Integrationsmöglichkeiten. http://www.resco.de/downloads/Ihns_OS_03_02.pdf, Abruf am 2003-11-30.

[Info2002] *Infoskill* J2EE Architecture, Abruf.

[Lehn2003] *Lehner, P.*: Employing a Java based Application Server. Diplomarbeit, Wirtschaftsuniversität Wien. Wien 2003.

[OmgMod2003] *OMG* Catalog of OMG Modeling and Metadata Specifications. http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML, Abruf am 2003-12-05.

[OmgUML2003] *OMG* Introduction to OMG's Unified Modeling Language™ (UML™). http://www.omg.org/gettingstarted/what_is_uml.htm, Abruf am 2003-11-25.

[Sadt2003] *Sadtler, C.*: WebSphere Application Server V5.0 System Management and Configuration. IBM, 2003.

[Sche2002] *Schellhorn*: Vorlesungsskript Softwaretechnik 2002.

[ScVi2003] *Schiffer, B.; Violka, K.*: Spielzimmer aufräumen - Refaktorisieren macht Quellcode lesbarer. In: c't 17/2003 (2003), S. 204-207.

[SunJ2EE2003] *Sun Microsystems, I.* Java 2 Platform Enterprise Edition Sepzification v1.3. http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf, Abruf am 2003-12-10.

[SunJMS2003] *Sun Microsystems, I.* Java Message Service Documentation. http://java.sun.com/products/jms/docs.html, Abruf am 2003-11-03.

[WNA+2003] *Wahli, U.; Norguet, J.-P.; Andersen, J.; Hargrove, N.; Meser, M.*: WebSphere Version 5 Application Development Handbook. IBM, 2003.

D Erklärung

Hiermit versichern wir, dass die vorliegende Arbeit von uns selbständig verfasst wurde und dass wir alle verwendeten Quellen, auch Internetquellen, ordnungsgemäß angegeben haben.

Augsburg, den 11. Dezember 2003 _____
(Fabian Matthiessen)

_____
(Ralph Sauer)