Thorsten Bludau

Diplomarbeit

Portierung von EJB-basierten Anwendungen

Sommersemester 2001
9. September 2001

Universität Dortmund
Fachbereich Informatik
Lehrstuhl für Software-Technologie

Betreuer:
Prof. Dr. Volker Gruhn
Prof. Dr. Heiko Krumm

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Seit dem Erscheinen der ersten offiziellen Spezifikation für *Enterprise JavaBeans* (EJB) im März 1998 (Version 1.0) bieten mittlerweile immer mehr Hersteller einen Applikationsserver mit Unterstützung der EJB-Technologie an. Dabei unterscheiden sich die einzelnen Produkte nicht nur hinsichtlich der unterstützten EJB-Spezifikation (1.0, 1.1 oder 2.0). Viele Server verfügen über einen erweiterten Leistungsumfang, der von Generatoren für den DeploymentDescriptor über spezielle APIs bis hin zu Modellierungswerkzeugen für EntityBeans reicht. Diese Fülle von Features stellt Entwickler und Anwender vor die Frage, welcher dieser zahlreichen Server für seine Probleme die beste Lösung darstellt.

Im Kontext der komponentenbasierten Softwareentwicklung werden an EJB-Komponenten zahlreiche Anforderungen gestellt. Dabei konzentriert sich die Diplomarbeit auf den Bereich der *Portabilität*. Um dies zu motivieren, sei folgendes Szenario erwähnt:

Eine Firma entwickelt ein Softwareprodukt auf der Basis von Enterprise JavaBeans, das heißt ihr Produkt stellt keine monolithische Einheit dar, sondern besteht aus mehreren Komponenten. Da zum Zeitpunkt der Entwicklung der Applikationsserver der Firma A eine besonders gute Unterstützung für die Arbeit mit EJBs bietet (z.B. Tools zum Generieren von Rahmenquellcode aufgrund eines zuvor mit dem Server erzeugten Modells der EntityBeans), entscheidet man sich dann auch, die Anwendung für den Einsatz auf diesem Server zu optimieren. Kurze Zeit später zeigt ein anderes Unternehmen Interesse an diesem Softwareprodukt und würde es gerne selbst einsetzen. Da jedoch bereits der Server der Firma B eingesetzt wird, stellt sich nun die Frage, wie groß der Aufwand ist, um die existierende EJB-Anwendung von Applikationsserver A auf Applikationsserver B zu portieren.

Das „*Write-Once-Run-Anywhere*“-Prinzip (*WORA*) der Programmiersprache Java und der Enterprise JavaBeans selbst lassen vermuten, daß dieses Problem mit nicht besonders viel Aufwand verbunden ist. Diesem Prinzip nach

sind Applikationen, die unter Java entwickelt wurden, *plattformunabhängig* und können somit ohne erneutes Kompilieren auf jeder beliebigen Plattform eingesetzt werden. Da die EJBs auf Java aufsetzen, folgen sie dem *WORA*-Prinzip und erweitern es noch, indem nicht nur die *Plattformunabhängigkeit*, sondern auch eine Unabhängigkeit von den Herstellern der Applikationsserver spezifiziert wird:

> „*Enterprise JavaBeans technology takes the WORA concept to a new level. Not only can these components run on any platform, but they are also completely portable across any vendor's EJB-compliant application server. The EJB environment automatically maps the component to the underlying vendor-specific infrastructure.*“
>
> ([Tho98, S.2])

Es wird also gefordert, daß eine EJB-Applikation ohne Probleme zwischen Applikationsservern, die die EJB-Technologie unterstützen, portiert werden kann. Änderungen am Quellcode sollen dabei vermieden werden und höchstens auf den DeploymentDescriptor beschränkt sein. Aufgrund des geringen Alters der EJB-Technologie gibt es hierzu jedoch bisher noch keine bzw. kaum Erfahrungen. Selbst Schulungsreferenten der Anbieter diverser Server konnten auf die Frage nach den Problemen einer Portierung von EJB-Anwendungen keine Kenntnisse in diesem Bereich vorweisen. Die Diplomarbeit schließt diese Lücke, indem eine konkreten Anwendung auf verschiedene Applikationsserver portiert wird. Dabei werden aus den auftretenden Problemen und ihren entsprechenden Lösungen Rahmenbedingungen abgeleitet, die eine Aussage über die Portabilität von EJB-basierten Anwendungen zulassen.

Insgesamt läßt sich die Frage, ob und unter welchen Umständen das WORA-Prinzip wirklich umgesetzt werden kann, nicht nur durch das bereits genannte Beispiel motivieren. In diesem Zusammenhang tauchen noch weitere Problemstellungen auf, die sowohl von wissenschaftlichem, als auch von wirtschaftlichem Interesse sind.

Aus wissenschaftlicher Sicht stellt sich die Frage, ob EJB-basierte Komponenten wirklich so portabel sind, daß sich aus ihnen eine von jeglichem Applikationsserver unabhängige Anwendung ohne Probleme „zusammenstöpseln“ läßt oder ob diese Portabilität unter Umständen von gewissen Rahmenbedingungen abhängig ist. In diesem Fall besteht ein Interesse darin, diese Rahmenbedingungen zu ermitteln, um sie zur Gewährleistung maximaler Portabilität bei späteren Softwareprojekten umzusetzen. Da die Idee der Portabilität mit die Grundlage der komponentenbasierten Softwareentwicklung bildet, ist es sicherlich interessant zu sehen, inwiefern die Theorie momentan in der Praxis umgesetzt werden kann.

Aus wirtschaftlicher Sicht wiederum kann das Thema der Diplomarbeit aus anderen Problemstellungen heraus motiviert werden. So wurden zum Beispiel Unternehmen gegründet, die sich mit der Entwicklung und dem Vertrieb von EJB-basierter Software beschäftigen. Dabei kann es sich sowohl um einzelne Komponenten, als auch um komplette Anwendungen handeln. Dahinter verbirgt sich die Idee, daß das Hinzukaufen dieser Komponenten für andere Unternehmen unter Umständen billiger und weniger zeitaufwendig ist, als eine Entwicklung im eigenen Hause. Erste Ansätze für einen solchen „EJB-Handel“ finden sich bereits im Internet ([S&SV00, S.10, http://www.ejbprovider.com]) und werden sicherlich in Zukunft auch noch ausgebaut. Funktionieren kann dieses System jedoch nur dann, wenn die angebotenen Komponenten unabhängig von jeglichem Applikationsserver sind. Die entsprechende Portabilität muß gewährleistet sein. Eine nachträgliche Bearbeitung des Quellcodes soll dabei aus zwei Gründen ausgeschlossen werden:

1. Die Anbieter solcher Komponenten können den Quellcode nicht an jeden am Markt verfügbaren Applikationsserver anpassen.
2. Der Kunde darf den Quellcode nicht bearbeiten, da er die Komponenten nur nutzen und nicht noch Einblick in die Software-Lösungen fremder Unternehmen haben soll.

Das Problem der „Portierung von EJB-basierten Anwendungen“ läßt sich also hinreichend motivieren. Im Folgenden stellt sich die Frage, auf welches konkrete Ziel im weiteren Verlauf hingearbeitet und wie dieses Ziel erreicht werden soll.

1.2 Ziel und Aufbau der Arbeit

Das Ziel der Arbeit ist es, Rahmenbedingungen zu ermitteln, die den Grad der Portabilität einer EJB-basierten Anwendung beeinflussen oder aber Rückschlüsse auf die Portabilität existierender Anwendungen zulassen. Diese Rahmenbedingungen sollen vor allem für die Bereiche „Entwicklung von EJB-basierten Anwendungen“, „Einsatz von Applikationsservern“ und „EJB-Spezifikation“ gefunden werden. Dahinter verbergen sich für die einzelnen Bereiche folgende Ideen:

- Bei der Entwicklung von EJB-basierten Anwendungen ist es dem Entwickler möglich, den Grad der Portabilität dieser Anwendungen zu beeinflussen. Die Orientierung an den erarbeiteten Rahmenbedingungen soll dazu führen, daß eine möglichst gute Portabilität erreicht wird. Eine Verletzung der Rahmenbedingungen hingegen wirkt sich negativ auf die Portabilität aus. Ob sich die Aussage der Literatur, EJBs könnten problemlos portiert und auf jedem Applikationsserver eingesetzt werden, in

der Praxis wirklich umsetzen läßt, soll im weiteren Verlauf der Arbeit geklärt werden. Des weiteren können mit Hilfe der Rahmenbedingungen unter Umständen aber auch Aussagen über die Portabilität bereits existierender Anwendungen getroffen werden. Sollte es sich bei einer Anwendung zum Beispiel andeuten, daß aufgrund bestimmter Rahmenbedingungen, die nicht eingehalten wurden, eine Portierung nicht möglich ist oder der Aufwand einer Portierung den Zeit- und Kostenaufwand für eine Neuentwicklung bei weitem übersteigen würde, könnten Zeit und Geld für weitere Untersuchungen hinsichtlich der geplanten Portierung eingespart werden.

- Der Einsatz eines Applikationsservers ist für EJB-basierte Anwendungen unerläßlich. Daher ist es wichtig zu wissen, welche Rahmenbedingungen beim Einsatz eines Applikationsservers Einfluß auf die Portabilität von EJB-basierten Anwendungen nehmen, zumal die meisten Applikationsserver über die Anforderungen der EJB-Spezifikation hinausgehen und einen erweiterten Leistungsumfang für die Arbeit mit EJBs anbieten.

- Die EJB-Technologie ist noch relativ jung und entwickelt sich schnell weiter. Daher gilt es auch hier zu klären, ob sich eventuelle Rahmenbedingungen ableiten lassen, die im Zusammenhang mit der EJB-Spezifikation Auswirkungen auf die Portabilität entsprechender Anwendungen haben.

Um diese Rahmenbedingungen herauszuarbeiten sollen Probleme, die bei der Portierung einer EJB-basierten Anwendung zwischen Servern verschiedener Anbieter auftreten, aufgezeigt und entsprechende Lösungen ermittelt werden. Die Arbeit sieht dazu vor, daß eine existierende EJB-Beispielapplikation von einem Referenzserver auf zwei Server verschiedener Anbieter portiert wird. Dabei werden sowohl allgemeine Probleme betrachtet, als auch Probleme, die nur bei einzelnen Servern auftreten. Von besonderem Interesse sind in diesem Zusammenhang jedoch immer die Ursachen der Portierungsprobleme, da sie die Grundlage für die aufzuzeigenden Rahmenbedingungen bilden.

Um das beschriebene Ziel zu erreichen, sehen die ersten sechs Kapitel die Erarbeitung von Grundlagen vor. Zunächst wird in Kapitel 1 das Thema motiviert und sowohl das Ziel, als auch der genaue Aufbau der Arbeit beschrieben. Anschließend findet in Kapitel 2 eine Einordnung in die Literatur bzw. den wissenschaftlichen Kontext statt. Das Ergebnis der ersten beiden Kapitel ist unter anderem eine grobe Aufteilung der Diplomarbeit in die Bereiche *Portabilität*, *Enterprise JavaBeans* und *Applikationsserver*. Diese Bereiche werden in den Kapiteln 3 und 4 ausführlich beschrieben, wobei sich Kapitel 3 mit allgemeinen Beschreibungen beschäftigt und Kapitel 4 wichtige, in den einzelnen Bereichen vorkommende Fachbegriffe definiert. Besonders wichtig ist in diesem Zusammenhang Kapitel 4.4, da dort elementare Vereinbarungen ge-

troffen werden, die den Rahmen der Arbeit abgrenzen. Das nachfolgende Kapitel 5 erläutert die Beispielanwendung von Malte Hülder, die auf die verschiedenen Applikationsserver zu portieren ist. Diese Server werden in Kapitel 6 beschrieben. Dabei konzentriert sich die Beschreibung vor allem auf den Leistungsumfang, der für die Arbeit mit EJBs zur Verfügung steht. In diesem Kontext sollen Thesen aufgestellt werden, die sich mit dem Einfluß dieser Leistungen auf die Portabilität der Anwendung beschäftigen.

Die Kapitel 7 und 8 behandeln den praktischen Teil der Diplomarbeit - die Portierung der Beispielanwendung auf die Applikationsserver *Powertier* (Kapitel 7) und *WebSphere* (Kapitel 8). Im Zuge dieser Portierungen können die bis dahin aufgestellten Thesen aus Kapitel 6 entweder bewiesen oder wiederlegt werden. Es sind jedoch auch Probleme denkbar, die zuvor nicht abzusehen waren. Ein Vergleich der theoretischen und praktischen Ergebnisse findet sich in Kapitel 9. Diese Ergebnisse fließen anschließend in die Rahmenbedingungen mit ein, die die Portabilität von EJB-basierten Anwendungen beeinflussen. Zum Abschluß beschäftigt sich Kapitel 10 noch mit einem Ausblick auf zukünftige Ereignisse (EJB-Spezifikation 2.0, neue Server, etc.) und zieht ein Fazit aus den Ergebnissen der Diplomarbeit.

Kapitel 2

Einordnung in die Literatur

Die vorliegende Diplomarbeit beschäftigt sich mit *Portierungen*, *Enterprise JavaBeans* und *Applikationsservern*. An dieser Stelle soll nun die Einbettung in den wissenschaftlichen Kontext bzw. die Einordnung in die wissenschaftliche Literatur vorgenommen werden um einen Überblick darüber zu bekommen, wo sich die Diplomarbeit thematisch einfügt und wo sie bestehende Ergebnisse erweitert bzw. wo sie neue Erkenntnisse bringen soll.

2.1 Portierung

Das Thema *Portierung* läßt sich sowohl von einer theoretischen, als auch von einer praktischen Seite betrachten. Dabei geht es bei den praktischen Betrachtungen hauptsächlich um einzelne, sehr konkrete Portierungen, während die theoretischen Ansätze versuchen, allgemeine Definitionen und Grundlagen für die Bereiche *Portierung* und *Portabilität* zu finden.

Einen ersten allgemeinen Überblick über den theoretischen Bereich von *Portierungen* im Kontext der Softwareentwicklung können [Bal96, S.776], [Nag90, S.24/S.330] und [Som89, S.339 ff.] entnommen werden. Eine intensivere Betrachtung findet sich bei [Bro77], [Sch80] und [Wal82]. Aufbauend auf diesen Grundlagen versuchen [BK88] und [Roe90] *Portabilitätsmaße* zu ermitteln und gehen der *Standardisierung von Software-Portabilität* nach. Den Anspruch der EJB-Technologie nach einer problemlosen Portabilität entsprechender Applikationen nennen unter anderem [MH99b, S.2] und [Tho98, S.4/5].

Im Gegensatz dazu finden sich für den praktischen Bereich meist nur sehr spezielle Portierungsprobleme, die nichts mit dem Thema der Diplomarbeit gemeinsam haben. Ein Transfer von bekannten Problemen und Lösungen auf diese Arbeit ist daher nicht vorgesehen. Als Grundlage können jedoch [Con00], [MH99a], [MH00] und [Per00] hinzugezogen werden, die sich in ihren Artikeln mit dem Problem der Portierung zwischen zwei Servern bzw. zwischen zwei EJB-Spezifikationen beschäftigen. Desweiteren finden sich bei [Bro77, S.119 ff.] im Kapitel „Techniques for Enhancing Portability“ Abschnitte über eine logische

und physische Trennung der Anwendung in spezifische und unspezifische Komponenten/Module. Der Begriff „spezifisch“ meint dabei eine Abhängigkeit zur jeweiligen Umgebung (z.B. Programmiersprache, Datenbank, etc.). Diese Aufsplittung sollte von Anfang bis Ende der Softwareentwicklung beachtet werden, da sich die unspezifischen Komponenten meist mit keinem oder nur sehr geringem Aufwand portieren lassen, was den Gesamtaufwand für eine Portierung unter Umständen dramatisch senken kann. Im späteren Verlauf der Portierung der Beispielapplikation soll dieser Ansatz weiter vertieft werden.

2.2 Enterprise JavaBeans

Einen ausführlichen Einstieg in den Bereich der EJBs bieten [MH99b], [Rom99] und [Tho98] an, wohingegen [Fie00], [jGu00] oder [Zuk99] sich jeweils lediglich auf ein notwendiges Grundwissen beschränken. Grundlagen für die Programmiersprache Java von Sun Microsystems Inc., die als Basis für die EJBs verwendet wird, finden sich bei [Fla97], [Gra97a] oder [Gra97b]. Entsprechend wird an dieser Stelle auf [Gru99] für eine Einführung in die komponentenbasierte Softwareentwicklung verwiesen. Nennenswert sind an dieser Stelle aber auch noch die Artikel von [BM00], [Haw00] und [Mer00c], die sich mit der EJB-Spezifikation 2.0 beschäftigen, wie auch [Kle00], der an einem kurzen Beispiel die Entwicklung einer EJB-basierten Applikation für den PowerTier-Server von Persistence darstellt.

Um die Probleme, die sich bei einer Portierung ergeben, verstehen zu können, sind Kenntnisse der bisherigen EJB-Spezifikationen eine zwingend notwendige Voraussetzung, da sich die Unterschiede zwischen den einzelnen Versionen unmittelbar auf die Realisierung einer EJB-basierten Applikation auswirken (z.B. Format des DeploymentDescriptor, Verwendung von Container-/Bean-Managed-Persistence). Insbesondere Monson-Haefel beschäftigt sich in seinen Artikeln [MH99a] und [MH00] mit dem Problem der Portierung von EJBs zwischen der EJB-Spezifikation 1.0 und 1.1. Die Spezifikationen selbst finden sich unter [HM98], [HM99] und [DeM00].

2.3 Applikationsserver

Im Verlauf der Diplomarbeit wird die vorliegende Beispielapplikation auf zwei Applikationsserver portiert, um aus den sich ergebenden Problemen und Lösungen die in Kapitel 1 beschriebenen Rahmenbedingungen zu ermitteln. Ein weiteres Kernthema der Arbeit stellen somit die EJB-Technologie unterstützende Applikationsserver dar, die mittlerweile von zahlreichen Anbietern angeboten werden.

Die Diplomarbeit beschäftigt sich mit genau drei dieser Server. Einen allgemeinen Überblick über Applikationsserver und ihre Aufgaben liefern [Har00],

[Sun99] und [Tho98]. Die speziellen für die Diplomarbeit erforderlichen Informationen und Dokumentationen zu den verwendeten Servern kommen von den Anbietern selbst und finden sich unter [Per01], [Web01a] und [Web01b]. Eine von den Anbietern unabhängige Vorstellung der einzelnen Server und ihrer Leistungsmerkmale enthalten [Fla00], [Gla00] und [Mer00b].

Kapitel 3

Grundlagen

Dieses Kapitel gibt nun einen kurzen Überblick über die drei Kernthemen der Diplomarbeit. Dies sind die Themen *Portabilität*, *Enterprise JavaBeans* und *Applikationsserver*. Die jeweiligen Abschnitte beschränken sich auf das für das Verständnis der Arbeit notwendige Grundwissen und werden teilweise in den weiteren Kapiteln noch vertieft.

3.1 Portierung und Portabilität

In den ersten zwei Kapiteln wurde bereits kurz erwähnt, daß sich die Diplomarbeit mit der Portierung einer EJB-basierten Applikation zwischen Servern verschiedener Hersteller beschäftigt. Dabei stellt sich zunächst einmal die grundsätzliche Frage, was eigentlich unter einer Portierung bzw. unter dem Begriff der Portabilität zu verstehen ist. Erste Ansätze und Definitionen hierfür gibt es bereits seit den 50-er und 60-er Jahren, so zum Beispiel von A.J. Perlis im Jahre 1968, als er auf einer NATO-Arbeitstagung in Garmisch den Begriff der Portabilität wie folgt definierte:

> „*Portability is the property of a system which permits it to be mapped from one environment to a different environment.*“
>
> ([Sch80, S.5])

Da es sich hierbei jedoch um eine sehr simple Definition handelt nach der eigentlich jedes System portierbar ist, gibt es zahlreiche weitere Ansätze, die sich um eine entsprechende Verfeinerung bemühen. Dabei ist der zugrundeliegende Begriff der *Portierung* stets wie folgt zu interpretieren:

Eine *Portierung*, wie sie auch im Kontext der Diplomarbeit zu verstehen ist, bezeichnet die Übertragung eines Softwareprodukts von einer fest definierten Umgebung (Hard- und Software, etc.) auf eine davon abweichende Umgebung. In der Literatur und vor allem in den hier aufgeführten Definitionen der *Portabilität* wird der Begriff *Portierung* meist nur indirekt definiert. In seiner oben aufgeführten Definition beschreibt Perlis eine *Portierung* als eine Abbildung („...to

be mapped...“) eines Systems von einer Umgebung („environment“) auf eine abweichende Umgebung ohne jedoch den Begriff *Portierung* explizit zu definieren. Desweitern werden in der Literatur die Begriffe *Abbildung* und *Transformation* synonym für den *Portierung* verwendet.

3.1.1 Theorie und Praxis

Allgemein teilt sich der Bereich „Portabilität“ in der Literatur in zwei Unterbereiche auf.

- Der *praktische* Bereich beschäftigt sich mit speziellen Portierungsproblemen der realen Welt. Das kann zum Beispiel die Portierung einer Windows-Anwendung auf ein Unix-System oder auch die Portierung von Daten zwischen verschiedenen Datenbanksystemen sein. Dabei ist es wichtig zu wissen, welcher Teil (Betriebssystem, Programmiersprache, Datenbank, etc.) variabel und welcher konstant ist. Im Kontext der Diplomarbeit bleiben sowohl die Programmiersprache Java, als auch die Datenbank und das Betriebssystem erhalten (→ konstanter Teil). Lediglich der verwendete Applikationsserver wird ausgetauscht (→ variabler Teil).

- Der *theoretische* Bereich hingegen versucht zu den Begriffen der *Portierung* und der *Portabilität* Aussagen und wissenschaftliche Definitionen zu finden, die möglichst allgemein formuliert sind.

Die Diplomarbeit ist eher dem theoretischen Bereich zuzuordnen, da sie Rahmenbedingungen ermittelt, die eine Aussage über die Portabilität von EJB-basierten Anwendungen zulassen. Die Portierungen selbst, welche den praktischen Teil darstellen, sind nur Mittel zum Zweck, um die theoretischen Ergebnisse herzuleiten und zu festigen.

3.1.2 Definitionen

Mittlerweile gibt es zahlreiche theoretische Ansätze, die sich mit der Beschreibung und Definition des Begriffs *Portabilität* beschäftigen. Einige Beispiele hierfür sind:

> „*Portability is a measure of ease with which a program can be transferred from one environment to another: If the effort required to move the program is much less than that required to implement it initially, then we say that it is highly portable.*“
>
> ([BK88])

> „*Ein Softwaresystem heißt portabel, wenn es mit im Verhältnis zu den Herstellungskosten geringem Aufwand auf andere Grundsysteme umgestellt werden kann.*“
>
> ([BK88])

> „*The portability of a program depends upon the ease with which it can be transferred and made useful, without modification of its properties, in a new environment. ...A program is portable if the effort required for its transport is much less than the effort required for its initial implementation and if it retains its initial qualities after the transport.*“
>
> ([Roe90])

Bei Balzert ([Bal96]) wird die Portabilität sogar in einem weiteren Schritt in die verschiedenen Abstufungen Objektcode-, Quellcode- und Entwurfsportabilität unterteilt.

Neben solchen Definitionen gibt es aber auch sogenannte *Portabilitätsmaße*, wie sie zum Beispiel von Hommel aufgestellt werden:

> „*Portabilität ist eine Eigenschaft von Software (Programme und Daten), die es erlaubt, diese Software mit relativ geringem Aufwand von einem Grundsystem auf ein anderes zu übertragen. Bei dieser Übertragung dürfen andere Softwareeigenschaften nicht (Korrektheit) oder nur unwesentlich (Effizienz) beeinträchtigt werden, es kann erforderlich sein, zusätzlichen Aufwand für die Adaptierung zu treiben.*“
>
> *Ein Software-Produkt gilt dabei als portabel, wenn die Bedingung* $0,5 \leq P \leq 1$ *erfüllt ist. Dabei sei* P *das Portabilitätsmaß, welches eine Beziehung zum Implementationsaufwand herstellt:*
>
> $P = \frac{A_I}{A_I + A_P + A_A}$ *,mit*
> A_I = *Aufwand für die erste Implementierung eines Software-Produktes*
> A_P = *Aufwand für die Portierung*
> A_A = *Aufwand für die Adaptierung*
>
> (vgl. [BK88])

Im Rahmen dieser Definitionen taucht neben dem Begriff der Portierung auch der Begriff der *Adaptierung* auf. Dieser bezeichnet die Anpassung des portierten Systems an die neue Umgebung. Im Kontext der Diplomarbeit bedeutet dies, daß im Zuge der Portierung die Beispielapplikation zunächst auf dem neuen Server lauffähig und unabhängig vom alten Server gemacht wird. Anschließend kann die portierte Applikation an die Leistungen und Fähigkeiten des neuen Servers

„adaptiert“ werden. Dem Anwender steht somit nach der *Adaptierung* ein möglichst großer Leistungsumfang des neuen Applikationsservers zur Verfügung.

Diese Anzahl von Definitionen zeigt, daß es nicht möglich ist, von DER Definition für den Portabilitätsbegriff zu sprechen. Für das weitere Verständnis ist es jedoch wichtig zu klären, welche Definition als Grundlage verwendet wird und was der Begriff „Portabilität“ im Kontext der Diplomarbeit eigentlich meint. Diese Definition und einige weitere, für die Arbeit notwendige Vereinbarungen, finden sich in Kapitel 4.4.

3.2 Komponenten und ihre Eigenschaften

In den letzten 30 Jahren hat sich die Softwareentwicklung immer mehr von den großen monolithischen Programmen entfernt und über die objektorientierte bis hin zur heutigen komponentenbasierten Softwareentwicklung weiterentwickelt (vgl. [Gru99]). Ein maßgeblicher Grund für diese Entwicklung bestand darin, daß diese großen monolithischen Programme einfach zu unhandlich und unüberschaubar waren. Die Wartung und Wiederverwendung war dann nur selten und wenn, dann meist nur unter enormem Kosten- und Zeitaufwand zu realisieren. Um dem ein Ende zu bereiten, wurde das jeweils zu lösende Problem aufgesplittet und durch sogenannte Objekte beschrieben, deren Eigenschaften und Zusammenspiel dann das reale Problem wiederspiegeln sollten. Durch diese Abstraktion entstand die objektorientierte Softwareentwicklung.

Bei der komponentenbasierten Softwareentwicklung wiederum geht der Schritt der Abstraktion noch etwas weiter. Hier werden sogenannte Komponenten gebildet, die aus mehreren Klassen (4-10) bestehen. Diese Komponenten können als eine Art Baustein verstanden werden, die zu einer kompletten Anwendung zusammengesetzt werden können. Dabei ist es unerheblich, wie die zu leistende Funktion innerhalb der Komponente realisiert ist (Black Box). Sie wird nach außen hin durch vertraglich spezifizierte Schnittstellen definiert.

Weitere Vorteile der komponentenbasierten Softwareentwicklung bestehen zum Beispiel in der Wartung, Wiederverwendung, Entwicklungszeit und dem Kostenaufwand:

- Komponenten können ohne Probleme gegen andere Komponenten mit denselben gewünschten Funktionalitäten ausgetauscht werden oder können schnell und einfach in eine andere Anwendung eingebaut werden. Dies wird insbesondere durch die Festlegung der Schnittstellen gewährleistet, was wiederum zu einer Unabhängigkeit von der verwendeten Programmiersprache führt und ein erneutes Kompilieren verhindert.

- Es muß nicht immer die komplette Anwendung getestet werden, sondern oftmals nur die Funktionalität einzelner Komponenten.

- Bereits existierende Komponenten müssen nicht noch einmal programmiert werden, sondern können von anderen Anbietern / Entwicklern zugekauft werden.
- Der Entwickler behält vor allem bei großen Softwareprojekten einen besseren Überblick

Allgemein gibt es 16 Eigenschaften die von Komponenten erfüllt werden. Auszugsweise seien an dieser Stelle Konfigurierbarkeit, Wiederverwendbarkeit, Ortstransparenz, Bewährtheit, Portabilität und Programmiersprachenunabhängigkeit genannt. Standards für die komponentenbasierte Entwicklung sind zum Beispiel JavaBeans oder Enterprise JavaBeans, die auf der Programmiersprache Java von Sun Microsystems Inc. aufsetzen.

3.3 Die EJB-Technologie

Im März 1998 veröffentlichte SunSoft die erste Spezifikation der Enterprise JavaBeans (EJB) in der Version 1.0 ([HM98]). Standard ist mittlerweile Version 1.1 ([HM99]), die jedoch bald von der im Oktober 2000 veröffentlichten Spezifikation 2.0 ([DeM00]) abgelöst werden soll. Sie beschreibt ein Komponentenmodell für Server-Komponenten und ist auf die Programmiersprache Java zugeschnitten.

EJBs werden zur Modellierung von sogenannten Geschäftsobjekten verwendet. Diese kapseln kleinere, im Kontext eines Geschäftsprozesses zusammengehörige Objekte. Einfache Beispiele für solche Geschäftsobjekte sind zum Beispiel Personen, Konten oder aber auch Zahlungsvorgänge. Sie werden im Gegensatz zu JavaBeans, die auf Clientseite verwendet werden, als Serverkomponenten in verteilten Anwendungen entwickelt und eingesetzt. Auch für EJBs werden die 16 Eigenschaften für Komponenten aus Kapitel 3.2 gefordert, so daß auch hier bei der Entwicklung darauf geachtet werden muß, daß zum Beispiel die Eigenschaften Wiederverwendung und Portabilität erhalten bleiben.

Die EJB-Spezifikation setzt auf der in Abbildung 3.1 gezeigten 3-Schicht-Architektur auf.

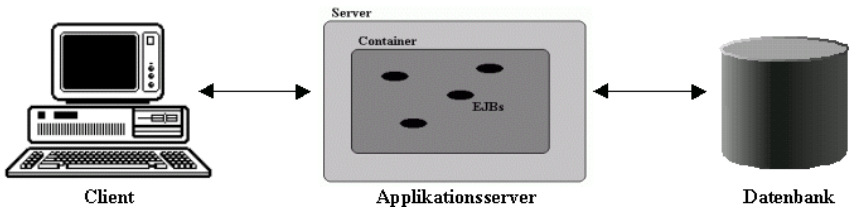

Abbildung 3.1: 3-Schicht-Architektur

Auf der obersten Schicht, der sogenannten Präsentationsschicht, befindet sich ein Client, der für die Darstellung der Informationen zuständig ist. Die unterste

Schicht wird von einer Datenbank gebildet, die über eine spezifizierte Schnittstelle (z.B. JDBC) angesprochen wird. Zwischen diesen beiden Schichten befindet sich ein Applikationsserver, der einen Container zur Verfügung stellt, welcher als Heimat der EJBs angesehen werden kann. Dieser Container enthält also die durch die EJBs repräsentierte Geschäftslogik und tauscht seine Informationen mit dem Client meist über die RMI-Schnittstelle aus.

Allgemein unterteilt man Enterprise JavaBeans in die drei großen Kategorien Entity Beans, Session Beans und Message-driven Beans ([Haw00]). Die Kategorie der Message-driven Beans ist erst in der Spezifikation 2.0 hinzugekommen. Abbildung 3.2 stellt die beschrieben Aufteilung grafisch dar.

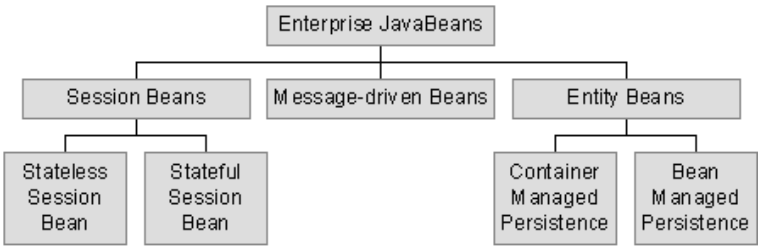

Abbildung 3.2: EJB-Typen

Die Informationen wie sich die einzelnen Beans zur Laufzeit verhalten sollen erhält der Container durch den DeploymentDescriptor. In der Version 1.0 lag dieser noch als serialisiertes File vor, seit der Version 1.1 wird er im XML-Format abgelegt. Der DeploymentDescriptor enthält zu den einzelnen EJBs Informationen über den kompletten Einsatz im Container. Er definiert zum Beispiel das Transaktions- und Sicherheitsverhalten.

3.4 Aufgaben eines Applikationsservers

Ein *Applikationsserver* ist ein Server-Programm auf einem Rechner in einer verteilten Umgebung, das für bestimmte Applikationsarten (z.B. Web- oder EJB-Applikationen) eine Infrastruktur zur Verfügung stellt, auf der die Applikationen aufsetzen können.

Da sich Kapitel 6 noch ausführlich mit *Applikationsservern* beschäftigt, wird an dieser Stelle nicht weiter auf diesen Begriff eingegangen.

Kapitel 4

Begriffe und Definitionen

4.1 Begriffe der Beispielapplikation

Die zu portierende Beispielapplikation stellt ein Beispiel aus der Versicherungsbranche dar. Dabei wird gezeigt, daß sich das entworfene Historisierungskonzept für Versicherungsverträge mit Hilfe von Enterprise JavaBeans realisieren läßt. Da das Thema dieser Diplomarbeit jedoch die Portierung von EJB-basierten Anwendungen ist, kann an dieser Stelle auf eine detaillierte Einführung in die Begriffe des Versicherungswesens verzichtet und auf die entsprechenden Stellen aus [Hue00] verweisen werden. Viel wichtiger ist eine Beschreibung der technischen Realisierung, die in Kapitel 5 behandelt wird.

4.2 Bestandteile von Enterprise JavaBeans

Die EJB-Spezifikation der Enterprise JavaBeans sieht für jede Bean bestimmte Bestandteile vor. Dies sind jeweils ein *HomeInterface*, ein *RemoteInterface* und eine *BeanClass*. Bei *EntityBeans* kommt auch noch eine *PrimaryKeyKlasse* hinzu. Des weiteren ist für den späteren Einsatz in einem Applikationsserver der sogenannte *DeploymentDescriptor* von großer Bedeutung.

4.2.1 EJB-Spezifikation

Die EJB-Spezifikation dient als Grundlage für die Arbeit mit Enterprise JavaBeans und beschreibt diese Komponentenarchitektur zur Entwicklung und zum Einsatz von objekt-orientierten Enterprise Applikationen unter anderem anhand von sechs Rollen. Dies sind der *Enterprise Bean Provider*, der *Application Assembler*, der *Deployer*, der *EJB Server Provider*, der *EJB Container Provider* und der *System Administrator*. Für diese Rollen schreibt die Spezifikation spezielle Aufgaben und Pflichten vor und gibt an, welche Beziehungen untereinander notwendig sind, um die EJB-Technologie auch wirklich realisieren zu

können. Der *EJB server provider* übernimmt zum Beispiel die Aufgabe, einen Applikationsserver zur Verfügung zu stellen, der die Aufgabe hat, dem Anwender eine Infrastruktur für die Arbeit mit EJBs anzubieten. Dabei kann es durchaus vorkommen, daß mehrere Rollen von ein und der selben Instanz übernommen werden. So stellt ein *EJB Server Provider* meist nicht nur einen Applikationsserver zur Verfügung, sondern fungiert gleichzeitig als *EJB Container Provider*, indem er in seinen Server zusätzlich einen Container integriert.

Bisher hat Sun die EJB-Spezifikationen im Abstand von jeweils einem knappen Jahr veröffentlicht und von Version zu Version teilweise recht umfangreiche Änderungen bzw. Erweiterungen vorgenommen. Diese Veränderungen beruhen meist auf Ungenauigkeiten der vorherigen Versionen oder auf Problemen, die erst nach der Veröffentlichung der vorherigen Versionen erkannt wurden. Ein Beispiel für eine solche Ungenauigkeit besteht zum Teil in den Beschreibungen der oben aufgeführten Rollen, ihrer Aufgaben und ihrer notwendigen Kommunikation. Sun hat sich mit diesen Problemen seit Erscheinen der ersten Spezifikation beschäftigt und die entsprechenden Bereiche noch weiter verfeinert. Insbesondere der Bean-Container Contract, also die Zusammenarbeit zwischen den EJBs und dem Container wurde detaillierter beschrieben. Auf der anderen Seite gibt es jedoch auch Bereiche, die weniger gut bis gar nicht erläutert werden, wie zum Beispiel der Server-Container Contract. Aufgrund solcher fehlenden Informationen seitens der Spezifikation liefern die Anbieter der Applikationsserver ihre Produkte bisher mit einem eigenen Container aus und realisieren die Kommunikation zwischen Server und Container nach eigenem Ermessen.

Fest steht jedoch, daß es aufgrund syntaktischer Unterschiede zu Portierungsproblemen kommen wird, wenn die beteiligten Applikationsserver auf verschiedenen EJB-Spezifikationen aufsetzen. Die größten Änderungen haben sich dabei in den Bereichen *Unterstützung von EntityBeans*, *XML DeploymentDescriptoren*, *Schaffen eines JNDI Context* und *Sicherheitsfragen* ([MH99b, S.306]) ergeben. So haben sich zum Beispiel die Rückgabewerte der `ejbCreate()`-Methode von Container-Managed und Bean-Managed EntityBeans zwischen den Versionen 1.0 und 1.1 verändert ([MH99b, S.307 ff.]. Weiterhin sind Umgebungsvariablen (*Environment Properties*) seit der Version 1.1 nicht mehr nur auf den Typ `String` begrenzt, sondern können alternativ durch den *Standard JNDI Context* mittels einfacher numerischer Datentypen (`Integer`, `Long`, `Double`, `Boolean`, `Byte` und `Float`) angegeben werden ([MH99a]).

Im Verlauf der Arbeit soll auf die Stellen eingegangen werden, an denen eine Veränderung der Spezifikation eine unmittelbare Auswirkung auf die Portabilität einer EJB-basierten Anwendung hat (s.a. *Rahmenbedingungen*, Kapitel 4.4). Für weitere Informationen zur Entwicklung der EJB-Spezifikation wird an dieser Stelle auf die entsprechende Literatur aus Kapitel 2.2 verwiesen.

4.2.2 HomeInterface

Das *HomeInterface* stellt die Schnittstelle zur Verfügung, über die der Lebenszyklus eines Beans verwaltet werden kann. Um mit einem Bean arbeiten zu können, muß zunächst das *HomeInterface* vom Applikationsserver zurückgeliefert werden. Über dieses stehen dem Client *create*-, *find*- und *remove*-Methoden zur Verfügung, mit deren Hilfe Instanzen eines Beans erzeugt, gefunden und entfernt werden können. Erst nach dem Erzeugen (create) einer Bean-Instanz können die im *RemoteInterface* definierten Geschäftslogik-Methoden angesprochen werden.

4.2.3 RemoteInterface

Das *RemoteInterface* stellt die Schnittstelle zur Verfügung, über die der Client auf Methoden der *BeanClass* zugreifen kann. Dieser Zugriff kann jedoch erst geschehen, nachdem sich der Client über das zuvor beschriebene *HomeInterface* eine Bean-Instanz erzeugt hat.

4.2.4 BeanClass

In der *BeanClass* werden die in den Interfaces definierten Methoden implementiert. Dies sind sowohl die `create`-, `find`- und `remove`-Methoden aus dem *HomeInterface*, als auch die im *RemoteInterface* definierten Methoden, die die eigentliche Geschäftslogik repräsentieren. Wichtig ist, daß es für jede in den *Interfaces* definierte Methode genau eine Implementierung in der *BeanClass* gibt. Der Client greift niemals direkt auf die *BeanClass* zu, sondern immer nur über die vom Container zur Verfügung gestellten Interfaces.

4.2.5 DeploymentDescriptor

Der *DeploymentDescriptor* ist für den Applikationsserver von großer Bedeutung. Er enthält die Informationen über eventuelle Umgebungsvariablen, serverspezifische Informationen oder auch das Verhalten der Beans zur Laufzeit. Auch die Unterscheidung der Beans in EntityBeans und SessionBeans, sowie Informationen bezüglich des Sicherheits-, Transaktions- oder Persistenzmanagements werden im *DeploymentDescriptor* hinterlegt. Dahinter verbirgt sich die grundlegende Idee, das Verhalten der Applikation und die Abstimmung auf die Umgebung (z.B. den Applikationsserver) nur durch Änderungen am *DeploymentDescriptor* zu beeinflussen (vgl. [MH99b, S.30]). Demnach wird eine Veränderung am eigentlichen Quellcode überflüssig.

Während der *DeploymentDescriptor* in der EJB-Spezifikation 1.0 noch als serialisierte Java-Datei vorlag, so sehen die EJB-Spezifikationen 1.1 und 2.0 das XML-Format vor, da dieses vor allem im Bezug auf die Lesbarkeit und bei einer eventuellen Bearbeitung wesentlich besser zu handhaben ist. Für das Arbeiten auf Basis der Spezifikation 1.0 bieten die meisten Server Tools an, mit deren

Hilfe der *DeploymentDescriptor* aus einer einfachen Text-Datei in das für den Server verständliche Format (*.ser) konvertiert werden kann.

4.2.6 EntityBeans

In einer EJB-basierten Applikation werden Geschäftsobjekte, die persistent in der Datenbank gehalten werden sollen, durch *EntityBeans* repräsentiert. Im Kontext der Beispielanwendung lassen sich als Beispiel für solche Geschäftsobjekte bzw. *EntityBeans*, zum Beispiel *Police*, *Produkt* oder auch *Vertrag* nennen. Die Eindeutigkeit innerhalb der Datenbank wird dabei durch den *Primary Key* (Kapitel 4.2.8) gewährleistet. Es gibt zwei Möglichkeiten mit *EntityBeans* umzugehen.

Container-Managed Persistence Bei der Verwendung von *Container-Managed Persistence* (CMP) kann sich der Entwickler ganz auf seine eigentliche Geschäftslogik konzentrieren, da das Persistenzmanagement dem Container des Applikationsservers überlassen wird. Dazu werden im DeploymentDescriptor sowohl die persistenten Attribute, als auch das entsprechende Mapping auf die Datenbank angegeben. Der Vorteil dieser Art des Persistenzmanagements liegt in der einfachen Handhabung. Die *EntityBeans* können unabhängig von der Datenbank entwickelt werden, da das komplette Persistenzmanagement vom Container übernommen wird und der Entwickler somit nicht gezwungen ist, die gesamte Kommunikation mit der Datenbank selbst zu implementieren. Auf der anderen Seite gibt es aber auch einige Nachteile. So ergibt sich durch die Notwendigkeit eines Tools für das Datenbankmapping zwangsweise auch eine Abhängigkeit von diesem entsprechenden Tool. Dieses kann entweder vom Anbieter eines Applikationsservers mitgeliefert, oder auch von Drittanbietern hinzugekauft werden. Je nach verwendetem Mapping-Tool ist es dabei mehr oder weniger möglich, komplexere Datenstrukturen in die Datenbank zu speichern und auch wieder auszulesen. Die Unterstützung von *EntityBeans* ist erst seit der EJB-Spezifikation 1.1 vorgeschrieben, so daß einige Applikationsserver, die auf der Version 1.0 aufsetzen, auch kein *Container-Managed Persistence* unterstützen.

Bean-Managed Persistence Im Gegensatz zum Container-Managed Persistence ist der Entwickler beim *Bean-Managed Persistence* (BMP) gezwungen, die Kommunikation des Beans mit der Datenbank selbst zu implementieren. Auch hier müssen bei der Entwicklung die Vor- und Nachteile sorgfältig gegenüber gestellt werden. So birgt die Verwendung von *Bean-Managed Persistence* ein größeres Fehlerpotential in sich und erschwert die Wartbarkeit des Quellcodes nicht unerheblich. Für Entwickler ohne bzw. mit nur geringen Kenntnissen der Datenbankprogrammierung

(z.B. über JDBC) empfiehlt sich also unter Umständen eher die Verwendung von Container-Managed Persistence. Auf der anderen Seite wird durch *Bean-Managed Persistence* eine Abhängigkeit zu den bereits erwähnten Mapping-Tools bzw. zu dem verwendeten Applikationsserver vermieden, da die entsprechenden Befehle für die Kommunikation mit der Datenbank vom Entwickler selbst implementiert werden.

4.2.7 SessionBeans

Im Gegensatz zu den EntityBeans wird mit *SessionBeans* die eigentliche Geschäftslogik realisiert. Diese greifen grundsätzlich nicht selbst auf die Datenbank zu, sondern nehmen die Anfragen des Client entgegen und leiten diese an die entsprechenden EntityBeans weiter. Bedingt durch ihr nicht-persistentes Verhalten benötigen *SessionBeans* keinen PrimaryKey. Auch die *SessionBeans* werden in zwei Kategorien unterschieden.

Stateful SessionBeans sind stets an einen bestimmten Client gebunden, da sie während der gesamten Zeit ihres Lebenszyklus einen durch globale Variablen bestimmten Zustand beinhalten. Sie werden daher manchmal auch als „erweiterter Client“ bezeichnet. Im Allgemeinen enthalten SessionBeans mehrere Methodenaufrufe, die nacheinander abgearbeitet werden. Bei der Arbeit mit *Stateful SessionBeans* ist es möglich, den mitgeführten Zustand zu verändern und für weitere Methodenaufrufe weiterzuverwenden. In diesem Zusammenhang ist es wichtig zu wissen, daß der Zustand eines solchen SessionBeans nicht persistent in einer Datenbank abgelegt wird. Somit können Fehler oder „Systemabstürze“ dazuführen, daß der Zustand unter Umständen rekonstruiert werden muß.

Stateless SessionBeans sind im Gegensatz zu *Stateful SessionBeans* an keinen Client gebunden. Sie besitzen keine globalen Variablen und somit auch keinen Zustand. Sie sind vergleichbar mit einfachen Batch-Prozessen, bei denen lediglich eine bestimmte Folge von Methoden abgearbeitet wird. Werden innerhalb dieser Methoden bestimmte Informationen benötigt, so müssen diese jeweils von Methode zu Methode mitübergeben werden. Aufgrund dieses einfachen Verhaltens ergeben sich bei *Stateless SessionBeans* keinerlei Probleme bei Fehlern oder „Abstürzen“. Sie können vom Client einfach angefordert werden und nach Abschluß der Methodenaufrufe sofort an den nächsten Client „weitergereicht“ werden.

4.2.8 PrimaryKey

Der *PrimaryKey* taucht nur im Zusammenhang mit EntityBeans auf. Zu jedem EntityBean muß eine *PrimaryKey*-Klasse definiert werden. Wie bereits erwähnt, wird über EntityBeans die Persistenz der Daten realisiert. Die Eindeutigkeit

innerhalb dieser Daten wird dabei durch den *PrimaryKey* gewährleistet. Dieser kann sowohl mittels einfacher Daten (z.B. einem Integer, der oftmals mit `ID` bezeichnet wird), als auch durch eine komplexe Kombination von Daten, dargestellt werden. Wichtig ist jedoch die Eindeutigkeit, um zu einem späteren Zeitpunkt zum Beispiel über die `findByPrimaryKey()`-Methode wieder die korrekten Daten aus der Datenbank lesen zu können.

Die Datenfelder, die in der angegeben werden, müssen alle auch in der BeanClass des zugehörigen EntityBeans enthalten sein. Die PrimaryKey-Klasse muß von `java.lang.Object` erben, so daß primitive Datentypen in einer entsprechenden Wrapper-Klasse gekapselt werden müssen.

4.3 Definition eines Applikationsserver

Eine Definition des Begriffs *Applikationsserver* und einiger weiterer in diesem Zusammenhang notwendigen Begriffe finden sich im Kapitel „Theorie der Applikationsserver“ (Kapitel 6).

4.4 Vereinbarungen

Im bisherigen Verlauf der Diplomarbeit wurden Grundlagen geschaffen, die für das weitere Verständnis notwendig sind. Dabei konnte jedoch nicht immer ein eindeutiger Rahmen für die Behandlung des Themas geschaffen werden. So lassen zum Beispiel die zahlreichen Definitionen des Begriffes *Portabilität* aus Kapitel 3.1 die Frage offen, was eigentlich im Kontext der Diplomarbeit unter diesem Begriff zu verstehen ist. Um diese und weitere Unklarheiten zu vermeiden, sollen nun Vereinbarungen getroffen werden, an denen sich der weitere Ablauf orientiert.

Zunächst wird eine Definition für den Begriff der *Portabilität* angegeben, wie er auch im Kontext der Diplomarbeit verwendet wird:

> „*Portabilität bezeichnet die Übertragbarkeit einer EJB-basierten Anwendung von einer Umgebung auf eine davon abweichende Umgebung. Damit eine Anwendung als portabel bezeichnet werden kann, darf der Aufwand für die Portierung den Aufwand einer Neuentwicklung nicht übertreffen. Der Begriff Umgebung bezeichnet lediglich einen Applikationsserver. Weitere Hard- und Softwareumgebungen (z.B. Betriebssystem, Datenbank, Prozessor) werden vorher als konstant festgelegt. Das Programm muß nach der Portierung den gleichen Funktionsumfang wie vorher besitzen und muß die vorgegebenen Anforderungen aus [Hue00] einhalten.*“

Um den Rahmen der geplanten Portierungen weiter abzugrenzen, werden außerdem folgende Vereinbarungen getroffen:

- Es werden lediglich die Serverkomponenten der Beispielanwendung portiert. Clientseitige Probleme werden nur dann untersucht, wenn sie einen problemlosen Einsatz der Beispielanwendung auf dem neuen Applikationsserver verhindern
- Es wird die relationale Datenbank Oracle 8i der Beispielanwendung ([Hue00]) verwendet, die über JDBC angesprochen wird
- Die weitere Entwicklungsumgebung entspricht den Angaben aus [Hue00, S.107]
- Eine detaillierte Betrachtung von Unterschieden zwischen den einzelnen Spezifikationen wird nur dann vorgenommen, wenn diese Unterschiede zu Portierungsproblemen führen
- Die im Beispiel verwendeten Persistenzmechanismen (CMP oder BMP) sollen im Zuge der Portierung weiterhin erhalten bleiben
- Die Portierung gilt dann als vollzogen, wenn die Beispielanwendung ohne „Altlasten“ des Quellservers auf dem Zielserver ausgeführt werden kann. „Altlasten“ bezeichnen dabei Treiber, Imports oder sonstige Quellen des alten Servers, ohne die die Applikation auf dem neuen Server nicht lauffähig ist.
- Im Anschluß an jede Portierung werden die Beispiele der Diplomarbeit von Malte Hülder zum Testen verwendet. Die dort ermittelten Ergebnisse müssen mit denen der portierten Applikation übereinstimmen.
- Das Nutzen von erweiterten Funktionen des Zielservers ist ebenfalls nicht Ziel der Arbeit. Diverse Server bieten einen sehr komplexen Leistungsumfang für die Arbeit mit EJBs (z.B. den „PowerTier Builder“ von Persistence zum Modellieren von EntityBeans, Erstellen von Relationen zwischen diesen und Generieren von Rahmenquellcode). Ist die Portierung von Server A auf Server B vollzogen und die Anwendung auf dem neuen Applikationsserver lauffähig, so könnten diese erweiterten Funktionen nachträglich verfügbar gemacht werden (Adaption, vgl. Kapitel 3.1.2). Geplant ist lediglich ein Ausblick auf diese Funktionalitäten im Hinblick auf Kapitel 6 (Theorie der Applikationsserver) und Kapitel 10 (Fazit und Ausblick).

Kapitel 5

Die Beispielapplikation

Als Grundlage für die geplanten Portierungen dient der von Malte Hülder in seiner Diplomarbeit entwickelte Prototyp zur „Erfassung der zeitabhängigen Entwicklung von Geschäftsobjekten im Kontext der Historisierung von Versicherungsverträgen mit Enterprise JavaBeans“. Dieses Kapitel beschreibt im folgenden den technischen Hintergrund dieser Beispielapplikation. Für fachliche Fragen wird auf die entsprechenden Stellen aus [Hue00] verwiesen.

Der entwickelte Prototyp realisiert ein Bestandsführungssystem für Versicherungsverträge. Der Benutzer kann Verträge über eine grafische Oberfläche (GUI) anlegen und verwalten. Dabei wird zu den einzelnen Versicherungsverträgen eine Historie mitgeführt, so daß der Anwender sich jederzeit über alle Zustände informieren kann, die der Vertrag seit Vertragsabschluß besessen hat.

Die Entwicklung und Umsetzung des gesamten Prototyps konzentrierte sich auf folgende Bereiche:

- Die gesamte Geschäftslogik wird durch EJBs repräsentiert
- Der Prototyp muß das erarbeitete Konzept zur Historisierung von Versicherungsverträgen umsetzen
- Die Entwicklung richtet sich nach der EJB-Spezifikation 1.0
- Abhängigkeiten vom verwendeten Applikationsserver sollen möglichst vermieden werden

5.1 Entwicklungsumgebung

Der Prototyp der Beispielapplikation wurde unter der folgenden Entwicklungsumgebung entwickelt (vgl. [Hue00, S.107]):

- Betriebssystem Microsoft Windows NT 4.0
- JDK 1.2.2

- Borland JBuilder 3
- Oracle 8i Datenbank
- Applikationserver BEA Weblogic 4.5.1.

5.2 Die Schicht-Architektur

In Kapitel 3.3 wurde bereits die 3-Schicht-Architektur beschrieben, die als Grundlage für die EJB-Architektur verwendet wird. Auch der Prototyp setzt auf dieser Architektur auf. So ist der Client (Presentationlayer) als „Thin-Client“ realisiert worden. Das bedeutet, daß dort möglichst wenig Geschäftslogik implementiert wurde. Der Client stellt also lediglich eine grafische Oberfläche zur Verfügung (Abbildung 5.1), über die der Benutzer seine Eingaben tätigen kann und die anschließend zur Darstellung der Daten verwendet wird. Der Vorteil besteht darin, daß bei eventuellen Änderungen innerhalb der Geschäftslogik lediglich auf Server-Ebene Änderungen vorgenommen werden müssen und nicht bei jedem einzelnen Client. Desweiteren kann der Zugriff auf die Datenbank zentral koordiniert werden und muß nicht über komplizierte Mechanismen gesteuert werden.

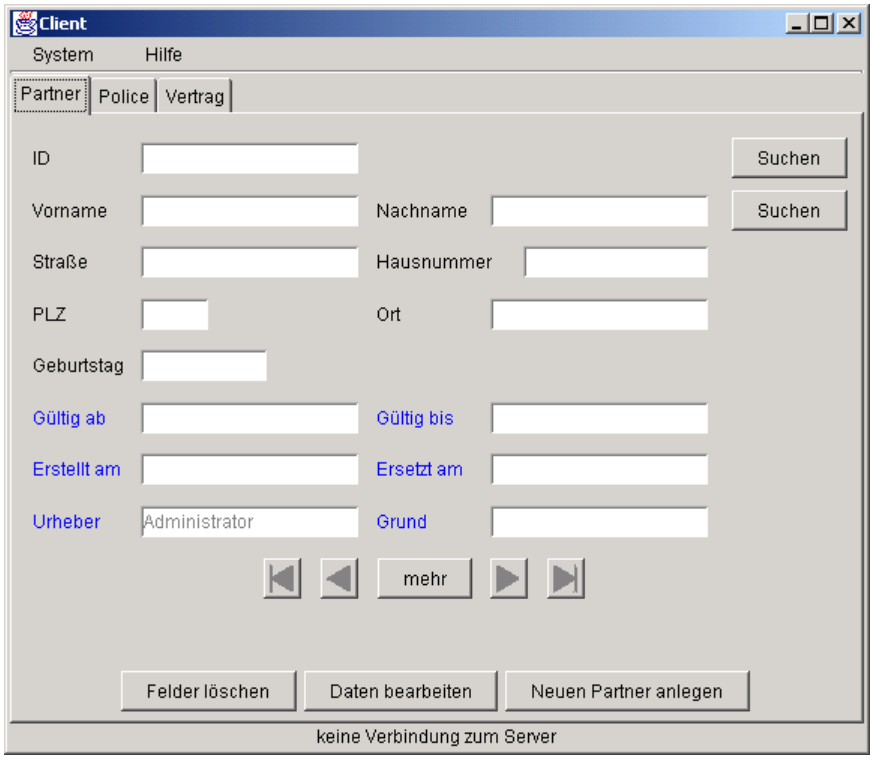

Abbildung 5.1: Client nach dem Start

Auf der *Datenschicht* (Datalayer) werden die persistenten Daten in einer Oracle-Datenbank verwaltet. Der Zugriff auf diese Daten erfolgt niemals direkt durch den Client, sondern immer nur über EntityBeans, die sich auf der sogenannten *Middletier*, also der Schicht zwischen Client und Datenbank, befinden. Hier werden sowohl die Anfragen des Client bearbeitet, als auch die Kommunikation mit der Datenbank abgewickelt.

Die Reaktion auf Eingaben des Benutzers in der GUI erfolgt in EJB-basierten Anwendungen stets nach folgendem Schema:

- Der Client stößt ein oder mehrere SessionBeans auf dem Server an, welche die eigentliche Geschäftslogik repräsentieren.
- Um auf persistente Daten zuzugreifen, werden die entsprechenden Anfragen von den SessionBeans an die EntityBeans übergeben.
- EntityBeans sind für die eigentliche Datenbankkommunikation verantwortlich. Sie stellen Anfragen an die Datenbank und liefern die Ergebnisse dieser Anfrage an die SessionBeans zurück. Diese wiederum reichen die Ergebnisse schließlich an die GUI weiter, wo sie dem Client präsentiert werden.

Stellt man diesen gesamten Ablauf grafisch dar, so erhält man Abbildung 5.2.

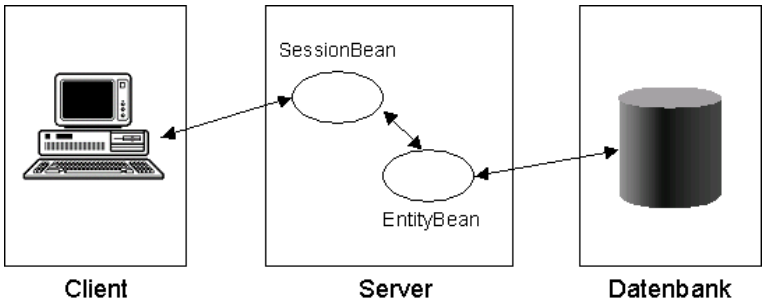

Abbildung 5.2: Kommunikation zwischen den Schichten

Dieses Grundschema wird von Malte Hülder in seiner Arbeit noch weiter spezifiziert, indem er die Mittelschicht ebenfalls in drei Schichten unterteilt.

Die erste Schicht der Mittelschicht bildet die sogenannte Verteilerschicht. Sie enthält lediglich das SessionBean *VerteilerBean*, welches als einziges vom Client angesprochen wird. Dessen Anfragen leitet das *VerteilerBean* an die zweite Schicht weiter, die die SessionBeans mit der eigentlichen Geschäftslogik enthält. In der dritten und letzten Schicht finden sich schließlich die EntityBeans. Über sie werden die Datenbankanfragen der SessionBeans realisiert. Insgesamt kommuniziert der Client also nur mit einem einzigen SessionBean, welches dann die weiteren Aufgaben auf dem Server an entsprechende weitere SessionBeans

verteilt. Eine grafische Darstellung dieses erweiterten Schichtenmodells findet sich in Abbildung 5.3.

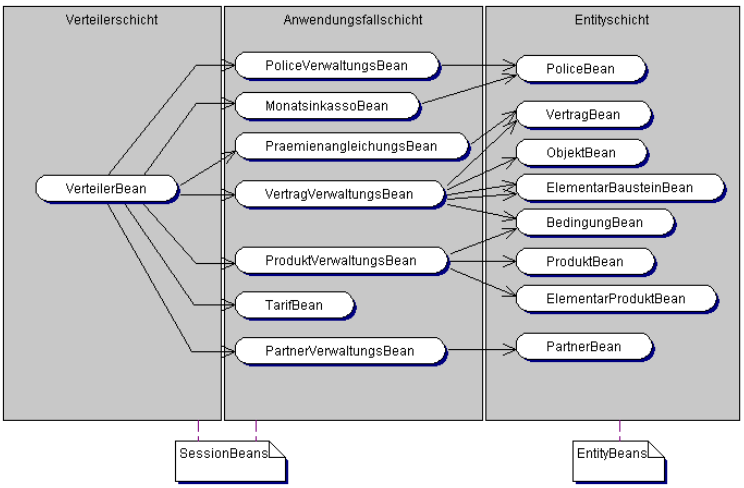

Abbildung 5.3: Verfeinerte Mittelschicht der Beispielapplikation

5.3 Klassendiagramm und Komponentenschnitt

In der komponentenbasierten Softwareentwicklung ist darauf zu achten, daß innerhalb einer Komponente eine starke und zwischen den Komponenten eine eher lose Bindung gegeben ist. Da Komponenten eine Sammlung von Klassen repräsentieren bedeutet dies, daß in einer Komponente ein starker Informationsaustausch zwischen den beteiligten Klassen gegeben ist. Der Austausch zwischen den Komponenten hingegen ist eher gering (lose Bindung). Dies hat zur Folge, daß die Komponenten besser ausgetauscht und wiederverwendet werden können ([Gru99]). Der aus dem Klassendiagramm entstandene Komponentenschnitt der Beispielapplikation spiegelt diese Bindungsformen sehr gut wieder. Abbildung 5.4 zeigt das komplette Klassendiagramm aus [Hue00]. Aus diesem entstand dann der in Abbildung 5.5 gezeigte Komponentenschnitt. Für eine detaillierte Beschreibung der einzelnen Klassen bzw. Komponenten aus fachlicher Sicht wird auch hier wieder auf die entsprechenden Stellen der Diplomarbeit von Malte Hülder verwiesen. Eine technische Beschreibung einzelner Teile findet sich in Kapitel 5.4.

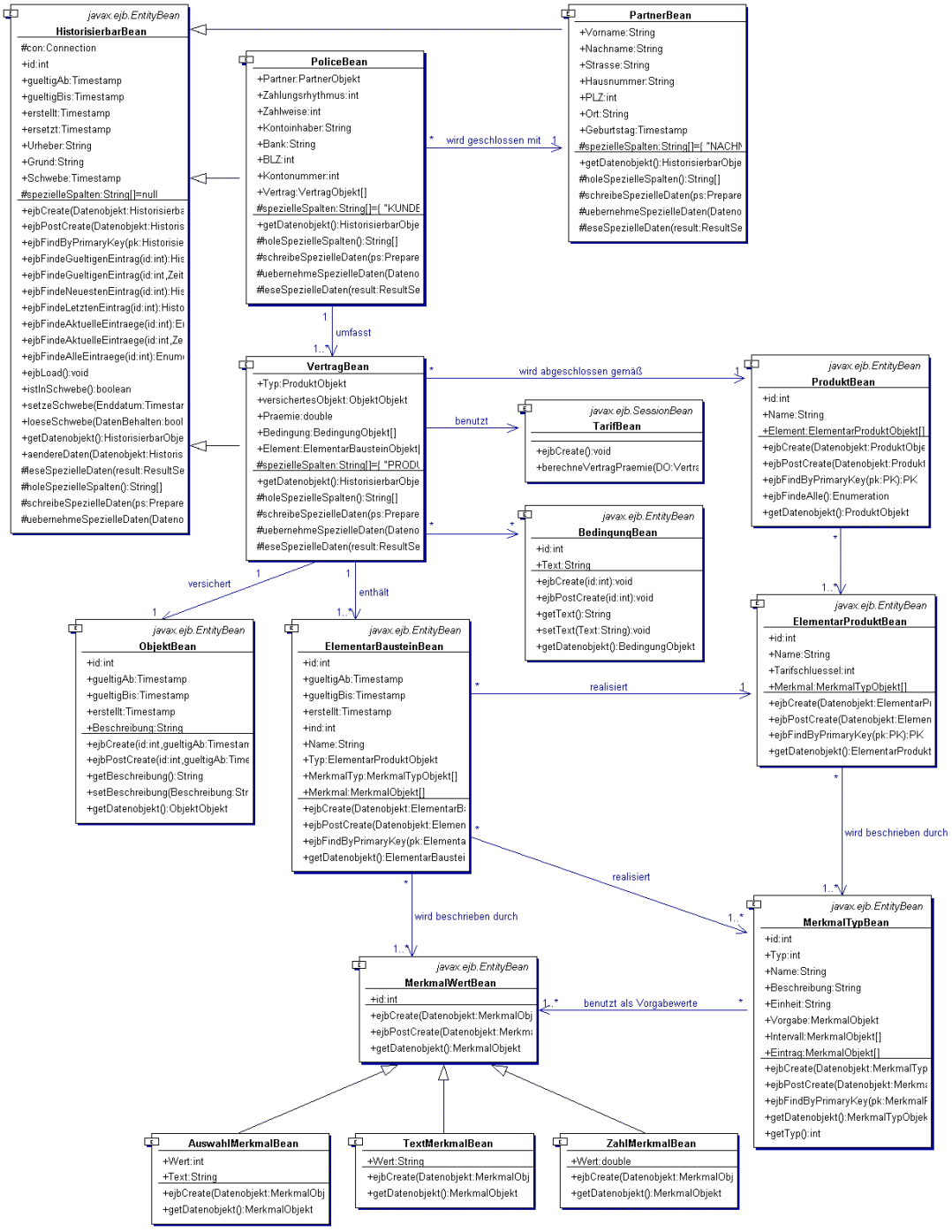

Abbildung 5.4: Klassendiagramm der Beispielapplikation

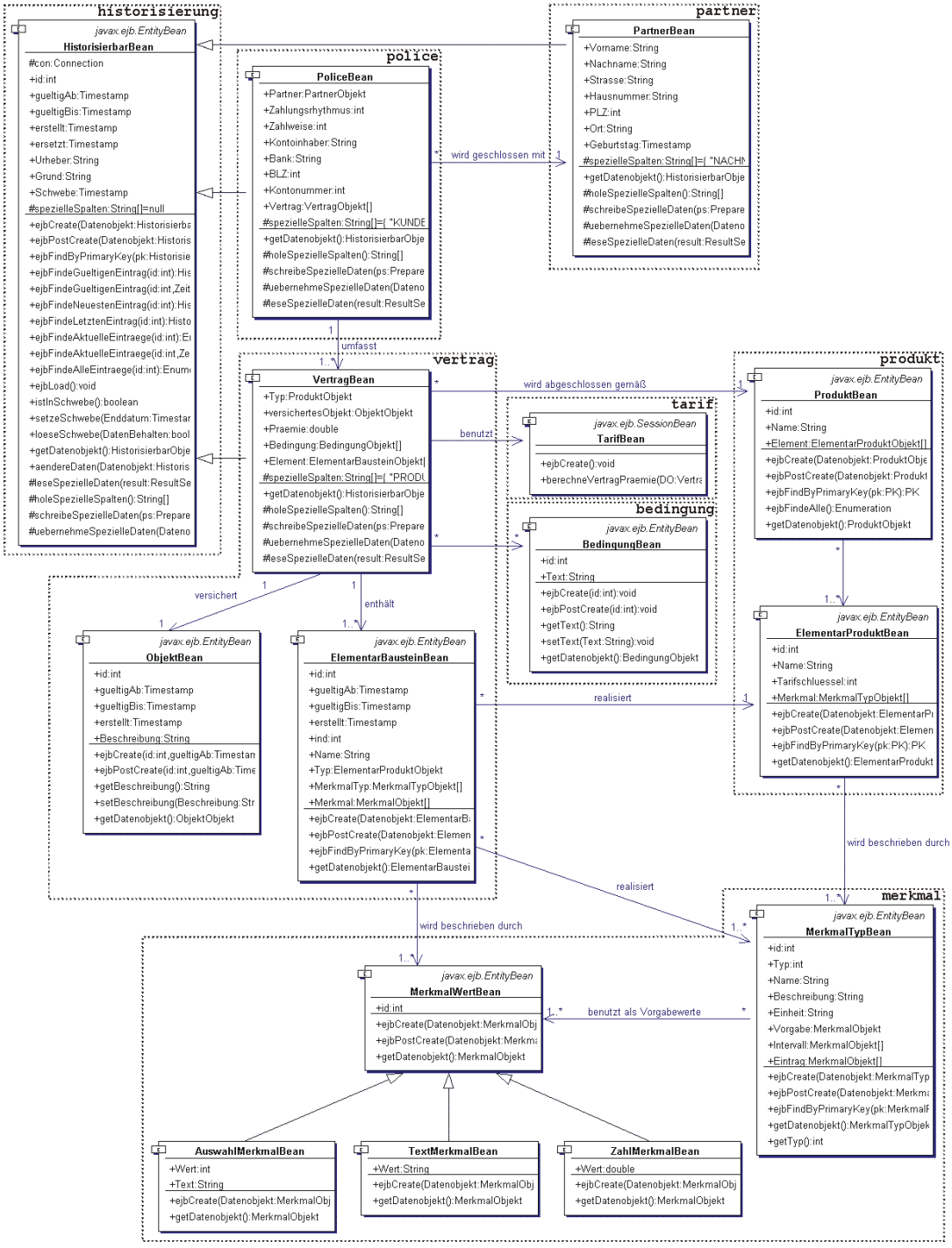

Abbildung 5.5: Komponentenschnitt der Beispielapplikation

5.4 Technische Eigenschaften

Zu guter Letzt geht dieser Abschnitt auf einige technische Eigenschaften der Beispielapplikation ein, die bei der Portierung besonders zu beachten sind, da sie unter Umständen zu Problemen größeren Ausmaßes führen können. Dabei ist stets daran zu denken, daß sich Malte Hülder bei der Entwicklung seines Prototypen an den Vorgaben der EJB-Spezifikation 1.0 orientiert und eine Verwendung von WebLogic-spezifischen Eigenschaften auf ein Minimum reduziert hat.

Das Persistenzmanagement wird in der Beispielapplikation hauptsächlich Bean-Managed realisiert. Lediglich in den Beans *BedingungBean* (Komponente `bedingung`), *ObjektBean* (Komponente `vertrag`) und *MerkmalWertBean* (Komponente `merkmal`) wird die Persistenz der Daten vom Container des Applikationsservers umgesetzt. Da die Rahmenbedingungen der Diplomarbeit in Kapitel 4.4 vorsehen, daß der jeweils verwendete Persistenzmechanismus auch im Zuge der Portierung erhalten bleibt, stellt sich natürlich die Frage, ob diese Bedingung überhaupt eingehalten werden kann. Ein Wechsel des Persistenzmechanismus würde sicherlich weitere Fragen und Probleme aufwerfen. Zur Zeit kann jedoch über dieses Problem nur spekuliert werden. Es bleibt also zunächst abzuwarten, ob ein Wechsel unter Umständen notwendig ist, und wie er sich auf die Portierung auswirkt.

Das erarbeitete Historisierungskonzept wird von Malte Hülder in der Beispielapplikation in den Komponenten `historisierung`, `partner`, `police` und `vertrag` umgesetzt. Dabei stellen die Komponenten `partner`, `police` und `vertrag` Methoden zur Verfügung die es erlauben, Versicherungsnehmer (Partner), Policen (Police) und die darin enthaltenen Versicherungsverträge (Vertrag) zu verwalten. Um nun die Zustände, die zum Beispiel ein Versicherungsvertrag seit dem ersten Abschluß besessen hat, nachvollziehen zu können, werden die einzelnen Vertragsstände durch die speziellen Historisierungsattribute `gueltigAb`, `gueltigBis`, `erstellt` und `ersetzt` eindeutig einem bestimmten Gültigkeitszeitraum zugeordnet. Da dieses Historisierungskonzept für die Komponenten `partner`, `police` und `vertrag` identisch ist, wurde die Komponente `historisierung` entwickelt, die die genannten Historisierungsattribute und allgemeine Methoden zur Historisierung enthält. Die drei anderen Komponenten erben diese „Historisierungsfähigkeit“ und implementieren selbst nur die speziellen, für die jeweilige Komponente notwendigen Methoden.

Eine weitere Vererbung findet sich in der Komponente `merkmal`. Dort erben die speziellen Beans `AuswahlMerkmalBean`, `TextMerkmalBean` und `ZahlMerkmalBean` vom `MerkmalWertBean`. Wichtig ist hier der

Unterschied, daß es sich bei der „Merkmal-Vererbung“ um eine komponenteninterne und bei der „Historisierungs-Vererbung“ um eine komponentenübergreifende Vererbung handelt. Im Komponentenschnitt (Kapitel 5.5) lassen sich diese Vererbungen sehr gut nachvollziehen. Inwiefern sich diese Vererbungen jedoch auf die Portierung auswirken, ist an dieser Stelle noch nicht abzuschätzen.

Eine Modularisierung wie sie in der Literatur zum Beispiel Brown ([Bro77, S.119 ff.]) im Kapitel „Techniques for Enhancing Portability“ vorschlägt, wird auch in der Beispielapplikation ansatzweise umgesetzt. Bei dieser Technik wird die Anwendung in spezifische und unspezifische Komponenten bzw. Module aufgeteilt. Dadurch soll erreicht werden, daß die unspezifischen Komponenten zu einem späteren Zeitpunkt schnell und unkompliziert portiert werden können, während die spezifischen Komponenten entweder durch neue spezifische Komponenten ersetzt oder entsprechend an die neuen Spezifika angeglichen werden. Im Kontext der Diplomarbeit würde dies bedeuten, daß es Komponenten gibt, die unabhängig vom verwendeten Applikationsserver sind und somit ohne große Probleme portiert werden können (→ unspezifisch), als auch Komponenten, die wirklich nur serverspezifische Funktionen enthalten und somit nicht unbedingt trivial zu portieren sind (→ spezifisch). Die Beispielapplikation geht diesem Ansatz mit der Komponente `vendor` nach. Diese sieht wie folgt aus:

```
package vendor;

import java.util.Properties;
import javax.naming.Context;

public class vendor {
    public static Context getInitialContext(String URL)
       throws javax.naming.NamingException{

        Properties p = new Properties();
        // specify the JNDI properties specific to the vendor
        p.put (Context.INITIAL_CONTEXT_FACTORY,
               ''weblogic.jndi.WLInitialContextFactory'');
        p.put (Context.PROVIDER_URL, URL);
        //
        return new javax.naming.InitialContext(p);
    }
}
```

Mittels dieser Komponente erhält der Anwender die Möglichkeit, bestimmte Umgebungsvariablen (`Properties`) zu setzen, die innerhalb der EJB-Anwendung verwendet werden können. In diesem Fall handelt es sich um JNDI-Umgebungsvariablen, die speziell für den WebLogic notwendig sind. Denkbar sind hier auch andere WebLogic-spezifische Variablen oder sogar weitere, serverspezifische Komponenten. Insgesamt erscheint diese Aufsplittung in spezifische und unspezifische Komponenten als sehr sinnvoll, da somit der Vorgang einer Portierung nicht nur übersichtlicher, sondern auch einfacher wird.

Kapitel 6

Theorie der Applikationsserver

Nachdem in den vorherigen Kapiteln Grundlagen zu den Themen „Portabilität“ und „Enterprise JavaBeans“ geschaffen wurden, beschäftigt sich dieses Kapitel ausschließlich mit *Applikationsservern*. Dabei werden zunächst auch für diesen Bereich Grundlagen geschaffen, indem beschrieben wird, was *Applikationsserver* eigentlich sind und welche Aufgaben sie vor allem bei der Arbeit mit Enterprise JavaBeans übernehmen. Im Anschluß daran werden die im Zuge der Portierung verwendeten Applikationsserver unter zwei Geschichtspunkten vorgestellt und untersucht. Zum einen soll der jeweilige Leistungsumfang des Servers beschrieben werden, der für die Arbeit mit EJBs zur Verfügung steht. Diese Leistungen reichen von mitgelieferten Tools aller Art über serverspezifische Datenstrukturen bis hin zu umfangreichen Entwicklungsumgebungen. Zum anderen sollen anhand der Serverbeschreibungen Probleme ermittelt werden, die bei den geplanten Portierungen zu erwarten sind. Im Anschluß an die eigentlichen Portierungen in den Kapiteln 7 und 8 werden die in diesem Kapitel erarbeiteten Thesen mit den Ergebnissen der Portierungen verglichen. Aus diesen Vergleichen und den erarbeiteten Problemen und Lösungen soll dann ein Resümee bezüglich der *Portierung von EJB-basierten Anwendungen* gezogen werden.

6.1 Grundlagen

Wie schon in Kapitel 3.4 kurz erwähnt, ist unter dem Begriff *Applikationsserver* ein Server-Programm auf einem Rechner in einer verteilten Umgebung zu verstehen. Der Server stellt dabei für bestimmte Applikationen bzw. Dienste eine Infrastruktur zur Verfügung, auf der diese aufsetzen können. Die hier verwendeten Server unterstützen vor allem die *Java 2 Enterprise Edition* (J2EE) Plattform. Diese besteht aus einer Menge von Diensten, APIs und Protokollen, mit deren Hilfe es möglich ist, mehrschichtige, webbasierte Anwendungen zu erstellen und auszuführen (vgl. [Jee01]). Die Enterprise JavaBeans bilden einen Kernbereich dieser Plattform.

Für die Arbeit mit der EJB-Technologie muß der Server unter anderem einen

oder mehrere sogenannte *Container* bereitstellen. Diese haben für die Beans den Charakter einer Laufzeitumgebung in dem Sinne, daß bestimmte Dienste zur Verfügung gestellt werden, wie zum Beispiel zur Namensgebung (zum Auffinden von Objekten) oder aber auch zum Transaktions- und Sicherheitsmanagement. Dahinter verbirgt sich die Idee, daß bei der Entwicklung von EJBs diese Schnittstellen und Dienste somit nicht noch erst von Hand programmiert werden müssen, sondern als gegeben angenommen werden können. Dadurch soll es dem Entwickler möglich sein, sich mehr auf die eigentliche Geschäftslogik zu konzentrieren.

Von technischer Seite wird eine EJB-Anwendung stets auf dieselbe Art und Weise installiert. Der Entwickler implementiert zunächst sogenannte Packages bzw. Komponenten, die jeweils aus einer oder mehreren Beans bestehen. Anschließend werden zu jedem einzelnen Bean DeploymentDescriptoren erzeugt, die für den Server notwendige Informationen enthalten. Diese DeploymentDescriptoren und die zugehörigen Beans werden dann in einem JAR-Archiv pro Komponente zusammengefaßt und im Server installiert (deployed).

6.2 WebLogic

Bei dem ersten Applikationsserver, der in diesem Kapitel beschrieben wird, handelt es sich um den *WebLogic* der Firma *BEA*. Dieser Server wurde in der Version 4.5.1 als Grundlage für die zu portierende Beispielapplikation von Malte Hülder verwendet und setzt auf der EJB-Spezifikation 1.0 auf. Mittlerweile bietet *BEA* bereits die Version 6.1 an, die die EJB-Spezifikation 2.0 unterstützt.

Grundsätzlich stellt der *WebLogic* nicht nur eine Infrastruktur für EJB-basierte Anwendungen zur Verfügung, sondern für J2EE-Anwendungen (z.B. JSP) im Allgemeinen. Der Umfang für die Arbeit mit EJBs beschränkt sich im Wesentlichen auf folgende Leistungen:

- *ejbc* - Dieses Tool wird zum Kompilieren der Beans und zum Erzeugen der Stubs und Skeletons verwendet.
- *DDCreator* - Mit diesem Tool werden die DeploymentDescriptoren der einzelnen Beans erzeugt. Als Vorlage dient dabei jeweils eine Textdatei, deren Aufbau in der Dokumentation des *WebLogic* beschrieben wird ([Web01a, Developers Guide, S.60]).
- *ComplianceChecker* - Dieser testet ein *WebLogic*-EJB hinsichtlich der korrekten Umsetzung der Spezifikationsvorgaben.
- *EJB Deployment Wizard* - Mit Hilfe dieses Wizards können die Beans im Server installiert und konfiguriert werden.
- Server- und Container APIs - Über diese APIs kann der Entwickler auf spezifische Leistungen des Servers zugreifen.

Die Beschreibung dieser Leistungen soll an dieser Stelle aus verschiedenen Gründen nicht weiter vertieft werden. Zum einen wurde bei der Entwicklung der Beispielapplikation ganz bewußt auf den Einsatz der serverspezifischen APIs verzichtet. Lediglich zum Aufbau einer Datenbankverbindung wurde in den DeploymentDescriptoren für die benötigte URL der *WebLogic*-spezifische Wert `jdbc:weblogic:oracle` eingetragen. Zum anderen üben die genannten Tools keinen Einfluß auf den Quellcode aus und werden im Fall einer Portierung nicht benötigt. Des weiteren sind die Beschreibungen der Server auf die die Beispielapplikation zu portieren ist wichtiger, da in diesem Zusammenhang die Thesen hinsichtlich späterer Portierungsprobleme ausgearbeitet werden sollen.

Der Server selbst läßt sich zentral über eine grafische Oberfläche verwalten, die in Abbildung 6.1 dargestellt ist.

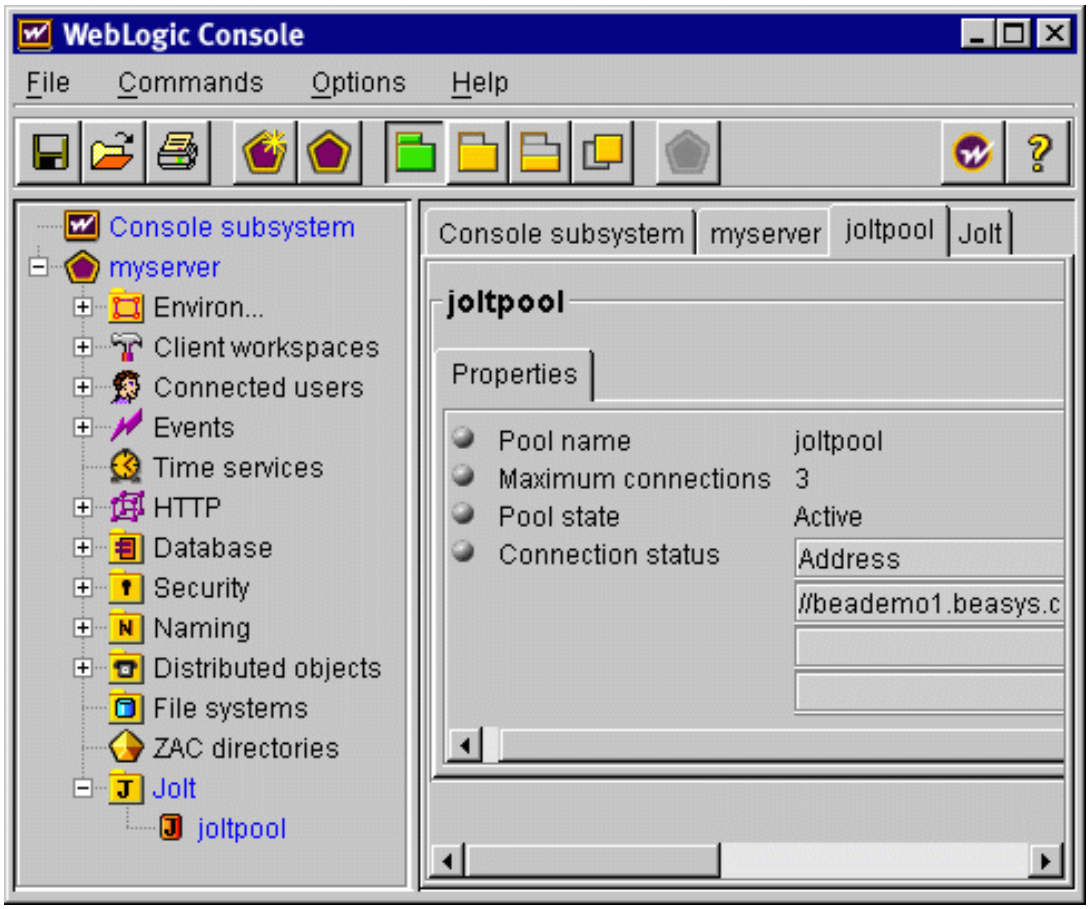

Abbildung 6.1: Die WebLogic Administrationskonsole

6.3 Powertier

Die Firma *Persistence* stellt mit dem *PowerTier* einen Applikationsserver zur Verfügung, der sich ganz auf die Unterstützung der EJB-Technologie konzentriert. Im Rahmen der Diplomarbeit wird für die Portierung die Version 6.00 verwendet. Der Nachfolger in der Version 6.5 wurde im Dezember 2000 veröffentlicht, wobei auf entsprechende Verbesserungen bzw. Neuigkeiten nur im Kapitel 10 (Fazit und Ausblick) eingegangen wird. Dem Entwickler selbst stehen für die Arbeit mit EJBs zahlreiche Tools zur Verfügung. Dabei handelt es sich sowohl um Tools mit einer grafischen Benutzeroberfläche, als auch um diverse Kommandozeilenbefehle. Im Vergleich zum WebLogic fällt die Beschreibung dieses Servers aufgrund des größeren Leistungsumfangs etwas umfangreicher aus.

Bei der nachfolgenden Beschreibung geht es vor allem um den Umgang mit EJBs und den möglichen Einfluß der enthaltenen Tools auf eventuelle Portierungen von EJB-basierten Anwendungen.

6.3.1 Allgemeines

Die Grundlage für die verwendete Version des PowerTier bildet die EJB-Spezifikation 1.1. Aufgrund der zahlreichen Änderungen, die sich im Vergleich zu Version 1.0 (WebLogic) ergeben haben, ist somit bereits an dieser Stelle mit Portierungsproblemen zu rechnen, deren Ursache in genau diesen Änderungen zu suchen ist. Beweise hierfür finden sich in der Literatur zum Beispiel bei Monson-Haefel ([MH99b, S.306 ff.]) und Merkle ([Mer00a]), die sich in ihren Berichten mit den Unterschieden der EJB-Spezifikation 1.0 und 1.1 beschäftigt haben. So werden die Änderungen, die sich im Bereich `JNDI` ergeben haben, vermutlich alle Komponenten betreffen und die Änderungen aus dem Bereich Container-Managed Persistence die Komponenten `bedingung`, `merkmal` und `vertrag`. Dort wurden zum Beispiel die Rückgabewerte der `ejbCreate()`-Methoden der BeanClasses geändert. Anstatt eines `null`-Rückgabewertes wird seit der EJB-Spezifikation 1.1 der PrimaryKey des jeweiligen Beans zurückgegeben. Ob noch weitere Anpassungen an die neue Spezifikation notwendig sind bleibt zunächst abzuwarten.

6.3.2 Die Pantry

Die *Pantry* ist ein Verzeichnis, in dem der PowerTier-Server verschiedene Dateien ablegt. Es handelt sich hierbei um `.class`-, `.ser`- und `.jar`-Dateien, welche bei der Arbeit mit dem Befehl `ps-makeejb` (s. Kapitel 6.3.6) erzeugt werden. Für die Portierung ist die *Pantry* jedoch von keinerlei Bedeutung.

6.3.3 Profile Wizard

Mit dem *Profile Wizard* liefert Persistence dem Entwickler ein Tool, über welches die grundlegenden Einstellungen für die gesamte Arbeit mit dem PowerTier-Applikationsserver vorgenommen und getestet werden können. So ist es dem Benutzer möglich zu testen, ob der installierte Server eine Verbindung zur Datenbank aufbauen kann oder ob die notwendige Java-Entwicklungsumgebung korrekt eingerichtet wurde. Aus diesen Informationen wird ein Profil erstellt, welches beispielsweise unter Windows in der Registry gespeichert wird und anschließend wieder abgerufen und gegebenenfalls verändert werden kann. Bei Änderungen am System kann anschließend wieder über den *Profile Wizard* getestet werden, ob der Server ohne Probleme weiterarbeitet. Dabei ist zu beachten, daß eventuelle Probleme lediglich angezeigt und nicht über den *Profile Wizard* gelöst werden können. Abbildung 6.2 zeigt den *Profile Wizard* unmittelbar nach dem Start.

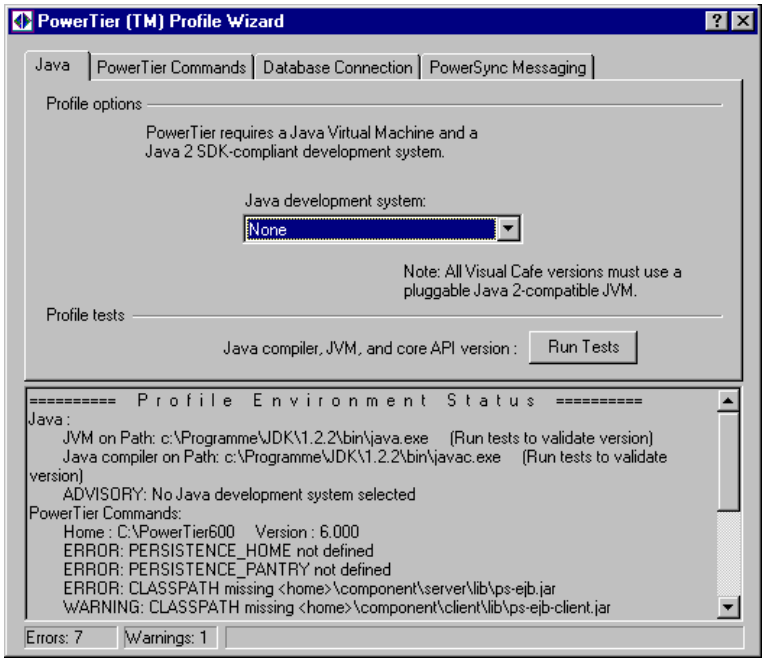

Abbildung 6.2: Der Profile Wizard

Insgesamt stehen die Kategorien *Java*, *PowerTier Commands*, *Database Connection* und *PowerSync Messaging* zur Auswahl. Alle Einstellungen, die in diesen Kategorien getätigt werden, können mittels eines „Test“-Buttons auf ihre Korrektheit hin überprüft werden. Hierbei öffnet sich jeweils kurz eine DOS-Box, in der die Tests durchgeführt und anschließend die Ergebnisse angezeigt werden. Der Status aller Kategorien wird die ganze Zeit über im unteren Fensterabschnitt (*Profile Environment Status*) aufgeführt. Im Folgenden sollen nun die gennanten

vier Kategorien kurz erläutert werden.

Unter dem ersten Reiter (*Java*, Abbildung 6.3) verbirgt sich eine Auswahlliste möglicher Entwicklungssysteme, die *Java 2 SDK*-kompatibel sind und eine „Virtual Machine“ (VM) zur Verfügung stellen. Der zweite Reiter (*PowerTier Commands*, Abbildung 6.4) erlaubt die Umgebungsvariablen einzustellen, die beim PowerTier-Commando-Prompt automatisch gesetzt werden sollen und anzugeben, in welchem Verzeichnis sich die Pantry befindet. Eventuelle Datenbankverbindungen können im dritten Reiter (*Database Connection*, Abbildung 6.5) getestet werden. Innerhalb des vierten und letzten Reiters schließlich besteht die Möglichkeit, das *PowerSync Messaging* zu testen (*PowerSync Messaging*, Abbildung 6.6). Dieses erlaubt dem Entwickler, einen „Shared Cache“ über mehrere Server hinweg zu synchronisieren.

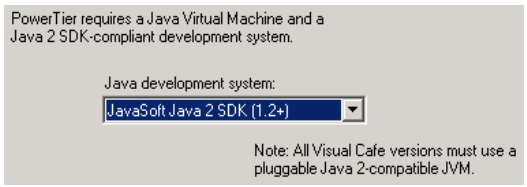

Abbildung 6.3: Auswahl der Java-Entwicklungsumgebung

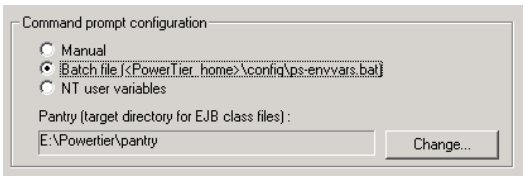

Abbildung 6.4: Einstellung des Commando-Prompts

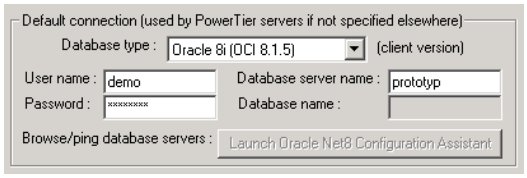

Abbildung 6.5: Auswahl der Datenbankanbindung

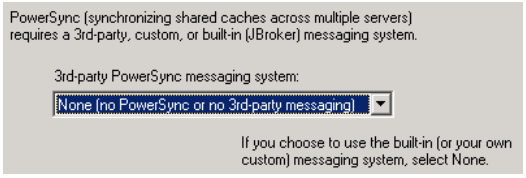

Abbildung 6.6: Das PowerSync Messaging

Mögliche Fehlermeldungen und entsprechende Lösungsansätze innerhalb der vier Kategorien des *Profile Wizard* beschreibt *Persistence* im Kapitel 7.3 (TroubleShooting) in [Per01, PowerTier-Quickstart]. Einen Einfluß auf die Portabilität von EJB-basierten Anwendungen hat dieses Tool jedoch nicht, da es zu keiner Zeit Veränderungen an der eigentlichen Implementierung oder an der Umgebung vornimmt. Es hilft lediglich dabei, die Anbindung des Servers an seine Umgebung auf mögliche Fehler hin zu untersuchen.

6.3.4 PowerTier Builder

Wie schon in Kapitel 4.2 beschrieben, werden in einer EJB-basierten Applikation die Geschäftsprozesse in den SessionBeans und die persistenten Geschäftsobjekte in den EntityBeans umgesetzt. Mit dem *PowerTier Builder* (Abb. 6.7) erhält

der Entwickler die Möglichkeit, EntityBeans und Relationen zwischen diesen zu modellieren. Die Modellierung von SessionBeans aufgrund von „UseCase“- oder „Activity“-Diagrammen ist erst für zukünftige Versionen vorgesehen.

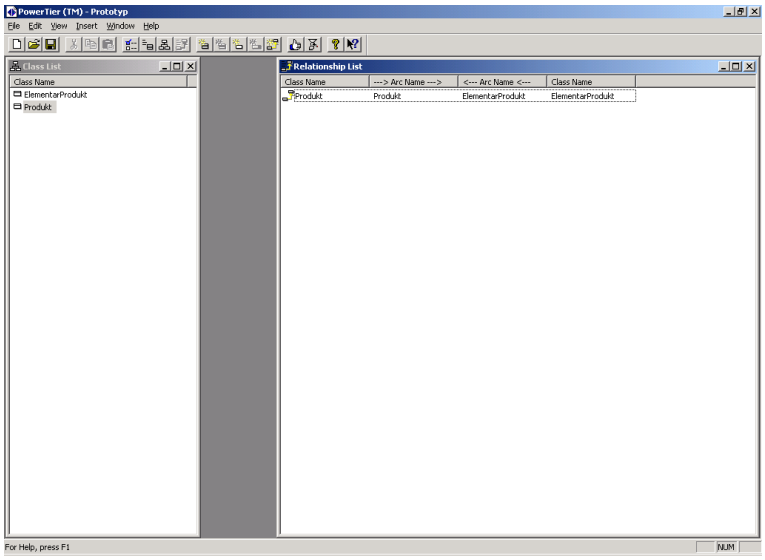

Abbildung 6.7: Der PowerTier Builder

Der *PowerTier Builder* kann aus dem Modell der EntityBeans Rahmenquellcode generieren (Home- und RemoteInterface, BeanClass, PrimaryKey), der anschließend wiederum vom Entwickler angepasst werden kann. Als Alternative zum *PowerTier Builder* ist es auch möglich, den Rahmenquellcode über den Kommandozeilenbefehl `ps-gen` generieren zu lassen. Grundlage ist stets das Modell der EntityBeans, das in der `.per`-Datei (Kapitel 6.3.7) abgelegt wird. In beiden Fällen ist es sehr wichtig zu wissen, daß bei der Arbeit mit dem *PowerTier Builder* nur Container-Managed EntityBeans erzeugt werden. Dies erscheint auf den ersten Blick zunächst auch logisch und wenig problematisch. Bei näherer Betrachtung fallen jedoch sofort gewisse Probleme auf, die besonders für eine spätere Portierung von Interesse sind. So finden sich in dem vom *PowerTier Builder* generierten Quellcode sehr viele PowerTier-spezifische Kommandos. Diese führen im Fall der späteren Portierung auf einen anderen Server zwangsläufig zu Portierungsproblemen, da diese Kommandos dort nicht zur Verfügung stehen. Unter Umständen können solche Probleme auf folgende Arten gelöst werden:

- Dem neuen Server werden die benötigten spezifischen Bestandteile (z.B. Imports) hinzugefügt. Dies kann jedoch durchaus zu zwei neuen Problemen führen. Zum einen müssen die Lizenzrechte gesichert sein und zum anderen enthält die portierte Applikation nunmehr serverspezifische Bestandteile von zwei Applikationsservern. Im Falle einer weiteren, zweiten Portierung ergeben sich somit also noch mehr Probleme.

- Die PowerTier-spezifischen Bestandteile werden aus der Applikation entfernt und vom Entwickler durch eigene „neutrale“ Teile ersetzt. Der Begriff „neutral“ ist dabei als allgemein und von jeglichem Server unabhängig zu verstehen. Aber auch diese Art des Vorgehens wirft Fragen auf. Wie aufwendig ist diese Vorgehensweise? Ist es dem Entwickler überhaupt möglich, den generierten Quellcode einfach umzuschreiben oder wäre eine Implementierung ohne den PowerTier Builder unter Umständen nicht sogar sinnvoller? Auf jeden Fall setzt die Bearbeitung des vom PowerTier Builder generierten Quellcodes sehr gute Kenntnisse des PowerTier-Servers und insbesondere seiner mitgelieferten APIs (vgl. Kapitel 6.3.8) voraus.

Teilweise setzt Persistence in seiner mitgelieferten Dokumentation solchen Überlegungen schnell ein Ende. So wird im *Quickstart Tutorial* eine Portabilität von „PowerTier Container-Managed EntityBeans“ von Beginn an ausgeschlossen:

> „*A bean that manages its own persistent objects is portable across all containers that are EJB-compliant, while PowerTier container-managed entity beans only work in PowerTier containers.*“
>
> ([Per01, Quickstart Tutorial, S.96])

Dies verdeutlicht, daß der Einsatz solcher Tools mit Hinblick auf eine eventuelle Portierung nicht ganz unproblematisch, wenn nicht sogar unmöglich ist. Im Folgenden sollen diese und weitere Probleme noch etwas detaillierter untersucht werden.

Der vom *PowerTier Builder* generierte Code wird durch PowerTier-spezifische Kommentare in einzelne Abschnitte unterteilt. Ein Beispiel für einen solchen Abschnitt stellen die folgenden drei Zeilen dar:

```
//BEGIN ---------------------- PS(Untitled,MyClass,bean_imports)
// Custom imports.
//END   ---------------------- PS(Untitled,MyClass,bean_imports)
```

Die erste Zeile leitet dabei einen solchen speziellen Abschnitt ein, während die dritte Zeile ihn wieder schließt. Den eigentlich wichtigen Teil jedoch stellt die mittlere Zeile dar. Je nachdem an welcher Stelle im Quellcode ein solcher Kommentar vorkommt, kann die Zeile `// Custom imports.` durch speziellen Quellcode des Entwicklers ersetzt werden. Das oben gezeigte Beispiel stammt aus dem `import`-Bereich einer Beanclass `MyClass`. An dieser Stelle können Imports hinzugefügt werden, die im Kontext dieser Beanclass benötigt werden, vom *PowerTier Builder* jedoch nicht eingefügt wurden. Hintergrund dieser speziellen Kommentare ist das folgende Prinzip:

Sind an einem mit dem *PowerTier Builder* erstellten Modell der EntityBeans Änderungen notwendig, so werden diese zunächst im Modell selbst umgesetzt.

Anschließend wird erneut der Rahmenquellcode generiert. Doch was, wenn der Entwickler den generierten Quellcode bereits verfeinert und eigene Codefragmente hinzugefügt hat? Um zu verhindern, daß diese Codefragmente beim Generieren überschrieben werden, kann der Entwickler sie wie oben beschrieben in speziellen Abschnitten einfügen. Der Generator erkennt diese Abschnitte und behält den jeweiligen Inhalt bei. Es wird nur Quellcode außerhalb dieser Kommentare vom *PowerTier Builder* ersetzt.

Eine ausführlichere Beschreibung dieser Vorgehensweise findet sich in der mitgelieferten Dokumentation des *PowerTier*-Servers unter [Per01, PowerTier - Programming Guide (S.65)].

Ein einfaches Beispiel für Rahmenquellcode, der vom *PowerTier Builder* generiert wurde, findet sich in Anhang A. Das dort zugrunde liegende Beispielmodell bestand lediglich aus dem Bean `MyClass`. Dieses wiederum enthielt nur das Attribut `id`, welches gleichzeitig auch den PrimaryKey repräsentierte. Wichtig ist in diesem Zusammenhang die Tatsache, daß der gesamte Anhang nur Quellcode enthält, der vom *PowerTier Builder* generiert wurde und der größtenteils nur aus sehr PowerTier-spezifischen Inhalten besteht. Die eigentliche Geschäftslogik fehlt komplett und muß vom Entwickler noch per Hand eingepflegt werden.

Abschließend sind noch zwei Abbildungen aufgeführt, die einen kleinen Eindruck von der Oberfläche dieses Tools schaffen sollen. Abbildung 6.8 zeigt zunächst die Eingabemaske, über die die einzelnen Klassen modelliert werden können (hier z.B. die Klasse `Produkt`).

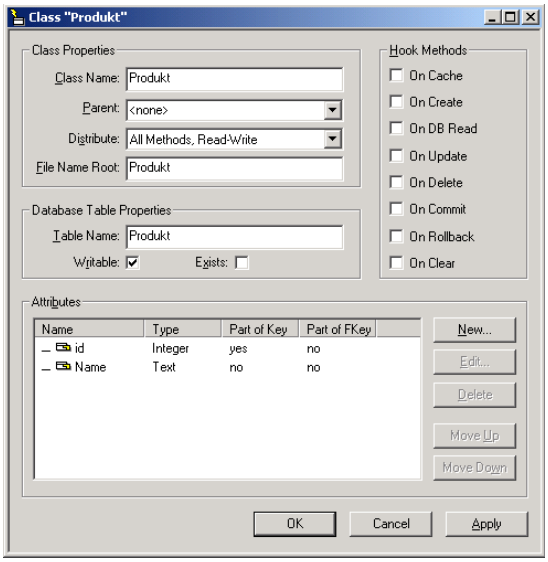

Abbildung 6.8: Die Klasse Produkt im PowerTier Builder

Im Zuge dieser Modellierung kann der Entwickler zum Beispiel Vererbungs-

beziehungen zu anderen Klassen bzw. Beans angeben (`Parent`-Feld). In diesem Zusammenhang fällt auf, daß zu einer „komponentenübergreifenden Vererbung“, wie sie in der Beispielapplikation zwischen den Komponenten `historisierung`, `partner`, `police` und `vertrag` besteht, keine Angaben gemacht werden können. Ob dies zu Problemen bei der Portierung führt, kann an dieser Stelle noch nicht abgeschätzt werden. Weiterhin können auch Informationen zum Mapping der EntityBeans auf die Datenbank angegeben werden (`Database Table Properties`). Die einzelnen Attribute der modellierten Klasse werden über die in Abbildung 6.9 gezeigte Maske eingegeben.

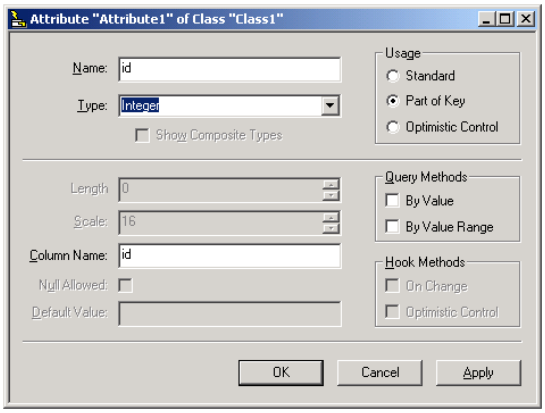

Abbildung 6.9: Eingabe der Attribute im PowerTier Builder

Interessant für die Entwicklung sind an dieser Stelle die vom *PowerTier Builder* angebotenen Datentypen. So stehen unter anderem der Datentyp `OID` (ObjectID) zur Verfügung, über den der PowerTier stets einen eindeutigen PrimaryKey zur Verfügung stellt (vom Typ `int`). Desweiteren werden neben Standarddatentypen, wie zum Beispiel `Integer` oder `Float` auch PowerTier-spezifische Datentypen wie `Blob` oder `Money` angeboten. In diesem Zusammenhang stellt sich die Frage, wie sich die Verwendung solcher Datentypen auf die Portierbarkeit der Anwendung auswirkt. Zu vermuten ist, daß diese Datentypen vom Entwickler entweder nachgebildet oder durch andere Typen ersetzt werden müssen. Die beschriebene `OID` zum Beispiel existiert nur für den PowerTier. Bei der Portierung einer Anwendung, die mittels dieses Datentyps PowerTier-spezifische PrimaryKeys erzeugt, stellt sich somit auf dem neuen Server das Problem, daß auch weiterhin PrimaryKeys benötigt werden. Dazu muß der Entwickler entweder selbst eindeutige Schlüssel erzeugen (die ebenso wie die `OID` vom Typ `int` sein sollten) oder er hat Glück und der neue Server stellt einen vergleichbaren Datentypen zur Verfügung. Ein konkretes Auf der anderen Seite besteht natürlich auch die Möglichkeit, daß im Zuge der Modellierung ein Datentyp benötigt wird, der vom *PowerTier Builder* nicht angeboten wird.

Der Aufwand, den solche Datentypenprobleme mit sich bringen, kann hier

nicht abgeschätzt werden, da er von Fall zu Fall unterschiedlich groß ist. Wichtig ist jedoch die Erkenntnis, daß Portierungsprobleme nicht nur durch die Tools eines Servers, sondern auch durch spezifische Datentypen entstehen können.

Abbildung 6.10 schließlich zeigt, daß es sogar möglich ist, die Relationen zwischen einzelnen Beans zu modellieren. Eine Einschränkung besteht allerdings darin, daß diese Relationen nur innerhalb eines Modells, also innerhalb einer Komponente modelliert werden können.

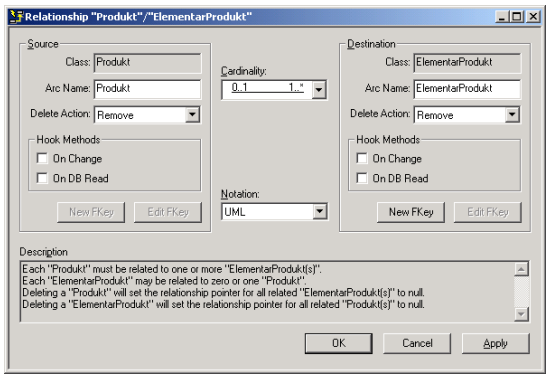

Abbildung 6.10: Eingabe der Attribute im PowerTier Builder

6.3.5 PowerTier Command Center

Das *PowerTier Command Center* stellt dem Benutzer eine grafische Oberfläche zur Verfügung, über die einer oder mehrere PowerTier Server gestartet, zur Laufzeit verwaltet und zu sogenannten Clustern zusammengeschlossen werden können. Für die Portierung ist dieses Tool nicht von Bedeutung, da es lauffähige Anwendungen voraussetzt und somit erst nach der Portierung verwendet werden kann.

6.3.6 Befehlszeilenkommandos

Alle hier aufgeführten Tools, die sich über eine grafische Oberfläche bedienen lassen, können auch über eine Eingabeaufforderung bedient werden (für Details siehe [Per01]). Zusätzlich bietet der *PowerTier* jedoch auch einige Dienste an, die sich nur über eine Eingabeaufforderung verwenden lassen.

Mittels des ps-makeejb-Befehls können verschiedene Aufgaben erledigt werden. Dies sind unter anderem:

- Erzeugen der `.jar`-Files, welche im Server deployed werden
- Löschen von Dateien der Pantry

- Erzeugen und aktualisieren der DeploymentDescriptoren
- Erzeugen, editieren, hinzufügen oder aktualisieren des Security-DeploymentDescriptors

Im Zuge der Portierung wird dieser Befehl mit den Parametern `-createDD` (Erzeugen des DeploymentDescriptors), `-all` (Erzeugen des Deployment-Descriptors und der JAR-Archive) und `-cleanAll` (Säubern der Pantry) verwendet. Da der Befehl auf die Portierung jedoch keinerlei Auswirkungen hat, wird für die weiteren Möglichkeiten auf die Dokumentation [Per01, Class Reference, S.63 ff.] verwiesen.

Mit dem Befehl ps-read-dd-* wird dem Nutzer des PowerTier die Möglichkeit gegeben, aus einer existierenden Oracle-Datenbank (`ps-read-dd-oracle`) oder einer Sybase-Datenbank (`ps-read-dd-sybase`) heraus eine PowerTier-Projektdatei (`.per`-Datei, s. Kapitel 6.3.7) zu erstellen. Dazu werden die tables und views der jeweiligen Datenbank eingelesen und in einem für den PowerTier Builder (Kapitel 6.3.4) verständlichen Format abgelegt. Der Entwickler kann nun diese Datei von Hand oder über den PowerTier Builder editieren und anschließend den Quellcode generieren lassen. Obwohl diese Möglichkeit sicherlich sehr interessant ist, so ist sie doch für die Portierung von keinerlei Bedeutung. Die verwendete Oracle-Datenbank bleibt im Zuge der Portierung bestehen und soll lediglich als Datenspeicher dienen. Somit wird für eine detailliertere Beschreibung dieses Befehls auf [Per01, PowerTier-Class Reference, S.70 ff. und S.76 ff.] verwiesen.

Der Befehl ps-run-server kann verwendet werden, um den Server per Kommandozeile zu starten. In diesem Zusammenhang ist besonders die `.ptc`-Konfigurationsdatei von Bedeutung (Kapitel 6.3.7), da sie alle Parameter enthält, die für den Einsatz des Servers notwendig sind.

6.3.7 Konfigurationsdateien

Wichtige Informationen zu den EJB-Anwendungen werden vom PowerTier in verschiedenen Konfigurationsdateien hinterlegt. Die wichtigsten sollen an dieser Stelle kurz vorgestellt werden.

Editierbare DeploymentDescriptoren legt der PowerTier in den `.des`-Dateien ab. Diese DeploymentDescriptoren liegen als Textdateien vor und werden später für den Einsatz im Server serialisiert (s. *Serialisierte DeploymentDescriptoren*). Für jedes Bean existiert genau eine `.des`-Datei, die entweder vom Codegenerator des PowerTier Builder (Kapitel 6.3.4), vom Benutzer selbst mittels des `ps-makeejb`-Befehls (Kapitel 6.3.6) oder

in einem Texteditor „von Hand“ erstellt wird. Da in diesen Dateien Informationen zu den einzelnen Beans vorliegen und der Anwender durch das Setzen bestimmter Werte Einfluß auf den Einsatz im Server ausüben kann, sind diese Dateien für die Portierung von Interesse. Inwiefern es hier jedoch zu Problemen kommen kann bzw. wird, läßt sich an dieser Stelle noch nicht sagen. Dies liegt daran, daß die `.des`-Dateien in diesem Fall erst für die einzelnen Beans erzeugt werden müssen, da die serialisierten DeploymentDescriptoren der Beispielanwendung nicht für den Einsatz im PowerTier geeignet sind. Nach dem Erzeugen können die Werte dann an den Einsatz im Server angepaßt werden und das JAR-Archiv der Komponente kann erzeugt werden. Von Vorteil ist dabei die Tatsache, daß die DeploymentDescriptoren der Beispielanwendung nicht nur als serialisierte Dateien, sondern auch als einfache Textdateien vorliegen. Somit können unter Umständen einige Werte, wie zum Beispiel die Transaktionsattribute (`TX_REQUIRED`, `TX_SUPPORTS`, etc.), übernommen werden.

In der Dokumentation findet sich unter [Per01, PowerTier-Class Reference, S.88/89] folgende Vorgabe einer `.des`-Datei, die beim Erstellen mit den zugehörigen Werten gefüllt wird und anschließend für den Einsatz im Server angepaßt werden muß.

```
BeanType // One of: entity or session
{
    VersionNumber // 1
    {
        1.0 // default
    }
    BeanHomeName // 1
    {
       ClassHome // PowerTier default
       // any fully-qualified package name is valid
       // an empty field disables JNDI publication
    }
    ControlDescriptor// 0:m
    {
       IsolationLevel// 1
       {
          TRANSACTION_READ_COMMITTED// default
       }
       Method // 1
       {
          // Method signature using fully qualified type names
          // for example, atm.AccountState deposit(float,java.lang.String)
          // throws java.rmi.RemoteException, javax.ejb.FinderException
```

```
   }
   RunAsMode // 1
   {
      SYSTEM_IDENTITY// default
   }
   TransactionAttribute// 1
   {
      TX_REQUIRED// default for customer-defined remote methods
      // Entity getters will be TX_SUPPORTS
      // Entity setters will be TX_REQUIRED
   }
}
EnterpriseBeanClassName// 1
{
   Package. ClassBean
}
EnvironmentProperty// 0:m (customer specified)
{
   PropertyKey
   PropertyValue
}
HomeInterfaceClassName// 1
{
   Package. ClassHome
}
Reentrant // 1
{
   false// false for session bean, but true for entity bean
}
RemoteInterfaceClassName// 1
{
   Package. Class
}
ContainerManagedField// 0:1, Entity Beans only
{
   FieldName
   FieldName
   FieldName // list of field names
}
PrimaryKeyClassName// 1, Entity Beans only
{
   Package. ClassKey// default for PowerTier Entity Beans only
}
```

```
    SessionTimeout // 1, Session Beans only
    {
       60 // default, set to zero for an infinite lifetime
    }
    StateManagementType// 1, Session Beans only
    {
       STATELESS_SESSION// default
    }
}
```

Serialisierte DeploymentDescriptoren finden sich in Dateien mit der Endung `.ser` wieder. Sie können im Gegensatz zu den *editierbaren DeploymentDescriptoren* (s.o.) nicht mehr von Hand editiert werden und werden ausschließlich vom Server zur Laufzeit zum Einlesen der Informationen über die einzelnen Beans verwendet. Eventuelle Änderungen müssen somit also in den `.des`-Dateien vorgenommen werden und anschließend in die `.ser`-Dateien nachgezogen werden. Diese Aktualisierung wird beim Erzeugen der JAR-Archive vorgenommen, in welchen neben den eigentlichen Beans auch die *serialisierten DeploymentDescriptoren* aufgenommen werden. Für eine Portierung sind diese Dateien nicht von Bedeutung, da sie lediglich die Informationen der *editierbaren DeploymentDescriptoren* in einem serialisierten Format wiedergeben und die Änderungen und Anpassungen in genau diesen vorgenommen werden.

Die Modelle des PowerTier Builders werden in den `.per`-Dateien hinterlegt. Dabei existiert für jedes Package bzw. jede Komponente genau eine `.per`-Datei, die die Informationen über das entsprechende mit dem PowerTier Builder modellierte Projekt enthält. Da diese Dateien im Textformat vorliegen, können auch sie ohne Probleme per Texteditor bearbeitet werden. Dies ist jedoch nicht empfehlenswert, da der Überblick, der sich bei der Arbeit mit einer grafischen Oberfläche bietet, sehr leicht verloren geht. Wichtig sind diese Projektdateien für den Prozeß der Quellcodegenerierung, da aus ihnen das Grundgerüst der jeweiligen Home- und RemoteInterfaces, der BeanClasses und der PrimaryKeyClasses erzeugt wird. Dies nimmt zwar dem Entwickler eine Menge Arbeit ab, beinhaltet jedoch die bereits beschriebenen Probleme aus Kapitel 6.3.4. Da sie lediglich das Ergebnis der Arbeit mit dem PowerTier Builder wiederspiegeln, haben diese Dateien keinerlei Einfluß auf die geplante Portierung, so daß an dieser Stelle nicht weiter auf sie eingegangen wird. Für weitere Informationen wird auf die entsprechende Literatur aus [Per01, PowerTier-Class Reference, S.93ff] verwiesen.

Eine Konfigurationsdatei zum Starten des Servers stellt die `.ptc`-Datei dar. Sie liegt im XML-Format vor und ist für die Arbeit mit dem PowerTier von sehr großer Bedeutung, da in ihr viele Einstellungen vorgenommen werden, die für einen späteren Einsatz des Servers notwendig sind. Für die Portierung ist sie zunächsteinmal nicht von Bedeutung. Es ist jedoch wichtig zu wissen, welche Einstellungen in der `.ptc`-Datei vorgenommen wurden. Ansonsten kann es leicht passieren, daß zum Beispiel die Tabellen der Datenbank beim Start des Servers gelöscht werden, weil der Eintrag `DropTablesBeforeCreate` auf `true` gesetzt wurde. Aufgrund der zahlreichen Einstellungsmöglichkeiten in der `.ptc`-Datei wird an dieser Stelle auf die Angabe einer kompletten Vorlage wie im Kontext der `.des`-Dateien verzichtet. Sollte es sich allerdings als nötig erweisen, wird im weiteren Verlauf der Arbeit auf wichtige Einstellungen detaillierter eingegeangen. Ansonsten wird auch hier auf [Per01, PowerTier-Class Reference, S.110ff] verwiesen.

Informationen für den ps-makeejb-Befehl werden der Datei `ps-makeejb.cfg` entnommen. Diese Datei wird entweder vom PowerTier Builder beim Generieren des Rahmenquellcodes oder vom Anwender selbst in einem Texteditor erstellt. Wie bereits in Kapitel 6.3.6 beschrieben, können mittels des `ps-makeejb`-Befehls JAR-Archive und DeploymentDescriptoren (`.des`-Files) erzeugt werden. Die Informationen, zu welchen Beans bzw. Packages die entsprechenden Teile erstellt werden sollen, sind wie folgt in der Datei `ps-makeejb.cfg` codiert (vgl. [Per01, PowerTier-Class Reference, S.108]):

```
PROJECT_NAME=projectName
PACKAGE_NAME=packageName
[PACKAGE_NAME= ...]
```

Enthalten sind hier in der ersten Zeile der Name für das gesamte Projekt und anschließend mindestens der Name eines zu verarbeitenden Package. Zu diesen Packages werden wahlweise nur die DeploymentDescriptoren erzeugt (Parameter: `-createDD`), oder aber zusätzlich die entsprechenden JAR-Archive (Parameter : `-all`), welche die entsprechenden Packages beinhalten und zu einem späteren Zeitpunkt im Server deployed werden soll. Für die geplante Portierung wird daher folgendes Vorgehen als sinnvoll erachtet:

Bei der Portierung soll Komponente für Komponente portiert werden. Der Projektname wechselt die ganze Zeit über nicht. Es wird jedoch stets nur ein Package in der `ps-makeejb.cfg` aufgeführt. Für dieses Package wird zunächst der DeploymentDescriptor erzeugt und anschließend ein JAR-Archiv. Insgesamt wäre auch ein einzelnes JAR-Archiv möglich, welches

alle Komponenten enthält. Dies widerspricht jedoch der Idee der komponentenbasierten Softwareentwicklung. So würde zum Beispiel dieses JAR-Archiv die gesamte Anwendung enthalten, so daß ein Austausch einzelner Komponenten nicht möglich wäre, sondern stets nur die gesamte Anwendung ausgetauscht werden könnte.

Insgesamt übt dieser Befehl jedoch keinen Einfluß auf die Portierungen aus.

6.3.8 Server- und Container APIs

Die bisher vorgestellten Tools des PowerTier beeinflussen den Quellcode einer EJB-Applikation in unterschiedlicher Intensität. Einige Tools, wie zum Beispiel der *PowerTier Profile Wizard*, haben keinerlei Auswirkung auf den Quellcode. Andere Tools wiederum, wie zum Beispiel der PowerTier Builder, sind speziell für die Generierung von Quellcode entwickelt worden. Um jedoch die Funktionalitäten des Applikationsservers und des Containers ansprechen und nutzen zu können, müssen, ebenso wie bei der normalen Arbeit mit Java, spezielle APIs importiert werden. Von diesen stellt Persistence dem Entwickler eine nicht gerade geringe Anzahl zur Verfügung. Da eine detaillierte Auflistung aller Möglichkeiten im Rahmen der Arbeit jedoch nicht möglich ist, sollen an dieser Stelle lediglich die Bereiche genannt werden, auf die der Entwickler durch die Verwendung entsprechender APIs Einfluß nehmen kann (vgl. [Per01, PowerTier-Class Reference, S.289]):

- Administrative
- Cache
- Cache Synchronization
- Constants
- Database
- Exceptions
- Failover
- JNDI
- Load Balancing
- PowerTier Server
- Transactions
- Utility

Einen Einfluß auf eine Portierung üben diese APIs jedoch nur dann aus, wenn es gilt eine Applikation zu portieren, die diese APIs bereits nutzt. Dies ist in der Diplomarbeit jedoch nicht gegeben. Ansonsten müssten auch für diesen Fall Lösungen gefunden werden, da nach der Portierung keine „Altlasten“ (Kapitel 4.4) des ursprünglich verwendeten Servers weiterverwendet werden sollen. Somit ergibt sich folgende Schlußfolgerung: Je mehr APIs verwendet werden, um so größer ist der Aufwand einer späteren Portierung.

6.3.9 Zusammenfassung

An dieser Stelle folgt nun eine Zusammenfassung dessen, was die Beschreibung des PowerTier-Applikationsservers mit Hinblick auf eventuelle Portierungsprobleme ergeben hat. Insgesamt gesehen muß das Problem der Portierung zunächst einmal in zwei Bereiche aufgegliedert werden. Dies sind zum einen Portierungen auf den PowerTier und zum anderen Portierungen vom PowerTier auf andere Server.

Portierungen auf den PowerTier sind zunächst einmal unabhängig von dessen Tools, da diese lediglich im Kontext der Entwicklung von EJB-basierten Anwendungen verwendet werden. Es kann jedoch sein, daß das eine oder andere Tool im Zuge der Portierung auf den PowerTier als Hilfsmittel verwendet werden kann. Von weitaus grösserem Interesse sind diese Tools im Anschluß an die Portierung wenn es darum geht, eine Adaption (Kapitel 3.1.2) an den neuen Server vorzunehmen. Diese stellt allerdings kein Ziel der Arbeit dar und wird somit auch nicht weiter untersucht.

Portierungen vom PowerTier werden nur von der theoretischen Seite her betrachtet, da eine Portierung vom PowerTier auf den WebLogic (oder einen beliebigen anderen Applikationsserver) nicht vorgesehen ist. Um dies zu realisieren, müßte die Beispielapplikation zunächst vom WebLogic auf den PowerTier portiert werden und anschließend wieder auf einen oder mehrere andere Server zurück. Aus Zeitgründen wird auf diesen Schritt jedoch verzichtet. Es ist allerdings erstaunlich, welche Portierungsprobleme sich bereits durch eine rein theoretische Betrachtung des PowerTier-Servers ergeben. Die meisten dieser Probleme beruhen auf der Verwendung von PowerTier-spezifischen Leistungen. Besonders durch die Verwendung der speziellen Container- und Server-APIs wird die Möglichkeit einer späteren Portierung vermindert. Der Anwender ist in diesem Fall gezwungen, diese speziellen Teile der Applikation durch PowerTier-unabhängige Teile zu ersetzen. Ob dies jedoch generell möglich ist, und falls ja, unter welchem Aufwand, hängt sicherlich mit der Anzahl der verwendeten PowerTier-Spezifika zusammen. Dabei kann die Verwendung dieser Spezifika sogar dazu führen, daß eine Portierung der Applikation grundsätzlich ausgeschlossen werden kann. So besteht zum Beispiel laut Persistence keine Möglich-

keit, Applikationen, die PowerTier-spezifisches Container-Managed Persistence verwenden, auf einem anderen Server einzusetzen. Eine ähnliche, wenn auch allgemeine Aussage, findet sich bei Klein:

> *„...scheitert der Versuch, ein PowerTier-Programm auf einen anderen Application Server zu transferieren.“*
>
> ([Kle00, S.58])

Insgesamt scheinen sich die Probleme, die bei der Portierung vermutlich auftreten werden, im Rahmen zu halten. Die Thesen hinsichtlich der Portierungsprobleme beschränken sich somit auf folgende Bereiche:

Der Wechsel der EJB-Spezifikation von Version 1.0 (WebLogic) hinzu Version 1.1 (PowerTier) bringt unter anderem einige Veränderungen am Quellcode mit sich, die sich vor allem im Bereich des Container-Managed Persistence auswirken (vgl. [MH99b, S.307]). Betroffen sind in diesem Fall die Beans `Bedingung` (Komponente `bedingung`), `Objekt` (Komponente `vertrag`) und `MerkmalWert` (inklusive aller erbenden Beans, Komponente `merkmal`). Aber auch Probleme mit dem Client werden wahrscheinlich nicht zu vermeiden sein, da sich auch im Bereich des JNDI einige Änderungen ergeben haben und somit das Auffinden der Interfaces bzw. die Verbindung zwischen Client und Server anders als in der Beispielapplikation umzusetzen ist.

Die implementierten Vererbungen können zumindest dann übernommen werden, wenn es sich um „komponenteninterne Vererbungen“ (Komponente `merkmal`) handelt. Wie es um „komponentenübergreifende Vererbungen“ (Komponenten `historisierung`, `partner`, `police` und `vertrag`) steht, läßt sich noch nicht absehen, zumal die Spezifikation hinsichtlich einer solchen Vererbung keine Angaben macht.

Da die im PowerTier enthaltenen Tools und APIs nicht verwendet werden, sind von dieser Seite her zunächst einmal keine Probleme zu erwarten. Eventuelle weitere Probleme und die Bestätigung oder Wiederlegung der in diesem Kapitel aufgestellten Thesen werden im Zuge der Portierung in Kapitel 7 behandelt.

6.4 WebSphere

Im Gegensatz zu dem in Kapitel 6.3 beschriebenen PowerTier-Server stellt der *WebSphere*-Applikationsserver der Firma IBM nicht nur eine Infrastruktur für EJB-basierte Anwendungen zur Verfügung, sondern für J2EE- und Web-Anwendungen im Allgemeinen. Für die Portierung der Beispielapplikation wird die *Advanced Edition 3.5.0* verwendet, die als 30-Tage-Testversion vorliegt. Auf

die neueste Version dieses Servers (Version 4.0), die erst vor Kurzem erschienen ist, soll in Kapitel 10 ein wenig näher eingegangen werden.

Zu Beginn der theoretischen Betrachtung des *WebSphere* werden wie bereits beim PowerTier zunächst allgemeine Informationen zum Server erörtert, bevor anschließend die speziellen Eigenschaften näher untersucht werden.

6.4.1 Allgemeines

Der *WebSphere*-Applikationsserver von IBM setzt auf der EJB-Spezifikation 1.0 auf. Bereits diese Tatsache läßt vermuten, daß die Portierung der Beispielapplikation weniger Probleme aufwirft, als es bei der Portierung auf den PowerTier der Fall sein wird. Der Grund für diese Vermutung liegt darin, daß der BEA WebLogic 4.5.1, auf dem die Beispielapplikation entwickelt wurde, ebenfalls auf der EJB-Spezifikation 1.0 aufsetzt. Es handelt sich in diesem Fall also um eine Portierung, bei der keine Probleme aufgrund unterschiedlicher Spezifikationsversionen zu erwarten sind.

Aber nicht nur über die Spezifikation kann eine Ähnlichkeit zum WebLogic erkannt werden. Auch das äußere Erscheinungsbild des *WebSphere* und sein Leistungsumfang für die Arbeit mit EJBs ähneln denen des WebLogic. Im Folgenden werden diese Eigenschaften des *WebSphere* nun detailliert dargestellt, um auch für die zweite Portierung Thesen bezüglich eventueller Portierungsprobleme aufstellen zu können.

6.4.2 Die Administrationskonsole

Die gesamte Verwaltung der im WebSphere-Server installierten Anwendungen erfolgt ebenso wie beim WebLogic über eine grafische Oberfläche, die sogenannte WebSphere-Administrationskonsole. Abbildung 6.11 zeigt die Konsole, die sich in drei große Bereiche aufteilt. Von Interesse sind dabei eigentlich nur die beiden oberen Fenster. Das linke dieser Fenster enthält eine Baumstruktur, hinter der sich die zu verwaltenden Bereiche des Servers verbergen (z.B. Datenbankanbindung, EJB-Containerverwaltung). Die Einstellungen innerhalb dieser Bereiche wiederum können im rechten Fenster vorgenommen werden. Im unteren Fenster schließlich werden Informationen zu den Vorgängen im Server ausgegeben.

Für die Portierung selbst ist die Administrationskonsole nicht von Bedeutung, da mit ihr lediglich existierende Anwendungen installiert und gestartet werden. Allerdings stellt der WebSphere zwei interessante Features zur Verfügung, die an dieser Stelle noch kurz erwähnt sein sollten:

- Es besteht die Möglichkeit des sogenannten *Hot-Deployment*, das heißt es können zur Laufzeit des Servers Komponenten hinzugefügt, entfernt und ausgetauscht werden

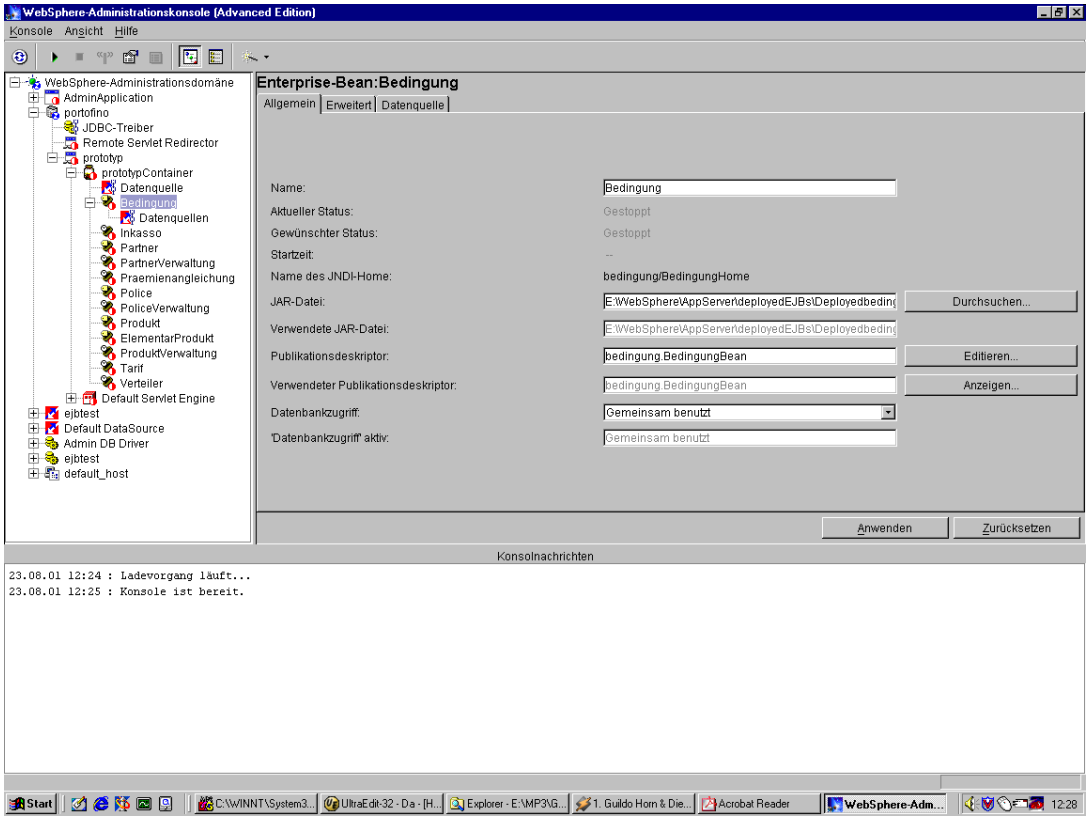

Abbildung 6.11: Die WebSphere-Administrationskonsole

- Der Anwender hat jederzeit die Möglichkeit, Einfluß auf die Werte der einzelnen DeploymentDescriptoren zu nehmen. Dies geschieht über die nachfolgend abgebildeten Masken

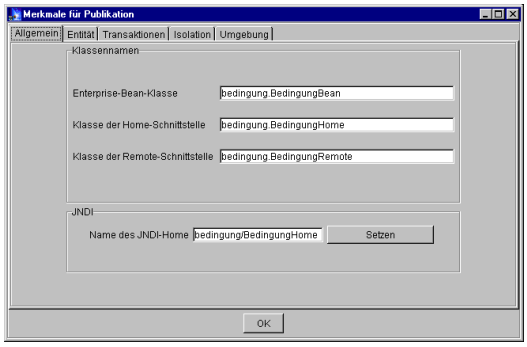

Abbildung 6.12: Allgemeine Informationen zur Bean

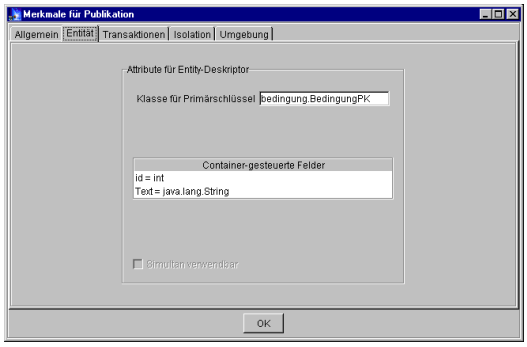

Abbildung 6.13: Informationen zur Entität

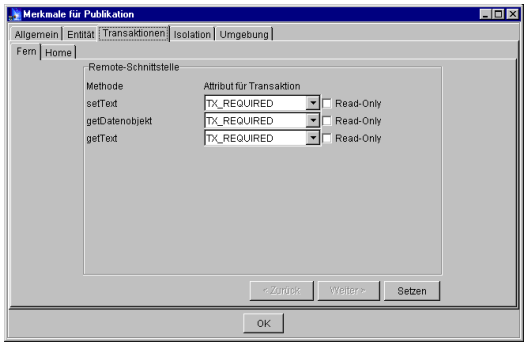

Abbildung 6.14: Transaktionsverhalten der Bean-Methoden

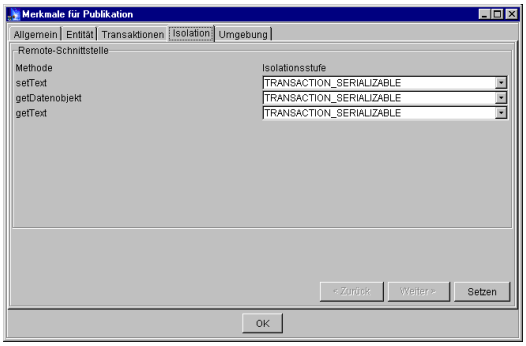

Abbildung 6.15: Isolationsstufe der Remote-Methoden

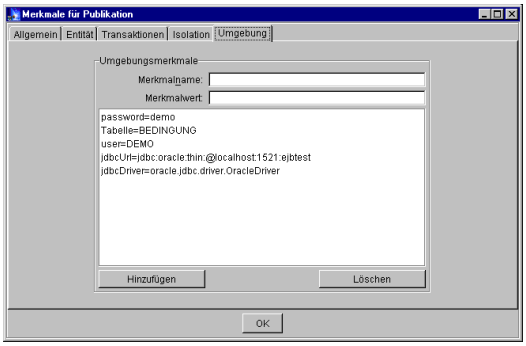

Abbildung 6.16: Umgebungsvariablen der Bean

6.4.3 Arbeiten mit EJBs

Für den Umgang mit EJBs gibt es die zwei folgenden Möglichkeiten (vgl. [Web01b]):

1. Die EJBs werden mittels eines beliebigen Texteditors erstellt und mit Hilfe diverser Programme des JDK (Java Development Kit) bearbeitet. Diese Bearbeitung sieht zunächst das Kompilieren der Quelldateien durch das Programm `javac` vor. Anschließend werden die kompillierten Beans mit Hilfe des Programms `jar` in JAR-Archiven zusammengefaßt. Dabei ist es dem Anwender überlassen, ob er für die gesamte Anwendung ein JAR-Archiv erstellt, welches alle Komponenten enthält, oder ob für jede Komponente ein eigenes JAR-Archiv erzeugt wird. Die zuletzt genannte Variante erscheint jedoch hinsichtlich des Komponentengedankens sinnvoller.

   Bevor die JAR-Archive nun im Server installiert werden können, müssen noch die DeploymentDescriptoren der einzelnen Beans erzeugt und hinzugefügt werden. Für diesen Vorgang stellt der WebSphere das Tool *Jetace* zur Verfügung, welches in Kapitel 6.4.4 noch detailliert erläutert wird.

Sind die DeploymentDescriptoren schließlich erstellt, können sie in das JAR-Archiv der jeweiligen Komponente integriert werden.

Grundsätzlich erscheint diese Möglichkeit des Umgangs mit EJBs hinsichtlich einer Portierung als sehr vorteilhaft, da über das JDK nur auf reine Java-Lösungen zurückgegriffen wird, die in keinerlei Abhängigkeit zu einem Applikationsserver und auch nicht in Abhängigkeit zur EJB-Technologie stehen. IBM schränkt die Verwendung dieser reinen Java-Lösung jedoch an einigen Stellen der Dokumentation stark ein ([Web01b]):

(a) Für die Zusammenarbeit mit dem WebSphere-Applikationsserver wird das von IBM mitgelieferte JDK vorausgesetzt, welches eine Modifikation des von Sun zur freien Verfügung gestellten JDK darstellt. Inwiefern dies zu Problemen bei der Portierung führt muß die Portierung in Kapitel 8 zeigen.

(b) Werden die DeploymentDescriptoren im XML-Format angelegt, um aus ihnen serialisierte Descriptoren für den Einsatz im Server zu erstellen, wird von der Verwendung des Jetace-Tools abgeraten.

(c) Teilweise wird sogar von dem oben beschriebenen Vorgehen insgesamt abgeraten. Stattdessen empfiehlt IBM die Verwendung der Entwicklungsumgebung *Visual Age für Java Enterprise Edition*. Dabei handelt es sich allerdings nur solange um eine Empfehlung, wie keine Container-Managed EntityBeans verwendet werden. Eine Begründung dieser Einschränkung behandelt Kapitel 6.4.5.

2. Die gesamte Verwaltung der EJBs wird über die bereits erwähnte Entwicklungsumgebung *Visual Age für Java Enterprise Edition* realisiert. Der Begriff „Verwaltung“ umfaßt in diesem Zusammenhang nicht nur die gesamte Entwicklung und Wartung der einzelnen Beans, sondern auch die mögliche Integration einer bestehenden Datenbank oder das *Hot-Deployment* fertig entwickelter Komponenten in einen laufenden WebSphere-Applikationsserver. Die Problematik dieser Möglichkeit besteht in der entstehenden Abhängigkeit zur vorgegebenen Entwicklungsumgebung, die unter Umständen nicht so einfach wieder aufgelöst werden kann und die zusätzlich zum Applikationsserver weitere Kosten verursacht. Angesichts der bereits erwähnten Probleme mit Container-Managed EntityBeans läßt sich eine solche Abhängigkeit unter Umständen allerdings nicht vermeiden.

Insgesamt empfiehlt sich für die Diplomarbeit die erste Variante, da sie mit wenig Aufwand sehr schnell und effektiv zu realisieren ist und gleichzeitig die Abhängigkeit zu weiteren Tools vermeidet. Angesichts der geringen Anzahl von Container-Managed EntityBeans scheint es sinnvoller zu sein, bei der Portierung dieser Beans entstehende Probleme auf eine andere Art und Weise als über Tools

wie *Visual Age* zu lösen. Ob und wie sich dies im weiteren Verlauf umsetzen läßt, muß in Kapitel 8 geklärt werden.

6.4.4 DeploymentDescriptoren mit Jetace

DeploymentDescriptoren werden vom WebSphere-Applikationsserver ebenso wie beim PowerTier als serialisierte Dateien (.ser) erwartet. Um diese Dateien zu erhalten, gibt es zwei grundlegende Möglichkeiten:

1. Es werden XML-Dateien angelegt, die dem geforderten Aufbau des WebSphere entsprechen und die die notwendigen DeploymentDescriptor-Informationen enthalten. Diese XML-Dateien können zum Beispiel über das Jetace-Tool eingelesen, bearbeitet und als serialisierte DeploymentDescriptoren im JAR-Archiv der Komponente gespeichert werden. Der WebSphere stellt dazu XML-Beispieldateien zur Verfügung, die vom Entwickler angepaßt werden können.

2. Der Entwickler verwendet ein Tool, über welches die Informationen der DeploymentDescriptoren vom Entwickler eingegeben und in einer für den WebSphere gültigen Art und Weise serialisiert werden. IBM liefert hierzu das Tool *Jetace* mit dem WebSphere aus, welches nachfolgend kurz vorgestellt wird.

Die zuerst genannte Möglichkeit kann laut Dokumentation des WebSphere ausgeschlossen werden, da es bei der Verwendung von XML-Vorlagen für das Tool *Jetace* zu Problemen kommt. Somit werden die DeploymentDescriptoren im Zuge der Portierung ohne XML-Vorlagen und nur mit dem *Jetace*-Tool generiert.

Das erstellen der DeploymentDescriptoren selbst setzt ein JAR-Archiv voraus, welches die `CLASS`-Dateien mindestens einer Bean enthält. Dieses Archiv wird zunächst mittels des *Jetace*-Tools eingelesen. Anschließend können die DeploymentDescriptoren über eine grafische Oberfläche erstellt und mit den Beans zusammen in einem JAR-Archiv abgelegt werden. Dieses Archiv wiederum kann dann im Server als fertige Komponente installiert werden. Die nachfolgenden Abbildungen zeigen die Oberfläche des *Jetace*-Tools nach dem Laden des `PartnerBean` (Abb. 6.17) und die Eingabemaske für den entsprechenden DeploymentDescriptor (Abb. 6.18) - in diesem Fall den Reiter mit den Eingaben für die Umgebungsvariablen.

6.4.5 Probleme mit Container-Managed EntityBeans

Wie bereits erwähnt, empfiehlt IBM in der Dokumentation des WebSphere bei der Verwendung von Container-Managed EntityBeans den Gebrauch der Entwicklungsumgebung *Visual Age für Java Enterprise Edition*. Diese Empfehlung ist im Zusammenspiel zwischen Server und Datenbank begründet. Um

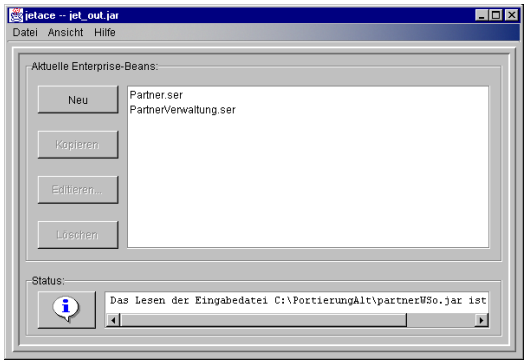

Abbildung 6.17: Das Jetace-Tool

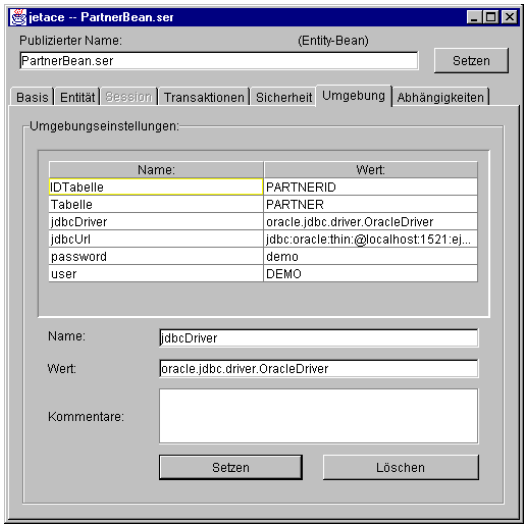

Abbildung 6.18: Eingabe eines DeploymentDescriptors mit dem Jetace-Tool

die gewünschten Attribute einer Bean persistent in der Datenbank ablegen zu können, muß dem Server mitgeteilt werden, welches Attribut auf welche Spalte einer bestimmten Tabelle abgebildet wird. Im Falle der bereits behandelten Server geschah dies über den DeploymentDescriptor (WebLogic) bzw. mit Hilfe eines Tools (PowerTier Builder), welches vom Server zur Verfügung gestellt wurde. Der WebSphere-Applikationsserver bietet für die Lösung dieses Problems nur die WebSphere-Administrationskonsole an. Über diese kann bei der Installation eines Container-Managed EntityBeans im Server angegeben werden, ob die jeweilige Tabelle angelegt werden soll. Abbildung 6.19 zeigt als Beispiel den entsprechenden Ausschnitt der Administrationskonsole für das Bean `BedingungBean`. Es fällt auf, daß die Möglichkeiten, die die Konsole für das Mapping auf die Datenbank bietet, sehr begrenzt sind. Einzig über die Option „Tabelle erstellen“ kann hier Einfluß ausgeübt werden. Ist diese Option aktiviert, so legt der WebSphere beim Zugriff auf die Datenbank eine Tabelle für die entsprechende Bean an, sofern diese noch nicht existiert. Der Name der Tabelle ist dabei fix und wird mit dem Namen `EJB.Bean-NameBeanTbl` belegt, wobei `Bean-Name` durch den Namen der jeweiligen Bean zu ersetzen ist (z.B. Partner). Auch bei den Namen der Datenbankspalten und deren Reihenfolge sind laut Dokumentation einige Dinge zu beachten:

> „*Hinweise zum Publizieren von CMP-Entity-Beans*
>
> *[...]Wenn Sie CMP-Entity-Beans für eine Legacy-Anwendung (oder Beans von anderen Herstellern) verwenden, müssen Sie VisualAge für Java zur Erstellung der publizierten JAR-Datei verwenden. Anschlieend können Sie das Bean erstellen (installieren), indem Sie die WebSphere Administrationskonsole verwenden.*

Abbildung 6.19: Datenbankmapping eines CMP-Beans

> *Es wird dringend empfohlen, zum Publizieren von Beans, die in Legacy-Anwendungen eingesetzt werden und komplexe Zuordnungen zu einer Datenbanktabelle erforderlich machen, VisualAge für Java zu verwenden. Wenn Sie den automatischen Publikationsprozess in der Konsole ausführen, ist nicht sichergestellt, dass die Reihenfolge und die Namen der Spalten in der generierten Tabelle mit der Tabellenkonfiguration übereinstimmen, die für die Legacy-Anwendung erforderlich ist. (Beim Publizieren über die Konsole wird eine bestimmte Reihenfolge der Container-gesteuerten Felder vorausgesetzt.)[...]*
>
> *[...]Wenn Sie sich für das automatische Publizieren über die Konsole entscheiden, jedoch die Datenbanktabelle manuell erstellen möchten, müssen Sie folgendes beachten:*
>
> *Der Name der Datenbanktabelle muss der Konvention EJB.Bean-NameBeanTbl entsprechen [...] Die Felder für den Primärschlüssel müssen zuerst erscheinen. Die Spaltenüberschriften in der Datenbank müssen mit den Namen und der Reihenfolge der Felder im Publikationsdeskriptor übereinstimmen.*“
>
> ([Web01b])

Dieser Hinweis zeigt, daß die bestehende Datenbankstruktur scheinbar nicht übernommen werden kann, ohne daß auf die Entwicklungsumgebung *Visual Age für Java* zurückgegriffen wird oder die Datenbankstruktur an die Vorgaben des WebSphere angepaßt wird. Der Grund für dieses Problem besteht in der Abbil-

dung der zu speichernden Bean-Attribute auf die Datenbank. Für diese Abbildung gibt es zwei Möglichkeiten:

1. Das Datenbankmapping wird dem Server über die Administrationskonsole überlassen. Dabei werden intern fixe Namen und Reihenfolgen für die Tabellen und Spalten vorgegeben, auf die der Anwender keinen Einfluß hat.

2. Das Datenbankmapping wird über die Entwicklungsumgebung *Visual Age für Java* realisiert. Mit Hilfe dieses Tools muß zunächst das Datenbankschema einer existierenden Datenbank eingelesen werden. Anschließend kann das Datenbankmapping vorgegeben und über den DeploymentDescriptor mit der entsprechenden Bean zusammen in einem JAR-Archiv abgelegt werden. Der Anwender besitzt somit auf Kosten der Abhängigkeit zu einem weiteren Tool vor allem im Umgang mit bereits existierenden Datenbanken mehr Möglichkeiten.

Mit Hinblick auf die geplante Portierung scheint es also zu Problemen mit den Container-Managed EntityBeans zu kommen.

Ein weiteres Problem, welches im Zuge der Portierung auftreten wird, betrifft die `Finder`-Methoden, die über das HomeInterface zur Verfügung gestellt werden. Die EJB-Spezifikation schreibt in diesem Zusammenhang lediglich eine `findByPrimaryKey`-Methode vor, die zu einem gegebenen PrimaryKey die entsprechende Bean-Instanz aus der Datenbank zurückliefert. Bei der Verwendung von Container-Managed EntityBeans ist dem Server aufgrund der Vorgaben der EJB-Spezifikation bekannt, wie diese Methode intern zu implementieren ist und welche Werte benötigt werden. Es sind jedoch auch weitere vom Entwickler benötigte `Finder`-Methoden denkbar, deren Verhalten der Server nicht automatisch intern implementieren kann. So ist zum Beispiel eine Methode `findAllPartners(Kriterien)` denkbar, die eine Menge von Partnern zurückliefert, die bestimmte Kriterien erfüllen. Diese Suche kann unter Umständen die Werte mehrerer Datenbanktabellen einbeziehen. Für Container-Managed EntityBeans mit speziellen `Finder`-Methoden wird daher jeweils eine spezielle Schnittstelle gefordert:

> „*Erstellen der Finder-Logik im EJB-Server (AE)*
>
> *In der EJB-Server (AE)-Umgebung ist die folgende Finder-Logik für jede der im Home-Interface eines Entity-Beans mit CMP enthaltenen Finder-Methoden erforderlich (mit Ausnahme der Methode findByPrimaryKey):*
>
> *Die Logik muss in einer allgemein zugänglichen Schnittstelle mit dem Namen NameBeanFinderHelper definiert sein, wobei Name der Name des Entity-Beans ist [...]*

> *Die Logik muss in einer Zeichenfolgekonstanten mit dem Namen findMethodNameQueryString enthalten sein, wobei findMethodName der Name der Finder-Methode ist. Die Zeichenfolgekonstante kann keine oder mehrere Fragezeichen (?) enthalten, die beim Aufruf der Methode von links nach rechts durch den Argumentenwert der Finder-Methode ersetzt werden.“*
>
> ([Web01b])

Für die Beans `BedingungBean`, `ObjektBean` und `MerkmalWertBean` (inklusive aller erbenden Bean) sind im Zuge der Portierung also noch die entsprechenden Interfaces zu erstellen.

6.4.6 Server- und Container APIs

In den Beschreibungen der Applikationsserver WebLogic und PowerTier wurden unter anderem Server- und Container APIs vorgestellt, die bei der Arbeit mit EJB-basierten Anwendungen verwendet werden können und über die spezielle, nur auf dem jeweiligen Server verfügbare Leistungen zur Verfügung stehen. Auch im WebSphere von IBM sind solche APIs enthalten. Sie üben allerdings (ebenso wie beim PowerTier) keinen Einfluß auf die Portierung aus, da sie in der zu portierenden Applikation nicht verwendet werden und vermutlich auch nicht verwendet werden müssen.

6.4.7 Vererbungen

In Kapitel 5.4 wurden die technischen Eigenschaften der Beispielapplikation beschrieben. Zu diesen zählen auch zwei Arten von Vererbung - die „komponenteninterne“ und die „komponentenübergreifende“ Vererbung. IBM geht in der Dokumentation des WebSphere kurz auf das allgemeine Thema *Vererbung* ein. Es finden sich in diesem Zusammenhang die folgenden Hinweise:

> *„Vererbung durch ferne Objekte*
>
> *Eine Enterprise-Bean oder ein anderes fernes Objekt kann aufgrund der Spezifikation von Java-IDL-Zuordnungen nicht von zwei Schnittstellen übernehmen, die Methoden mit demselben Namen enthalten, selbst wenn diese Methoden andere Kennungen besitzen. Java-Programmierer, die mit dem herkömmlichen Java-Vererbungsmodell vertraut sind, sollten diese Einschränkung der Spezifikation unbedingt beachten. Nach der Spezifikation für Enterprise JavaBeans (EJB) können Enterprise-Beans nicht so geschrieben werden, dass sie von zwei Schnittstellen übernehmen. Wird diese Einschränkung nicht berücksichtigt, treten bei der Publikation Fehler auf.“*
>
> ([Web01b])

Es wird allerdings nicht deutlich, auf welche Art von Vererbung sich diese Stellen beziehen. Da in der Beispielapplikation jedoch beide Vererbungsmechanismen verwendet werden, ist zu vermuten, daß es bei der Portierung einiger Beans, die eine Vererbung enthalten, durchaus zu Portierungsproblemen kommen kann.

6.4.8 Zusammenfassung

Die theoretische Betrachtung des WebSphere-Applikationsservers hat sich größtenteils mit der Beschreibung der Vorgänge beim Umgang mit EJBs bzw. den geplanten Abläufen der Portierung in Kapitel 8 beschäftigt. Dabei hat sich herausgestellt, daß die Portierung auf den WebSphere vermutlich weniger aufwendig ist, als die Portierung auf den PowerTier. Diese Vermutung ergibt sich durch folgende Umstände:

1. Sowohl der WebSphere, als auch der WebLogic setzen auf der EJB-Spezifikation 1.0 auf. Portierungsprobleme, die sich zum Beispiel bei der Portierung auf den PowerTier durch Spezifikationsunterschiede ergeben, sind somit nicht zu erwarten.

2. Im Leistungsumfang des WebSphere ist nur der Applikationsserver enhalten. Tools, die den Quellcode beeinflussen, sind nicht vorhanden. Einzig das Jetace-Tool zum Erzeugen der DeploymentDescriptoren wird mitgeliefert. Im Vergleich zum PowerTier, der gewissermaßen ein „All-In-One“ Paket für die Arbeit mit EJBs zur Verfügung stellt, splittet IBM seine die EJB-Technologie unterstützenden Produkte auf (WebSphere als Applikationsserver, Visual Age als Entwicklungsumgebung, DB2 als Datenbank, etc.). Diese Aufsplittung läßt eine unabhängige Verwendung dieser einzelnen Tools vermuten, so daß für die Portierung auch nur der Server benötigt wird.

Die Thesen hinsichtlich der Portierungsprobleme beschränken sich für diesen Server somit auf folgende Bereiche:

Die implementierten Vererbungen können zu Problemen führen, da das Thema „Vererbung“ in der Dokumentation eingeschränkt wird. Für welche Art von Vererbung diese Einschränkung gilt (Vererbungen innerhalb von Komponenten oder Vererbungen über Komponentengrenzen hinweg), kann der Dokumentation leider nicht eindeutig entnommen werden. Eine Antwort auf diese Frage wird daher von den Ergebnissen der Portierung erwartet.

Container-Managed EntityBeans erfordern laut Dokumentation unter Umständen den Einsatz weiterer Tools, wie zum Beispiel der Entwicklungsumgebung *Visual Age für Java*. Ob die Portierung nicht doch auf eine weniger aufwendige Art und Weise durchgeführt werden kann, bleibt abzuwarten.

Vom Entwickler benötigte Finder-Methoden müssen im Zusammenhang mit Container-Managed EntityBeans in einem eigenen Interface implementiert werden, da der Server ansonsten nicht weiß, wie er diese speziellen Finder intern implementieren muß. Für die Portierung der Container-Managed EntityBeans `BedingungBean`, `ObjektBean` und `MerkmalWertBean` (inklusive aller erbenden Beans) müssen diese Finder somit noch erstellt und der Anwendung hinzugefügt werden.

Analog zur Portierung auf den PowerTier werden auch bei dieser Portierung keine Probleme erwartet, deren Ursache in der Verwendung serverspezifischer Server- und Container-APIs liegt (Kapitel 6.4.6).

Kapitel 7

Portierung auf PowerTier

Aufbauend auf den bisher geschaffenen Grundlagen wird in diesem Kapitel die erste Portierung behandelt. Als Zielserver für die Beispielapplikation wird dazu der Applikationsserver *PowerTier* von Persistence verwendet. Die Portierung selbst soll Komponente für Komponente erfolgen. Dabei werden allerdings Fehler und Probleme, die bei mehr als einer Komponente auftreten, nicht stets aufs Neue ausführlich aufgeführt. Stattdessen werden diese Fehler und Probleme beispielhaft an einer Komponente erörtert und anschließend bei einem erneuten Auftreten nur kurz erwähnt.

7.1 Grundlagen

7.1.1 Verzeichnisstruktur

Nach der Installation des Servers wurde zunächst folgende Verzeichnisstruktur angelegt:

```
original\
    ps-makeejb.cfg
    bedingung\
    client\
    historisierung\
    inkasso\
    merkmal\
    partner\
    police\
    praemienangleichung\
    produkt\
    tarif\
    vendor\
    verteiler\
```

```
    vertrag\
prototyp\
    ps-makeejb.cfg
    bedingung\
    client\
    historisierung\
    inkasso\
    merkmal\
    partner\
    police\
    praemienangleichung\
    produkt\
    tarif\
    vendor\
    verteiler\
    vertrag\
```

Die Verzeichnisse unterhalb der `original`-Ebene enthalten jeweils die entsprechenden `.java`-Dateien der Beispielapplikation. Sämtliche Änderungen an diesen Dateien, die im Laufe der Portierung notwendig werden, werden auch in diesen Verzeichnissen vorgenommen. Anschließend sind die geänderten Dateien in das entsprechende Verzeichnis unterhalb der `prototyp`-Ebene zu kopieren. Auf dieser Ebene werden dann auch die notwendigen Aktionen, wie zum Beispiel das Erstellen der DeploymentDescriptoren, durchgeführt. Dies hat den Vorteil, daß im Falle eines eventuellen Fehlers alle Dateien eines Verzeichnisses der `prototyp`-Ebene gelöscht und durch die `.java`-Dateien der `original`-Ebene ersetzt werden können. Somit ist gesichert, daß der PowerTier keine Fehler aufgrund alter Dateien erzeugt. Dies können zum Beispiel alte und defekte `.class`-Dateien sein, die zur `prototyp`-Ebene hinzugekommen sind. Es ist in diesem Zusammenhang anzumerken, daß diese doppelte Verzeichnisstruktur nicht unbedingt notwendig ist. Da aus Projekten mit dem PowerTier jedoch bekannt ist, daß dieser desöfteren Probleme mit Dateien von mißlungenen Aktionen hat, erscheint dieses Vorgehen durchaus sinnvoll.

7.1.2 DeploymentDescriptoren

DeploymentDescriptoren können mittels des Befehls `ps-makeejb -createDD` generiert werden (Kapitel 6.3.6). Dabei wird für jede Bean eine Datei mit dem Namen `<BeanName>.des` erzeugt. Diese enthält Informationen über die entsprechende Bean und wird später in einer serialisierten Form vom Server benötigt. Die zum Erzeugen der DeploymentDescriptoren notwendige Konfigurationsdatei `ps-makeejb.cfg` enthält im Kontext der Portierung die zwei folgenden Einträge:

```
PROJECT_NAME=prototyp
PACKAGE_NAME=bedingung
```

Der `PROJECT_NAME` ist dabei von eher untergeordneter Bedeutung, da er lediglich einen beliebigen Namen des gesamten Projekts enthält. Wichtig ist an dieser Stelle der `PACKAGE_NAME`, der den Namen der jeweiligen Komponente beinhaltet, auf die sich der `ps-makeejb`-Befehl bezieht.

Zu Anfang dieses Kapitels wurde bereits die verwendete Verzeichnisstruktur aufgeführt. Um nun die DeploymentDescriptoren einer Komponente zu erzeugen, müssen folgende Schritte durchgeführt werden:

1. Eintragen der zu bearbeitenden Komponente in die `ps-makeejb.cfg`
2. Ausführen des Befehls `ps-makeejb -createDD` im Verzeichnis `prototyp`

Es wäre auch möglich, für alle Komponenten einmalig einen Eintrag in die `ps-makeejb.cfg` vorzunehmen. Diese sähe dann wie folgt aus:

```
PROJECT_NAME=prototyp
PACKAGE_NAME=bedingung
PACKAGE_NAME=historisierung
PACKAGE_NAME=inkasso
PACKAGE_NAME=merkmal
PACKAGE_NAME=partner
PACKAGE_NAME=police
PACKAGE_NAME=praemienangleichung
PACKAGE_NAME=produkt
PACKAGE_NAME=tarif
PACKAGE_NAME=verteiler
PACKAGE_NAME=vertrag
```

und hätte zur Folge, daß für alle angegebenen Komponenten die entsprechenden JAR-Archive in der angegebenen Reihenfolge erzeugt würden. Um jedoch die Fehler, die sich beim Kompilieren der einzelnen Komponenten ergeben, überschaubar zu halten und die einzelnen Komponenten Schritt für Schritt portieren zu können, soll jeweils nur ein `PACKAGE_NAME` eingetragen werden.

7.2 Geplantes Vorgehen

Die eigentliche Portierung gliedert sich in zwei Teile. Im ersten Teil wird der folgende Ablauf für jede einzelne Komponente durchlaufen:

1. Erstellen der Konfigurationsdatei `ps-makeejb.cfg` (vgl. Kapitel 6.3.7)

2. Generieren der DeploymentDescriptoren der in der Komponente enthaltenen Beans (`.des`-Format, Kapitel 6.3.7)

3. Anpassen der Werte der DeploymentDescriptoren

4. Generieren des JAR-Archivs mittels des `ps-makeejb`-Befehls (s. Kapitel 6.3.6)

Bei den Punkten zwei und vier ist zu beachten, daß die Beseitigung eventueller Fehler, die im Kontext des jeweiligen Vorgangs auftreten können, eingeschlossen ist. Abbildung 7.1 stellt diesen geplanten ersten Teil grafisch dar.

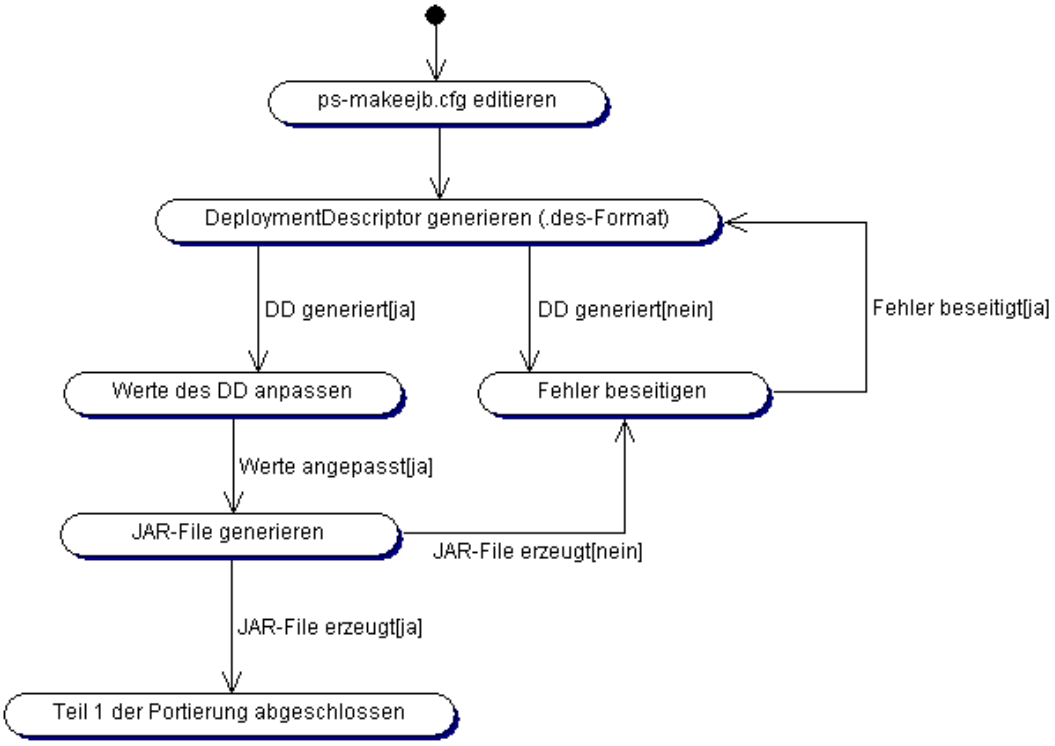

Abbildung 7.1: Vorgehen bei der Portierung auf den PowerTier

Anschließend müssen in einem zweiten Schritt die erzeugten JAR-Archive noch im Server deployed werden und die komplette Applikation muß auf ein korrektes Verhalten hin getestet werden. Auf den Test der portierten Beispielapplikation soll allerdings nur eingegangen werden, wenn die Portierung fehlschlägt und die Ergebnisse der Diplomarbeit von Malte Hülder nicht reproduziert werden können. Ist das korrekte Verhalten schließlich gewährleistet, so werden in Kapitel 9 noch die theoretischen Ergebnisse aus Kapitel 6.3 mit den praktischen Ergebnissen dieses Kapitels verglichen werden.

7.3 Die Anbindung an die Datenbank

Zu Beginn der Portierung ist bereits abzusehen, daß diverse Änderungen im Zusammenhang mit der Anbindung an die Datenbank notwendig sind. Diese Notwendigkeit ergibt sich durch die Tatsache, daß in den einzelnen EntityBeans jeweils über die Methode `getConnection()` eine Datenbankverbindung aufgebaut wird, die über eine WebLogic-spezifische URL (`jdbc:weblogic:oracle`) angesprochen wird. Diese URL wird über den `Context` der Beans aus dem DeploymentDescriptor eingelesen. Um diese Abhängigkeit zum WebLogic zu unterbinden, wurde die URL der Beispielapplikation in allen DeploymentDescriptoren der portierten Anwendung durch eine von jeglichem Applikationsserver unabhängige URL (`jdbc:oracle:thin:@localhost:1521:ejbtest`) ersetzt (vgl. `EnvironmentProperty`, Kapitel 7.5.2). Außerdem wurde in allen Beans die Methode `getConnection()` um die Zeilen

```
String driver = environmentProps.getProperty(''jdbcDriver'');
Class driverClass = Class.forName(driver);
driverClass.newInstance();
```

erweitert, die für das Laden des JDBC-Treibers erforderlich sind. Der Treiber selbst wird ebenfalls im DeploymentDescriptor angegeben und über den `Context` der jeweiligen Bean ermittelt.

Das beschriebene Vorgehen zur Auflösung der Abhängigkeit vom WebLogic läßt vermuten, daß diese Abhängigkeit lediglich durch die Abhängigkeit zu einem Datenbanktreiber (in diesem Fall dem Oracle-Treiber) ersetzt wurde. Eine Erklärung, warum dies nicht so ist, wird in Kapitel 9 diskutiert.

7.4 Die Klasse vendor

Die bereits in Kapitel 5.4 beschriebene Klasse `vendor` stellt keine Serverkomponente dar, die Geschäftslogik enthält und die im Server installiert wird. Stattdessen stellt Sie den einzelnen Serverkomponenten Methoden zur Verfügung, die speziell für die Arbeit mit dem WebLogic nötig sind. Im Beispiel enthält sie dazu lediglich die Methode `getInitialContext(String url)`. Mit ihrer Hilfe wird der InitialContext ermittelt, der dazu verwendet wird, über die JNDI-Schnittstelle die HomeInterfaces der einzelnen Beans zu finden. Da die EJB-Spezifikation die Implementierung des JNDI-Dienstes jedoch den Serveranbietern überläßt, unterscheiden sich die entsprechenden Quellcodestellen von Server zu Server und müssen jedesmal entsprechend angepaßt werden.

In der Beispielanwendung werden dem InitialContext diverse Properties (Umgebungsvariablen) zugeordnet, die für die Verwendung des JNDI-Dienstes nötig sind. So wird zum Beispiel, ähnlich wie bei Webanwendungen, eine URL

benötigt, über die der Server angesprochen werden kann. Im Fall der Beispielanwendung hat diese URL stets den Wert `t3://localhost:7001` und wird der Variablen `Context.PROVIDER_URL` zugewiesen.

Das Anpassen der Klasse `vendor` an den PowerTier bedurfte grundlegender Änderungen, die allerdings dazu führten, daß von der ursprünglichen Klasse nichts mehr übrig blieb:

- Die in der Beispielapplikation gesetzten Properties für die `Context.INITIAL_CONTEXT_FACTORY` und die `Context.PROVIDER_URL` werden nun in einer Server-Textdatei `context.properties` abgelegt und von der `vendor`-Klasse eingelesen. Dies hat den Vorteil, daß ein erneutes Kompilieren der Klasse nur bei Änderungen am Quellcode notwendig wird (z.B. beim Einfügen neuer Methoden). Einfache Änderungen der Properties-Werte (z.B. eine neue `PROVIDER_URL`) können schnell und problemlos an der Textdatei vorgenommen werden

- Die Methode `getInitialContext(String url)` wurde komplett entfernt. Stattdessen wurde die Methode `getHome(String homeName)` implementiert, die an beliebiger Stelle der Applikation eine Referenz auf das gewünschte HomeInterface `homeName` zurückliefert. Für das `PartnerHome`-Interface sieht der Befehl zum Beispiel wie folgt aus:

  ```
  PartnerHome p = (PartnerHome) vendor.getHome(''PartnerHome'');
  ```

  Der Unterschied zur Beispielapplikation liegt darin, daß die Möglichkeit zum Auffinden eines HomeInterface nun an zentraler Stelle umgesetzt wurde. Bisher enthielt jedes einzelne Bean eine eigene `getInitialContext()`-Methode und lediglich der Client hat wirklich auf die Hilfsklasse `vendor` zurückgegriffen. Im Zuge der Portierung hätte somit die `getInitialContext()`-Methode in allen Beans und in der Klasse `vendor` angepaßt werden müssen. Durch die neue `vendor`-Klasse muß die Anpassung an einen neuen Server nur noch einmal in der oben genannten Textdatei durchgeführt werden

7.5 Die Komponente bedingung

7.5.1 Erzeugen des DeploymentDescriptors

Bereits bei der Generierung des ersten DeploymentDescriptors waren erste wesentliche Unterschiede zwischen den verwendeten Applikationsservern festzustellen. Zunächst wurde (wie in Kapitel 7.1.2 beschrieben) die Konfigurationsdatei `ps-makeejb.cfg` mit folgendem Inhalt angelegt:

```
PROJECT_NAME=prototyp
PACKAGE_NAME=bedingung
```

Mit dem anschließend ausgeführten Befehl `ps-makeejb -createDD` im Verzeichnis `prototyp` sollte dann der DeploymentDescriptor `bedingung.des` erzeugt werden. Hierbei wurde allerdings folgender Fehler ausgegeben:

```
PowerTier (TM) Domain Builder v6.00
Compiling Java source files ...
Compiled Java source files.
Domain Builder status: com.persistence.tools.dpi.BuildException:
Unable to load class: bedingung.Bedingung. Check your CLASSPATH
setting...
```

Die Ursache dieses Fehlers ist jedoch nicht im `CLASSPATH` begründet, sondern ist in der Namensgebung für die einzelnen Beanbestandteile zu suchen. Die Beans der zu portierenden Applikation sind nach folgendem Schema benannt:

- `HomeInterface` : `<BeanName>Home.java`
- `RemoteInterface` : `<BeanName>Remote.java`
- `BeanClass` : `<BeanName>Bean.java`
- `PrimaryKey` : `<BeanName>PK.java`

Dabei ist `<BeanName>` jeweils durch den Namen des entsprechenden Beans zu ersetzen (z.B. `PartnerHome.java` für das HomeInterface des Partner-Beans). Da die EJB-Spezifikation bisher allerdings in keiner Version eine spezielle Namensgebung für die einzelnen Beanbestandteile vorschreibt, gibt es bei den von Malte Hülder verwendeten Namen im Zusammenspiel mit dem WebLogic-Server zunächst keine Probleme. Der PowerTier hingegen akzeptiert zwar die Namensgebung für das HomeInterface, die BeanClass und den PrimaryKey, nicht jedoch die des RemoteInterface. Für dieses setzt der PowerTier das Format `<BeanName>.java` voraus (anstatt also `PartnerRemote.java` muß es `Partner.java` heißen). Was zunächst nach dem einfachen Umbenennen der `RemoteInterface`-Dateien klingt, bringt eine unter Umständen aufwendige Bearbeitung aller weiteren Beans mit sich. So müssen nicht nur die Dateinamen der RemoteInterfaces bearbeitet werden, sondern auch alle Quellcodefragmente, die im Zusammenhang mit einer Referenz auf ein `RemoteInterface` stehen.

Um diesen Fehler zu beseitigen wurde also das RemoteInterface `bedingungRemote.java` in `bedingung.java` umbenannt und die enthaltene Codezeile

```
public interface BedingungRemote extends javax.ejb.EJBObject {
```

wurde in

```
public interface Bedingung extends javax.ejb.EJBObject {
```

umgeschrieben. Nach dieser Änderung ließ sich der DeploymentDescriptor wie gewünscht erzeugen. Eine Bestätigung dieser Namensgebung für den PowerTier findet sich zum Beispiel in folgendem Zitat der PowerTier-Dokumentation:

> „*You must create a remote interface Class and a home interface ClassHome for each object in your project's object model.*“
> ([Per01, Programming Guide, S.147])

Der Begriff *Class* entspricht hier dem obengenannten *BeanName*.

7.5.2 Anpassen des DeploymentDescriptors

Nachdem ein DeploymentDescriptor erzeugt wurde, müssen die in ihm enthaltenen Werte noch an die jeweilige Bean angepaßt werden. Das ist notwendig, da der Applikationsserver über diese Werte zum Beispiel Informationen zum Persistenz- oder Sicherheitsmanagement der jeweiligen Bean erhält. Generell gibt es bei jedem Server Werte, die speziell für diesen vorgesehen sind und Werte, die im Grunde genommen jeder Applikationsserver verwendet und die sich dann meist nur durch ihre Syntax unterscheiden. Da die DeploymentDescriptoren der Beispielapplikation nicht nur als serialisierte Dateien, sondern auch als einfache Textdateien vorliegen, besteht somit die Hoffnung, daß einige Werte in der portierten Anwendung übernommen werden können.

Für den DeploymentDescriptor des `BedingungBean` wurde die Datei `BedingungDescriptor.txt` (Komponente `bedingung` der Beispielapplikation) als Grundlage verwendet. Das Anpassen des DeploymentDescriptors wird an dieser Stelle einmalig detailliert aufgeführt, da er sich bei allen weiteren Beans nur minimal unterscheidet. Im weiteren Verlauf der Arbeit werden daher lediglich wesentliche Unterschiede der DeploymentDescriptoren dargestellt.

Zunächst aber der DeploymentDescriptor der Beispielapplikation:

```
(EntityDescriptor
  beanHomeName                          BedingungHome
  enterpriseBeanClassName               bedingung.BedingungBean
  homeInterfaceClassName                bedingung.BedingungHome
  remoteInterfaceClassName              bedingung.BedingungRemote
  isReentrant                           false
  (accessControlEntries
    DEFAULT                             [system guest]
  ); end accessControlEntries
```

```
  (controlDescriptors
    (DEFAULT
      isolationLevel                    TRANSACTION_READ_COMMITTED
      transactionAttribute              TX_REQUIRED
      runAsMode                         CLIENT_IDENTITY
    ); end DEFAULT
  ); end controlDescriptors
  (environmentProperties
    idleTimeoutSeconds                  5
    password                            demo
    Tabelle                             BEDINGUNG
    maxBeansInCache                     1000
    user                                DEMO
    jdbcURL                             jdbc:weblogic:oracle
    maxBeansInFreePool                  20
    (clusterProperties
    ); end clusterProperties
    (persistentStoreProperties
      persistentStoreType               jdbc
      (jdbc
        tableName                       BEDINGUNG
        dbIsShared                      false
        poolName                        demoPool
        (attributeMap
          Text                          TEXT
          id                            ID
        ); end attributeMap
        (finderDescriptors
        ); end finderDescriptors
      ); end jdbc

    ); end persistentStoreProperties
  ) ;end environmentProperties
  ; Entity bean-specific properties:
  primaryKeyClassName                   bedingung.BedingungPK
  containerManagedFields                [id Text]

); end DeploymentDescriptor
```

Von diesem DeploymentDescriptor konnten im Zuge der Portierung die meisten Werte übernommen werden. Dabei stellten sich Unterschiede bzw. Probleme wie folgt dar:

- Der Wert der Variablen `enterpriseBeanClassName` mußte aufgrund des Problems bei der Namensgebung der RemoteInterfaces von `bedingung.BedingungRemote` in `bedingung.Bedingung` geändert werden. Im DeploymentDescriptor des PowerTier wird dieser Wert der Variablen `RemoteInterfaceClassName` zugewiesen.

- Über das `transactionAttribute` wird sowohl beim WebLogic, als auch beim PowerTier das Transaktionsverhalten der einzelnen Methoden gesteuert. In der Beispielapplikation wird dazu allen Methoden des `BedingungBean` der Defaultwert `TX_REQUIRED` zugewiesen. Der PowerTier hingegen weist jeder Methode ein eigenes Transaktionsverhalten zu. Dabei wurde den Methoden ausschließlich der Wert `TX_REQUIRED` oder `TX_SUPPORTS` zugewiesen. Der Wert `TX_REQUIRED` bezieht sich auf Methoden, die aufgrund ihres Verhaltens einen eigenen Transaktionskontext benötigen. Existiert dieser Kontext bereits, wird die entsprechende Methode in diesem ausgeführt, ansonsten wird ein neuer Kontext erzeugt. Methoden, denen der Wert `TX_SUPPORTS` zugewiesen wird, benötigen einen derartigen Kontext nicht und nutzen diesen auch nur dann, falls er bereits existiert.Als Beispiele für Methoden, die einen Transaktionskontext benötigen, können alle Methoden dienen, die schreibend auf eine Datenbank zugreifen. Um Probleme und Inkonsistenzen zu vermeiden müssen die Daten innerhalb einer Transaktion ohne Unterbrechungen oder Seiteneffekte von außen geschrieben werden. Im Gegensatz dazu können alle Methoden, die lediglich Daten aus der Datenbank lesen, als ein Beispiel für die Verwendung des Wertes `TX_SUPPORTS` verwendet werden. Da das `BedingungBean` keine kritischen Transaktionen durchführt, konnten die generierten Werte übernommen werden, so daß eine Bearbeitung der Werte aller Methoden nicht nötig war.

- Die Umgebungsvariablen einer Bean werden unter den `environmentProperties` (WebLogic) bzw. `EnvironmentProperty` (PowerTier) abgelegt. Diese können innerhalb der Beans über einen `Context` abgerufen werden und enthalten sowohl serverspezifische, als auch serverunabhängige Informationen. Die serverspezifischen Informationen `idleTimeoutSeconds`, `maxBeansInCache` und `maxBeansInFreePool` konnten nicht übernommen werden, da der PowerTier diese Parameter so oder in ähnlicher Form nicht kennt. Von den serverunabhängigen Informationen konnten bis auf eine Ausnahme alle Werte weiterverwendet werden, da diese Informationen unabhängig vom Server zum Aufbau der Datenbankverbindung verwendet werden (User, Passwort und Tabellenname). Einzig die `jdbcURL` wurde durch den Wert `jdbc:oracle:thin:@localhost:1521:ejbtest` ersetzt (vgl. Kapitel 7.3), da sie zuvor mit dem WebLogic-spezifischen Wert `jdbc:weblogic:oracle` belegt war.

- Der `primaryKeyClassName` gibt dem WebLogic an, wie der PrimaryKey des jeweiligen EntityBeans bezeichnet wird. PowerTier vergibt hier unter der selben Bezeichnung standardmäßig den Wert `<BeanName>Key`, wohingegen in der Beispielapplikation die Bezeichnung `<BeanName>PK` verwendet wird. Im Gegensatz zu den RemoteInterfaces ergaben sich jedoch bei den PrimaryKeys keine Probleme mit den unterschiedlichen Bezeichnungen.

- Im Bereich `containerManagedFields` werden dem WebLogic bei Container-Managed EntityBeans diejenigen Attribute mitgeteilt, die vom Container persistent in der Datenbank verwaltet werden sollen. Auch diese wurden im generierten DeploymentDescriptor an der entsprechenden Stelle eingetragen.

7.5.3 Erzeugen des JAR-Archivs

Im Anschluß an den DeploymentDescriptor galt es, ein JAR-Archiv der Komponente `bedingung` zu erzeugen. Hier ergab sich allerdings sofort zu Beginn ein weiteres Problem. In Kapitel 5.4 wurde bereits erwähnt, daß in der zu portierenden Applikation sowohl Bean-Managed EntityBeans, als auch Container-Managed EntityBeans Verwendung finden. Die Rahmenbedingungen der Arbeit (Kapitel 4.4) sehen vor, daß der jeweils verwendete Persistenzmechanismus beibehalten wird. Im Falle der Komponente `bedingung` soll also weiterhin Container-Managed Persistence verwendet werden.

An dieser Stelle bestätigt sich ein Problem, auf welches bereits in Kapitel 4.2.1 hingewiesen wurde. Da der PowerTier auf einer anderen EJB-Spezifikation als der WebLogic aufsetzt und diese Spezifikationen sich in den Syntax-Vorgaben für Container-Managed EntityBeans unterscheiden, muß der Quellcode des `BedingungBean` von der EJB-Spezifikation 1.0 (WebLogic) zur EJB-Spezifikation 1.1 (PowerTier) portiert werden. Für diese Portierung konnten vor allem bei Monson-Haefel ([MH99b, S.306 ff.]) und Merkle ([Mer00a]) Hinweise auf die nötigen Änderungen gefunden werden, die dann auch auf das `BedingungBean` angewendet wurden. Da sich im Anschluß an diese Änderungen ein JAR-Archiv der Komponente `bedingung` ohne weitere Probleme erzeugen ließ, wurden zunächst die weiteren Komponenten portiert. Auch die zwei anderen Container-Managed EntityBeans `ObjektBean` und `MerkmalBean` wurden an die neue Spezifikation angeglichen. Nach dem Start des PowerTier mit der portierten Anwendung kam dann ein Hinweis, daß 22 Beans im Server installiert seien, darunter jedoch keine Container-Managed EntityBeans. Beim anschließenden Test der Applikation wurden beim Zugriff auf diese Beans stets zahlreiche PowerTier-spezifische Exceptions geworfen, die jedoch keine Rückschlüsse auf das eigentliche Problem zuließen. Auch die PowerTier-Dokumentation lieferte hierzu keine Hinweise. Sie beschäftigt sich hauptsächlich mit der Entwicklung solcher Beans mittels des PowerTier Builders und nicht mit der Entwicklung

„per Hand“.

Ein weiteres Problem bestand in der Anpassung des DeploymentDescriptors. Wie bereits beschrieben, wurde im generierten DeploymentDescriptor des `BedingungBean` auch der Abschnitt `ContainerManagedField` mit den notwendigen Attributen `id` und `Beschreibung` gefüllt. Da diese Werte vom PowerTier beim Erzeugen des JAR-Archivs wieder automatisch entfernt wurden, blieb lediglich die Schlußfolgerung, daß der PowerTier diese portierten Beans nicht als Container-Managed EntityBeans anerkennt. Die endgütlige Lösung dieses Problems bestand nun in der Verwendung des PowerTier Builders. Obwohl diesem bei der theoretischen Betrachtung in Kapitel 6 kein Einfluß auf die geplante Portierung zugeschrieben wurde, da der PowerTier Builder für die Entwicklung von EJBs konzipiert ist, mußte er doch an dieser Stelle eingesetzt werden. Die eigentliche Idee bestand nun darin, die Komponente `bedingung` mit dem PowerTier Builder nachzubilden, einen Rahmenquellcode erzeugen zu lassen und anschließend die fehlende Geschäftslogik der Beispielapplikation zu übernehmen.

7.5.4 Einsatz des PowerTier Builder

Die Umsetzung des `BedingungBean` durch den PowerTier Builder verlief ohne Probleme. Genau wie in Kapitel 6.3.4 am Beispiel beschrieben wurde die Klasse `Bedingung` modelliert, der die zwei Attribute `id` (PrimaryKey vom Typ `Integer`) und `Text` (vom Typ `Text`, welcher im generierten Quellcode dem Typ `String` entspricht) zugewiesen wurden. Desweiteren konnten auch die Angaben zum Mapping auf die Datenbank vorgegeben werden. Die Datenbanktabelle trug in diesem Fall den Namen `BEDINGUNG` und wurde als existent und nicht schreibgeschützt gekennzeichnet. Abbildung 7.2 zeigt die modellierte Klasse im PowerTier Builder. Aufgrund ihrer einfachen Struktur ergaben sich beim modellieren keine Probleme. Der generierte Quellcode ließ sich anschließend problemlos in die Anwendung integrieren und mußte lediglich um einige Imports und die zwei Methoden `findeAlleBedingungen()` und `getDatenobjekt()` erweitert werden. Diese beiden Methoden stellen die komplette Geschäftslogik des `BedingungBean` der Beispielapplikation dar. Der abschließende Test verlief ohne weitere Probleme.

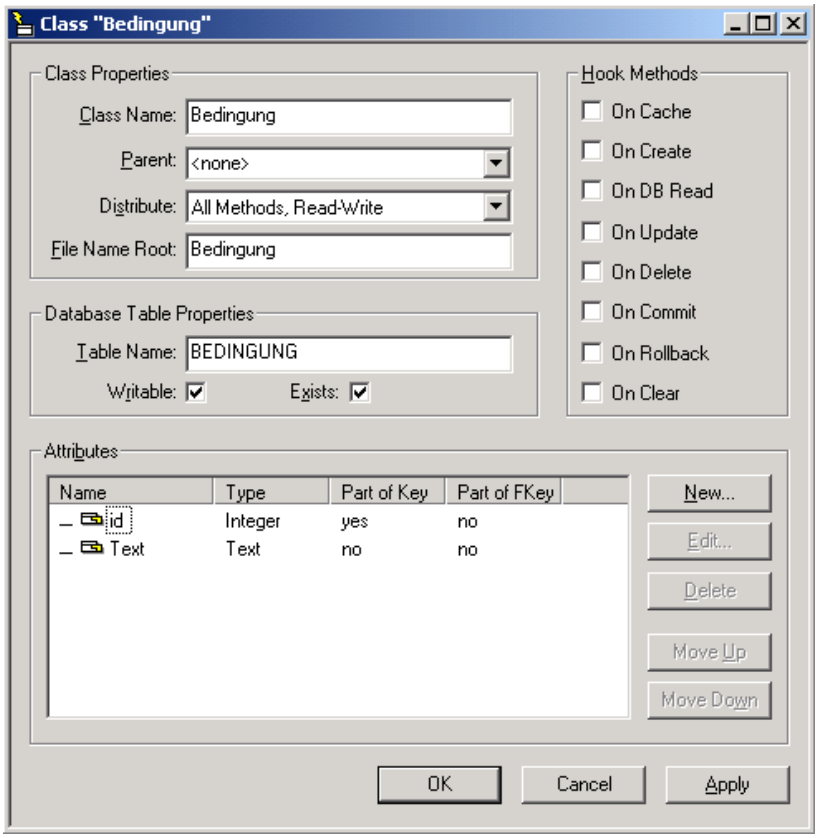

Abbildung 7.2: Modellierung des BedingungBean

7.6 Die Komponente inkasso

Die Komponente `inkasso` besteht nur aus dem SessionBean `MonatsinkassoBean`. Dieses Bean stellt zwar mit der Geschäftsmethode `holePolicedaten()` Geschäftslogik zur Verfügung, jedoch wird diese Methode in der Beispielapplikation noch nicht verwendet. Dies liegt daran, daß im Zuge eines Monatsinkasso die Daten einer bestimmten `Police` an ein externes Buchungssystem übergeben werden sollen. Da ein solches System im Rahmen der Diplomarbeit nicht zur Verfügung steht und mehr für zukünftige Weiterentwicklungen der Beispielapplikation gedacht ist, spielt diese Komponente eine untergeordnete Rolle.

Um die Portierung durchzuführen wurden alle Referenzen zum RemoteInterface des `MonatsinkassoBean` an die Namensgebung des PowerTier angepaßt und die Methode `getConnection()` zur Datenbankanbindung an den neuen Treiber angepaßt (vgl. Kapitel 7.3). Der Test, ob die Komponente korrekt portiert wurde, mußte aufgrund der oben genannten Gründe entfallen.

7.7 Die Komponente merkmal

Die Komponente `merkmal` nimmt innerhalb der Beispielapplikation aus zwei Gründen einen besonderen Stellenwert ein:

1. Lediglich die Komponente `merkmal` enthält eine „komponenteninterne Vererbung“. Der Begriff *komponentenintern* ist in diesem Zusammenhang so zu interpretieren, daß Vererbungen nur zwischen den Beans einer Komponente existieren. Betroffen sind davon die Beans `AuswahlMerkmalBean`, `TextMerkmalBean` und `ZahlMerkmalBean`, die jeweils von der Bean `MerkmalWertBean` erben. Eine „komponentenübergreifende Vererbung“, bei der eine Bean auch von einer Bean einer anderen Komponente erbt, wird in Kapitel 7.8 untersucht.

2. Die Komponente `merkmal` enthält sowohl Bean-Managed, als auch Container-Managed EntityBeans.

Zu Beginn der Portierung wurde zunächst das bereits bekannte Namensproblem aufgelöst, indem alle RemoteInterface-Referenzen an die Namensgebung des PowerTier angepaßt wurden (`<BeanName>Remote` in `<BeanName>`). Anschließend sollte zunächst das einzige Bean-Managed EntityBean portiert werden und anschließend die Container-Managed EntityBeans, zwischen denen zugleich die erwähnte Vererbungsbeziehung besteht.

7.7.1 Portierung des Bean-Managed EntityBean MerkmalTyp

Im Zuge der Portierung des einzigen Bean-Managed EntityBean `MerkmalTyp` waren abgesehen von der Anpassung an die Namensgebung des PowerTier lediglich zwei weitere, bereits bekannte Probleme zu lösen:

1. Die `getInitialContext()`-Methode wurde durch die neue `getHome()`-Methode der Hilfsklasse `vendor` (vgl. Kapitel 7.4) ersetzt.

2. Die Methode `getConnection()`, die zum Aufbau einer Datenbankverbindung verwendet wird (vgl. Kapitel 7.3), wurde an die Vorgaben aus Kapitel 7.3 angepaßt.

7.7.2 Vererbung der Container-Managed EntityBeans

Die Portierung der Container-Managed EntityBeans konnte relativ zügig durchgeführt werden, ohne daß neue Probleme auftraten. Aufgrund der Ergebnisse der Portierung der Komponente `bedingung` wurde sofort auf den PowerTier Builder zurückgegriffen. Mit diesem wurden die Beans `AuswahlMerkmalBean`,

`TextMerkmalBean`, `ZahlMerkmalBean` und `MerkmalWertBean` modelliert, wobei die oben beschriebene Vererbung ohne Probleme übernommen werden konnte. Aus dem Modell ließ sich Quellcode generieren, der anschließend an die Geschäftslogik der Beispielapplikation angepaßt wurde.

Auch die generierten DeploymentDescriptoren mußten nur an wenigen Stellen an die Vorgaben der Beispielapplikation angepaßt werden. Die wichtigste Änderung betraf das Transaktionsverhalten der `createMerkmalWert(MerkmalObjekt)`-Methode des SessionBeans `MerkmalVerwaltungBean`. Der Aufruf dieser Methode führte in der portierten Applikation zu einem *Deadlock*. Erst als dieser Methode über den DeploymentDescriptor explizit ein eigener Transaktionskontext zugewiesen wurde (über das Transaktionsattribut `TX_REQUIRES_NEW`), konnte ein Deadlock vermieden werden. Die Ursache für dieses Problem ist in der internen Realisierung von Transaktionen der einzelnen Server zu suchen. Bei der Verwendung des WebLogic reicht es aus, dieser Methode über das Transaktionsattribut `TX_REQUIRED` mitzuteilen, daß sie in einem Transaktionskontext ausgeführt werden soll. Falls ein solcher Kontext nicht existiert, muß er erzeugt werden, ansonsten wird die Methode in einem bereits bestehenden Kontext ausgeführt, der zum Beispiel von einer aufrufenden Methode erzeugt wurde. Der PowerTier hingegen verlangt für diese Methode bei jedem Aufruf einen eigenen Transaktionskontext. Dieses Portierungsproblem bereitet insgesamt gesehen keine großen Probleme, da das Transaktionsverhalten der einzelnen Methoden über den DeploymentDescriptor gesteuert wird. Da dieser für die einzelnen Server bei einer Portierung stets neu erstellt werden muß und ansonsten keinerlei Änderungen am eigentlichen Quellcode notwendig sind, ist der Aufwand entsprechend gering.

7.8 Die Komponenten historisierung, partner, police und vertrag

Die Portierung der Komponenten `historisierung`, `partner`, `police` und `vertrag` und der damit verbundenen „komponentenübergreifenden Vererbung“ (vgl. Kapitel 5.4) erwies sich als sehr aufwendig. Aufgrund der Erfahrungen mit den bisher portierten Komponenten konnten zunächst folgende Portierungsprobleme sofort gefunden bzw. gelöst werden:

- Aufgrund der Vererbung der Komponenten untereinander konnte die geplante Vorgehensweise (Komponente für Komponente) nicht umgesetzt werden. Stattdessen erforderten Änderungen an einer Komponente meistens entsprechende Änderungen an einer anderen Komponente, so daß diese vier Komponenten gewissermaßen „parallel“ portiert werden mußten.

- Die RemoteInterfaces aller Beans der vier genannten Komponenten mußten an die Namensgebung des PowerTier angepaßt werden (`<BeanName>` anstatt `<BeanName>Remote`).
- Das Auffinden der HomeInterfaces wurde an die `getHome(String homeName)`-Methode der neuen `vendor`-Klasse angeglichen. Die dadurch überflüssigen `getInitialContext()`- und `lookup()`-Methoden wurden deshalb aus allen Beans dieser vier Komponenten entfernt.
- Das `ObjektBean` der Komponente `vertrag` wurde als Container-Managed EntityBean entwickelt. Aus diesem Grund muß dieses Bean ebenso wie das `BedingungBean` (Kapitel 7.5) mit Hilfe des PowerTier Builders nachmodelliert werden, um ein Container-Managed EntityBean zu erhalten, welches ohne Probleme auf dem PowerTier eingesetzt werden kann.

Im Zuge der Portierung traten jedoch auch zwei neue Probleme auf, von denen sich nur eins aufgrund der theoretischen Betrachtungen aus Kapitel 6 bereits vor der Portierung erahnen ließ.

7.8.1 Komponentenübergreifende Vererbungen

Mit der Portierung der Komponenten `historisierung`, `partner`, `police` und `vertrag` ergab sich ein Problem, welches aus den theoretischen Betrachtungen zuvor nicht ersichtlich war. Dieses Problem besteht darin, daß der PowerTier nur mit Vererbungen von Beans innerhalb einer Komponente bzw. eines Package umgehen kann. Ein Beispiel für eine solche „komponenteninterne Vererbung“ wurde im Kontext der Portierung der Komponente `merkmal` in Kapitel 7.7 behandelt. Vererbungen zwischen Beans verschiedener Komponenten, wie sie in den hier betrachteten Beans umgesetzt wurden, akzeptiert der PowerTier jedoch nicht. Als grundsätzliche Ursache für dieses Problem hat sich auch hier wieder die ungenaue EJB-Spezifikation herausgestellt. Da diese zum Thema Vererbung keine Angaben macht, ist es den Anbietern der Applikationsserver freigestellt, ob und wie sie mit Vererbungen umgehen.

Um nun diese Vererbung zu umgehen und die betroffenen Komponenten trotzdem auf dem PowerTier einsetzen zu können, ergaben sich zwei mögliche Lösungsansätze:

1. Die Komponente `historisierung` wird aufgelöst und die in ihr enthaltenen Methoden und Attribute werden den drei erbenden Komponenten hinzugefügt, so daß jede einzelne dieser drei Komponenten gewissermaßen ihr eigenes Historisierungskonzept umsetzt. Der Nachteil ist sicherlich der Aufwand, der betrieben werden muß, um jede einzelne der drei erbenden Komponenten an den Inhalt der Historisierungskomponente anzupassen. Außerdem müssen zukünftige Änderungen, die bisher nur eine Komponente betrafen, in drei Komponenten eingepflegt werden. Auf der anderen

Seite sind die drei neuen Komponenten nun nicht mehr von einer anderen Komponente abhängig, so daß der Komponentengedanke noch verstärkt wird.

2. Die erbenden Komponenten `partner`, `police` und `vertrag` werden aufgelöst und komplett in die Komponente `historisierung` verlagert. Diese Variante hat jedoch den entscheidenden Nachteil, daß sie die Grundidee der komponentenbasierten Softwareentwicklung komplett ignoriert. Anstelle einzelner Komponenten mit exakt definierten Aufgaben gäbe es auf einmal lediglich eine Komponente, die zusätzlich zur Historisierung noch die Geschäftslogik von drei weiteren Komponenten zur Verfügung stellen würde. Ein Austausch einer einzelnen Komponente (z.B. der Vertragskomponente) wäre dann nicht mehr möglich. Desweiteren würde sich diese Lösung auch auf andere Eigenschaften auswirken, die von Komponenten gefordert werden (vgl. Kapitel 3.2). So würde zum Beispiel die Wartbarkeit einer solch großen Komponente deutlich reduziert.

Für die Portierung wurde aufgrund der Vorteile die erste Lösung verwendet, wobei sich keinerlei Probleme ergaben.

7.8.2 Das Container-Managed ObjektBean

Das `ObjektBean` der Komponente `vertrag` wurde in der Beispielapplikation Container-Managed realisiert. Um diesen Persistenzmechanismus beizubehalten war es ebenso wie bei der Komponente `bedingung` nötig, das Bean mit dem PowerTier Builder nachzumodellieren. Aus diesem Modell wurde anschließend Rahmenquellcode generiert, welcher an den entsprechenden Stellen um die eigentliche Geschäftslogik der Beispielapplikation erweitert wurde.

Dabei ergab sich jedoch ein Problem, welches die These bezüglich der vom PowerTier zur Verfügung gestellten Datentypen aus Kapitel 6.3.4 bestätigt. Im `ObjektBean` sind die für die Historisierung notwendigen Attribute `gueltigAb`, `gueltigBis` und `erstellt` vom Datentyp `java.sql.Timestamp`. Der PowerTier Builder bietet jedoch nur den allgemeinen Datentyp `Time` an, der dem Typen `java.util.Date` entspricht.

Nach der Quellcodegenerierung des PowerTier standen somit nur `ObjektBean`-Methoden zur Verfügung, die anstatt der bisherigen Argumente vom Typ `Timestamp` den Typ `Date` erwarteten. Da sich die Anzahl der betroffenen Methoden jedoch auf die `create`-Methode des HomeInterface beschränkte, mußte lediglich für diese Methode eine Lösung gefunden werden, die wie folgt aussieht:

1. Dem HomeInterface des `ObjektBean` wurde eine weitere `create()`-Methode hinzugefügt, die sich von der generierten `create`-Methode nur darin unterscheidet, daß die Argumente der Historisierungsattribute vom Datentyp `java.sql.Timestamp` sind.

2. In der BeanClass wurde die zugehörige `ejbCreate()`-Methode implementiert. Die Methode selbst wandelt die übergebenen `Timestamp`-Daten in Daten vom Typ `java.util.Date` um und reicht diese inklusive aller unveränderten Daten (`int id` und `String beschreibung`) an die vom PowerTier erzeugte `create`-Methode weiter.

Das auf die beschriebene Art und Weise erzeugte `ObjektBean` ließ sich problemlos in die Komponente `vertrag` integrieren und hat an einem kleinen Beispiel gezeigt, daß das bereits vermutete Datentypenproblem nicht nur im Zuge der Entwicklung einer EJB-basierten Anwendung Probleme bereiten kann, sondern auch bei Portierungen von EJB-basierten Anwendungen.

7.9 Die Komponente produkt

Die Komponente `produkt` besteht aus dem SessionBean `ProduktVerwaltungBean` und den beiden Bean-Managed EntityBeans `ProduktBean` und `ElementarproduktBean`. Probleme bei der Portierung dieser Komponente ergaben sich lediglich bei dem bereits bekannten Namensproblem der RemoteInterfaces, der Anbindung an die Datenbank und die Verwendung der `vendor`-Klasse. Die Lösungen dieser Probleme wurden bereits mehrfach besprochen und werden daher nicht weiter erläutert.

7.10 Die Komponente praemienangleichung

Die Portierung der Komponente `praemienangleichung`, welche aus dem SessionBean `PraemienangleichungBean` besteht, ergab keine großen Probleme. Dieses Bean stellt lediglich die Methode `Praemienangleichung` zur Verfügung und mußte nur in Hinsicht auf drei bereits bekannte Probleme angepaßt werden:

1. Die Bezeichnung der RemoteInterfaces mußte auch in dieser Komponente von `<BeanName>Remote` hinzu `<BeanName>` geändert werden.

2. Alle Aufrufe der Methode `getInitialContext()` bzw. alle `lookup()`-Methoden zum Auffinden der HomeInterfaces wurden durch die Methode `getHome(String homeName)` der neuen `vendor`-Klasse (Kapitel 7.4) ersetzt.

3. Da die Vererbung der Komponente `historisierung` aufgelöst und in die erbenden Beans verlagert wurde (Kapitel 7.8) galt es, Referenzen zu dieser Komponente aufzulösen und durch Referenzen zum an dieser Stelle benötigten Bean zu ersetzen (hier z.B. durch Referenzen zum `VertragBean`).

Ansonsten wurden auch für dieses Bean ein DeploymentDescriptor und ein JAR-Archiv erzeugt. Das korrekte Verhalten dieser portierten Komponente konnte jedoch nicht getestet werden, da sie in der Beispielapplikation noch keine Verwendung findet, das heißt, daß die zur Verfügung gestellten Methoden noch an keiner Stelle verwendet werden.

7.11 Die Komponente tarif

Ebenso wie die Komponente `praemienangleichung` besteht auch die Komponente `tarif` lediglich aus einem Bean, dem SessionBean `TarifBean`, und stellt ebenfalls nur eine Geschäftsmethode zur Verfügung (`berechneVertragPraemie(VertragObjekt DO)`). Im Vergleich zum `PraemienangleichungBean` mußten auch hier drei zum Teil bekannte Probleme gelöst werden, wobei das Erzeugen des DeploymentDescriptors zunächst ohne weitere Probleme durchgeführt werden konnte:

1. Die Anpassung der RemoteInterface-Namen an die Namensgebung des PowerTier.

2. Die Anpassung der `getConnection()`-Methode zum Aufbau einer Datenbankverbindung analog zu Kapitel 7.3.

3. Ein bis dahin unbekanntes Problem, welches auch in Kapitel 6 noch nicht abzusehen war, ist erstmalig im `TarifBean` aufgetreten und konnte in einigen wenigen weiteren Beans erneut beobachtet werden. Dieses Problem äußerte sich darin, daß der PowerTier nach der Generierung des DeploymentDescriptors die folgende Fehlermeldung ausgab:

```
PowerTier (TM) Domain Builder v6.00
  Compiling Java source files ...
  Compiled Java source files.
  Updated .des files.
  Creating E:\prototyp\prototyp-tarif.jar ...
  Generating Java Impl files ...
  Compiling Java source files ...
  Compiled Java source files.
  Creating stubs and skeletons ...
  Created stubs and skeletons.
  Compiling Java source files ...
E:\prototyp\tarif\TarifHomeImpl.java:49:
Exception javax.ejb.CreateException is never thrown in the
body of the corresponding try statement.
            catch(CreateException exc)
```

```
                ^
1 error
java.lang.IllegalStateException:
   Failed to compile Java source files.
     at com.persistence.tools.dpi.Compiler.compileSource(...)
     at com.persistence.tools.dpi.JarBuilder.createPTJar(...)
     at com.persistence.tools.dpi.JarBuilder.createAll(...)
     at com.persistence.tools.dpi.DomainBuilder.execute(...)
     at com.persistence.tools.dpi.DomainBuilder.main(...)
Domain Builder status:
   com.persistence.tools.dpi.BuildException:
  Failed to compile Java source files
```

Dieser Fehler beruht jedoch nicht auf einer unterschiedlichen Interpretation der EJB-Spezifikation durch die Applikationsserver, sondern in der Genauigkeit, wie diese Vorgaben überprüft werden. Schon in der Version 1.0 werden die Anforderungen an die einzelnen Beanbestandteile beschrieben. Dabei findet sich zum Beispiel zum Thema *HomeInterface* folgende Anforderung:

> „*The throws clause of every create method must include the java.rmi.RemoteException and the javax.ejb.CreateException. It may also include additional application-level exceptions.*“
>
> ([HM98, S.52])

Jede `create`-Methode muß also sowohl `RemoteExceptions` als auch `CreateExceptions` abfangen. Diese Forderung gilt ebenfalls für die entsprechenden `ejbCreate`-Methoden:

> „*There are zero 1 or more ejbCreate(...) methods, whose signatures match the signatures of the create(...) methods of the enterprise Beans home interface.*“
>
> ([HM98, S.64])

In der zu portierenden Applikation finden sich für die `create`-Methode des Tarif-Bean folgende Quellcodezeilen:

```
HomeInterface : public TarifRemote create()
                    throws RemoteException, CreateException;
BeanClass     : public void ejbCreate(){}
```

Hier fällt auf, daß das Abfangen der `RemoteException` und `CreateException` in der BeanClass nicht implementiert wurde. Genau dieses akzeptiert der PowerTier-Server jedoch nicht. Um den oben aufgeführten Fehler zu beseitigen, müssen diese Excpetions nicht nur im HomeInterface, sondern auch in der BeanClass abgefangen werden:

> „*You must pay special attention to writing create methods. At a minimum, the signature for ejbCreate() must be as follows:*
> ***`public void ejbCreate() throws java.rmi.RemoteException, java.ejb.CreateException`***“
> ([Per01, Programming Guide, S.204])

Der für den PowerTier korrekte Quellcode sieht demnach also wie folgt aus:

```
HomeInterface : public TarifRemote create()
                    throws RemoteException, CreateException;
BeanClass     : public void ejbCreate()
                    throws RemoteException, CreateException {}
```

Es handelt sich bei diesem Problem also eher um die Genauigkeit, mit der die Applikationsserver den Quellcode hinsichtlich der EJB-Spezifikation beim Kompillieren abgleichen. Was der WebLogic im `TarifBean` akzeptiert, obwohl es nicht der Spezifikation entspricht, wird beim PowerTier erkannt und als Fehler zurückgegeben.

Nachdem dann noch der entsprechende Import eingefügt wurde (`import javax.ejb.CreateException;`), ließ sich anschließend auch das JAR-Archiv der Komponente `tarif` problemlos mittels des `ps-makeejb -all`-Befehls erzeugen.

7.12 Die Komponente verteiler

Die Komponente `verteiler` enthält nur das SessionBean `VerteilerBean`, welches die Anfragen des Client entgegennimmt und diese an die jeweils für die Anfrage zuständigen SessionBeans weiterleitet. Aufgrund dieser Vermittleraufgabe beschränken sich alle vom `VerteilerBean` zur Verfügung gestellten Methoden auf folgende Funktion:

1. Auffinden der Home- und RemoteInterfaces der SessionBeans, an die die Anfragen des Client gerichtet sind.

2. Aufruf der vom Client gewünschten Methoden über die zurückgelieferten Interfaces.

Somit ergab sich bei der Portierung des `VerteilerBeans` lediglich das bereits beschriebene Namensproblem der RemoteInterfaces. Auch hier bestand die Lösung also in der Umbenennung aller RemoteInterfaces (`<BeanName>Remote.java` in `<BeanName>.java`).

7.13 Portierung der SessionBeans

Die Beschreibung der Portierung der bisher genannten Komponenten bezog sich in erster Linie auf die enthaltenen EntityBeans. Die SessionBeans `VertragVerwaltungBean`, `MerkmalVerwaltungBean`, `PartnerVerwaltungBean`, `PoliceVerwaltungBean` und `ProduktVerwaltungBean` wurden nicht weiter erwähnt, da sich die in diesem Zusammenhang beobachteten Probleme auf die bereits bekannten Bereiche

- Namensanpassung an den PowerTier
- Anpassung an die neue Hilfsklasse `vendor` (hinsichtlich des InitialContext)

beschränkten. Neue, noch nicht behandelte Portierungsprobleme waren nicht zu verzeichnen.

7.14 Portierung des Client

Um die Beispielanwendung zu portieren, waren auch auf Client-Seite zwei größere Anpassungen notwendig. Dabei beschränkten sich diese Anpassungen jedoch auf die Anbindung des Client an den Server:

Die Kommunikation mit dem Server über das `VerteilerBean` konnte aufgrund der Namensänderungen an den Beans nicht übernommen werden. Da der Client seine Anfragen jedoch ausschließlich über dieses SessionBean an den Server weitergibt, waren lediglich die Stellen zu bearbeiten, an denen das RemoteInterface `Verteiler` benötigt wird. Dazu mußten in der Client-Datei `MainFrame.java` alle Referenzen, die sich auf `verteiler.VerteilerRemote` bezogen, durch Referenzen auf `verteiler.Verteiler` ersetzt werden.

Der JNDI InitialContext stellt eine Art Startpunkt für jeglichen JNDI-Aufruf dar. Über ihn kann der Client in einer verteilten Umgebung zum Beispiel die vom Server zur Verfügung gestellten HomeInterfaces wiederfinden. Da die JNDI-Implementierung davon abhängig ist, wie der Anbieter des Applikationsservers diese intern umgesetzt hat, unterscheiden sich auch die entsprechenden Abschnitte des Quellcodes von Server zu Server. So wurde der InitialContext der Beispielapplikation mittels der Methode `getInitialContext()` realisiert. Durch die Anpassung der Klasse

`vendor` (s. Kapitel 7.4) fällt diese Methode jedoch in der portierten Anwendung weg. Statt dessen ermittelt der Client zu Beginn einmalig einen `InitialContext`. Dies geschieht über folgende Befehle:

```
String servername = ''server'';
String args[] = new String[2];
args[0]=''ORBDefaultInitRef'';
args[1]=''iiopboot://localhost:901'';

Services svcs = Services.init(args, null);
jndiContext = svcs.getInitialContext(null);
```

Die wesentlichen Unterschiede zum InitialContext des WebLogic bestehen in der Verwendung der PowerTier-spezifischen Klasse `Services` und den übergebenen Parametern (vergleichbar mit den Properties des WebLogic). Dabei unterscheidet sich die hier verwendete Server-URL im verwendeten Protokolltyp (IIOP, Internet Inter-ORB Protocol, anstelle des WebLogic-spezifischen T3-Protokolls) und im anzusprechenden Port des Servers (901 anstelle von 7001).

Solche Änderungen am Client sind also nicht zu vermeiden, da die EJB-Spezifikation den Serveranbietern keine Vorschriften bezüglich der JNDI-Implementierung macht. Ein weiterer Beleg für dieses Problem findet sich in der Literatur zum Beispiel bei Monson-Haefel ([MH99b, S.111]), der zusätzlich noch die `getInititalContext()`-Methode für den GemStone/J-Applikationsserver angibt:

```
public static Context getInitialContext() throws Exception {
   Properties p = new Properties();
   p.put(com.gemstone.naming.Defaults.NAME_SERVICE_HOST,
         ''localhost'');
   String port =
     System.getProperty(''com.gemstone.naming.NameServicePort'',
                        ''10200'');
   p.put(com.gemstone.naming.Defaults.NAME_SERVICE_PORT,
         port);
   p.put(Context.INITIAL_CONTEXT_FACTORY,
         ''com.gemstone.naming.GsCtxFactory'');
   return new InitialContext(p);
}
```

Der beste Lösungsansatz um solche Portierungsprobleme so gering wie möglich zu halten, liegt in der Verlagerung der Serverspezifischen Bestandteile in Hilfsklassen wie sie zum Beispiel die `vendor`-Klasse darstellt. Die

eigentlichen Komponenten sind dann „serverneutral“ und leichter zu portieren.

7.15 Zusammenfassung

Nachdem in diesem Kapitel die erste Portierung einer EJB-basierten Anwendung durchgeführt wurde, sollen nun noch einmal die Probleme und Lösungen zusammenfassend dargestellt werden.

Die Bezeichnung der RemoteInterfaces mußte an die Namensgebung des PowerTier angepaßt werden. Dazu waren sowohl die Dateien, als auch die Referenzen zu den RemoteInterfaces im Quellcode selbst umzubenennen (`<BeanName>Remote` in `<BeanName>`)

Die Anbindung an die Datenbank erforderte das Anpassen einiger Parameter in den DeploymentDescriptoren. Die wichtigste Änderung betraf die URL, über die eine Verbindung zur Datenbank aufgebaut wird. Dieser WebLogic-spezifische Wert wurde durch eine JDBC-Url der Oracle-Datenbank ersetzt. Als Folge dessen mußte die Methode `getConnection()` einer jeden Bean um die Zeile für das Laden des Oracle-Treibers erweitert werden.

Die Hilfsklasse vendor wurde komplett überarbeitet, indem die Methode `getInitialContext()` entfernt und durch die neue Methode `getHome(String homeName)` ersetzt wurde. Der Vorteil liegt darin, daß nicht in jeder Bean das Ermitteln des InitialContext und das Aufsuchen der benötigten HomeInterfaces separat an den neuen Server angepaßt werden muß.

Die komponentenübergreifende Vererbungen der Beispielapplikation ließ sich nicht auf den PowerTier übertragen. Die Ursache für dieses Problem ist in den fehlenden Informationen der EJB-Spezifikation bezüglich des Themas *Vererbung* zu suchen. Eine Lösung bestand darin die Vererbung aufzulösen, indem die Superklasse in jeder erbenden Klasse einzeln implementiert wurde. Die Begriffe *Klasse* und *Bean* sind an dieser Stelle austauschbar.

Die Komponenteninternen Vererbungen der Komponente `merkmal` bereiteten keine Probleme. Auch die Modellierung dieser Vererbung mit dem PowerTier Builder verlief problemlos.

Die Container-Managed EntityBeans konnten nicht ohne den PowerTier Builder portiert werden. Eine einfache Anpassung des Quellcodes dieser Beans an die EJB-Spezifikation 1.1 schlug fehl. Die Ursache hierfür

ist wie bereits bei der Namensgebung der RemoteInterfaces in der EJB-Spezifikation selbst zu suchen. Diese spezifiziert zwar den Aufbau einer solchen Bean, überläßt es jedoch den Anbietern der Applikationsserver, den CMP-Mechanismus umzusetzen.

Die Vorgaben der EJB-Spezifikation werden von den Servern unterschiedlich stark umgesetzt. Laut EJB-Spezifikation muß zum Beispiel die Signatur einer `create()`-Methode mit der Signatur ihrer `ejbCreate()`-Methode übereinstimmen. Dazu zählt auch das Abfangen eventueller Exceptions. Der PowerTier reagiert hier sehr empfindlich und akzeptiert es im Gegensatz zum WebLogic nicht, wenn die Exceptions zwar im HomeInterface, nicht jedoch in der BeanClass abgefangen werden. Im Zuge der Portierung waren also teilweise die Signaturen der einzelnen Methoden abzugleichen.

Kapitel 8

Portierung auf WebSphere

Dieses Kapitel behandelt die zweite Portierung der Beispielapplikation. Als Zielserver wird der in Kapitel 6.4 beschriebene *WebSphere* der Firma IBM verwendet.

8.1 Grundlagen

Bereits zu Beginn der zweiten Portierung ist abzusehen, daß einige der Probleme aus Kapitel 7 erneut auftreten werden, so daß die entsprechenden Lösungen an dieser Stelle ebenfalls angewendet werden können. Betroffen sind dabei die zwei folgenden Bereiche.

8.1.1 Anbindung an die Datenbank

Für die notwendigen Änderungen zur Anbindung an die Datenbank wird an dieser Stelle auf Kapitel 7.3 verwiesen. Die dort beschriebenen Probleme treten auch bei der Portierung auf den WebSphere auf, so daß die jeweiligen Lösungen in diesem Kapitel unverändert übernommen werden können.

8.1.2 Verwendung des JNDI-Context

Im Rahmen der Portierung der Beispielapplikation auf den PowerTier ist es hinsichtlich des `InitialContext` zu diversen Portierungsproblemen gekommen ist. Wie bereits beschrieben, wird der sogenannte `InitialContext` benötigt, um die HomeInterfaces der einzelnen Beans zu finden. In diesem Zusammenhang werden einige Informationen benötigt ohne die keine Verbindung zum Server aufgebaut werden kann. Kapitel 7.4 beschreibt diese Informationen und Zusammenhänge bezüglich des `InitialContext`. Für die Portierung auf den WebSphere konnten die dort beschriebenen Lösungen ohne große Probleme übernommen werden:

- Die Methode `getInitialContext()` wurde aus allen Beans entfernt und durch den Aufruf der Methode `findHome(String name)` der Hilfsklasse `vendor` ersetzt. Die Funktionalität dieser Methode entspricht der in Kapitel 7.4 beschriebenen Methode `getHome(String homeName)`. Der einzige Unterschied besteht darin, daß die Methode für den PowerTier die PowerTier-spezifische Schnittstelle (`Services`) verwendet um den `InitialContext` zu ermitteln. Für den WebSphere konnte an dieser Stelle auf eine serverunabhängige Java-Lösung zurückgegriffen werden.

- Die Werte der Variablen `Context.INITIAL_CONTEXT_FACTORY` und `Context.PROVIDER_URL` wurden ebenfalls an den WebSphere angepaßt:

  ```
  Context.INITIAL_CONTEXT_FACTORY =
     com.ibm.ejs.ns.jndi.CNInitialContextFactory und
  Context.PROVIDER_URL = iiop://localhost:900
  ```

8.2 Geplantes Vorgehen

Der Ablauf der Portierung auf den WebSphere entspricht dem Ablauf der Portierung auf den PowerTier. Es soll also Komponente für Komponente portiert werden. Dabei kann jedoch nicht ausgeschlossen werden, daß dieser Ablauf (ebenso wie bei der ersten Portierung) aufgrund von Abhängigkeiten der einzelnen Beans untereinander oder anderen Problemen zeitweise Änderungen unterworfen ist.

Die Portierung der Beans selbst orientiert sich an den Vorgaben der Dokumentation des WebSphere (vgl. Kapitel 6.4.3). Dabei wird im weiteren Verlauf auf die beschriebene Java-Lösung zurückgegriffen. Die Umsetzung dieser Lösung ist zunächst unabhängig davon, um welche Art von Bean es sich handelt. Bei den Container-Managed EntityBeans kann es jedoch zu den in Kapitel 6.4.5 beschriebenen Problemen kommen. Die Verwendung des in diesem Zusammenhang empfohlenen Tools *Visual Age für Java* soll nach Möglichkeit vermieden werden. Die Vorgehensweise der Java-Lösung sieht wie folgt aus:

1. Die `.java`-Dateien einer Komponente werden mittels des JDK-Programms `javac` kompiliert.

2. Die in Schritt 1 erzeugten `.class`-Dateien werden mittels des JDK-Programms `jar` in einem JAR-Archiv zusammengefaßt.

3. Das in Schritt 2 erstellte JAR-Archiv wird mit dem Jetace-Tool eingelesen, um zu jeder Bean des Archivs einen DeploymentDescriptor zu erstellen.

4. Sowohl die Beans als auch die DeploymentDescriptoren werden mit dem Jetace-Tool in einem JAR-Archiv zusammengefaßt.

Nachdem alle Komponenten portiert worden sind, können die JAR-Archive im Server installiert und gestartet werden. Auf diesen Vorgang wird jedoch wie bereits bei der Portierung auf den PowerTier nur eingegangen, wenn es zu Problemen kommt.

8.3 Portierung der SessionBeans

Die SessionBeans der Beispielapplikation konnten bis auf die bekannten Probleme der Datenbankanbindung und des `InitialContext` (Kapitel 8.1) problemlos portiert werden.

8.4 Portierung der Bean-Managed EntityBeans

Die Portierung der Bean-Managed EntityBeans verlief weitestgehend problemlos. Das größte Problem ergab sich bei der Portierung der Komponente `historisierung` und den erbenden Komponenten `partner`, `police` und `vertrag`. Ebenso wie bei der Portierung auf den PowerTier konnte auch an dieser Stelle die „komponentenübergreifende Vererbung“ nicht auf den WebSphere übertragen werden. Die Lösung dieses Problems bestand in der Auflösung der Vererbungsbeziehung, wie sie bereits beim PowerTier durchgeführt wurde (Kapitel 7.8.1). Die Komponente `historisierung` wurde entfernt und jeweils in den erbenden Komponenten implementiert. Danach ergaben sich abgesehen von den Problemen der Datenbankanbindung und des `InitialContext` keine weiteren Portierungsprobleme.

8.5 Portierung der Container-Managed EntityBeans

Zu Beginn der Portierung der Container-Managed EntityBeans wurden die benötigten Finder-Interfaces erstellt (Kapitel 6.4.5). Diese Interfaces werden vom WebSphere benötigt, da sie die Informationen enthalten, welche Werte der Datenbank eine bestimmte Find-Methode zurückliefern soll. Für die `findByPrimaryKey()`-Methode definiert die EJB-Spezifikation, wie der Server Applikationsserver diese intern umzusetzen hat. Die vom Entwickler benötigten Find-Methoden sind allerdings nicht bekannt, so daß sie über die zusätzlichen Interfaces definiert werden müssen.

Die Beispielapplikation selbst enthält lediglich sechs Container-Managed EntityBeans. Für die Bean `BedingungBean` der Komponente `bedingung` ist beispielhaft das notwendige Interface `BedingungBeanFinderHelper` aufgeführt:

```
package bedingung;

public interface BedingungBeanFinderHelper {
      String findeAlleBedingungenQueryString =
          ''select * from BEDINGUNG'';
}
```

Der String `findeAlleBedingungenQueryString` repräsentiert in diesem Zusammenhang das vom Server anzuwendende SQL-Statement dar. Dem Server wird auf diese Art und Weise für jede Find-Methode mitgeteilt, welche Anfrage an die Datenbank zu richten ist, um die gewünschten Werte zu erhalten. Da nur das `BedingungBean` eine derartige Find-Methode enthält (`findeAlleBedingungen`), sind die Interfaces der restlichen fünf Beans leer, müssen allerdings trotzdem erstellt werden.

Bis zum abschließenden Test der portierten Applikation traten keine weiteren Probleme auf. Es bestand allerdings zu keiner Zeit die Möglichkeit, das gewünschte Datenbankmapping vorzugeben. Das Jetace-Tool bietet in dieser Hinsicht keine Unterstützung und die Administrationskonsole des WebSphere bietet lediglich an, nicht existierende Tabellen bei Bedarf zu erstellen. Dies ist jedoch mit fixen Tabellennamen seitens des WebSphere verbunden. Im Verlauf der Testphase führten Zugriffe auf die Container-Managed EntityBeans dann auch stets zu Fehlern, die mit einem Programmabbruch verbunden waren. Als Ursache hierfür konnten schließlich die Datenbankzugriffe dieser EntityBeans ermittelt werden. Aufgrund dieser Ergebnisse und der Tatsache, daß das Problem mit den WebSphere-Tools nicht gelöst werden kann, wurde nun doch das von IBM geforderte Tool *Visual Age für Java* eingesetzt.

8.5.1 Visual Age für Java

Die Entwicklungsumgebung *Visual Age für Java* bietet zahlreiche Möglichkeiten für die Arbeit mit EJBs. Zur Lösung des beschriebenen Problems ist allerdings nur der Bereich „Datenbankmapping von Container-Managed EntityBeans“ von Interesse. Dazu wurde zunächst der Quellcode der Beispielapplikation eingelesen. Anschließend sieht die Dokumentation des WebSphere das Einlesen des Datenbankschemas vor, um anhand dieses Schemas die Abbildung der Beanattribute auf die Datenbank vorzunehmen. Abbildung 8.1 zeigt beispielhaft die Eingabemaske, über die die Attribute des `ObjektBean` auf die entsprechenden Spalten der Tabelle `OBJEKT` abgebildet werden.

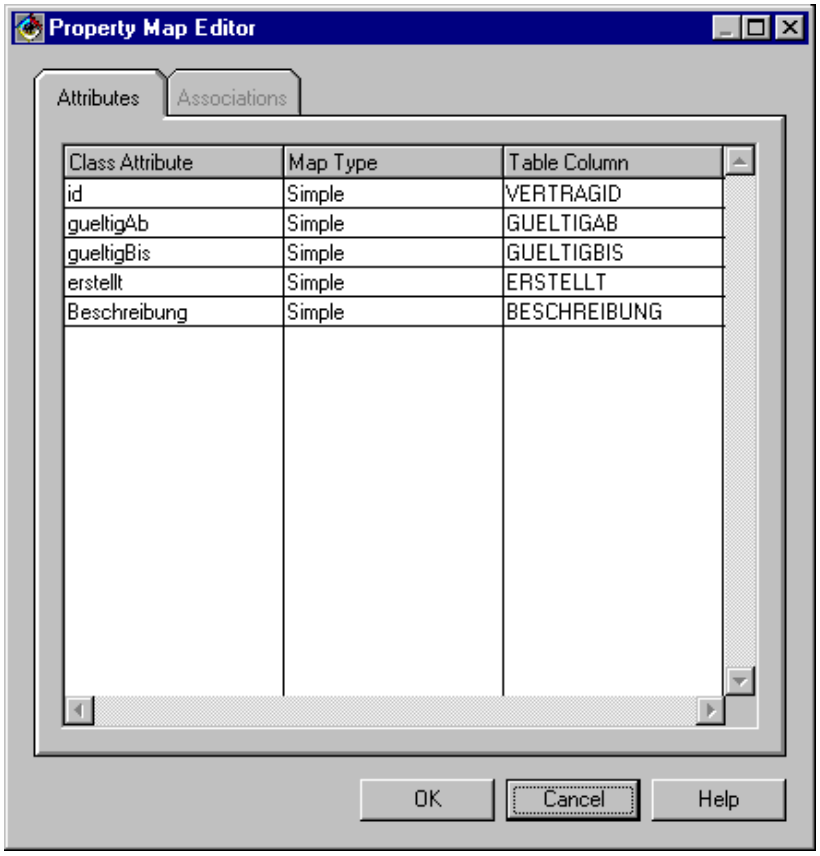

Abbildung 8.1: Datenbankmapping mit Visual Age für Java

Bereits das Einlesen des Datenbankschemas konnte nicht erfolgreich durchgeführt werden. In diesem Zusammenhang wurde die Struktur einiger Tabellen nicht korrekt erkannt, so daß es zu Folgefehlern kam:

1. Der PrimaryKey der betroffenen Tabellen wurde nicht erkannt.
2. Die Abbildung der Attribute auf die Datenbank schlug fehl, da die erkannten Datentypen der Datenbank nicht den Datentypen der Beanattribute entsprachen.

Als Reaktion auf die geschilderten Probleme wurden unter *Visual Age* folgende Lösungen vorgeschlagen:

1. Änderungen an der Datenbankstruktur. Diese Lösung ist nicht akzeptabel. Bereits der Einsatz der Entwicklungsumgebung *Visual Age für Java* war nicht vorgesehen. Wird nun auch noch die Datenbank bearbeitet oder neu aufgesetzt, so kommt dieser Aufwand deutlich in den Bereich einer Neuentwicklung. Die in Kapitel 4.4 aufgeführte Definition stuft in solch einem Fall die entsprechende Anwendung als nicht portabel ein. Diese Aussage gilt an dieser Stelle jedoch aus folgenden Gründen nicht uneingeschränkt:

   (a) Die SessionBeans und die Bean-Managed EntityBeans konnten ohne Probleme portiert werden. Das entspricht ca. 72

   (b) Mit Hilfe von *Visual Age* können Test-Clients generiert werden. Diese testen die grundlegenden Funktionen einer Bean (z.B. die Methode `create()` zum Anlegen einer Bean in der Datenbank). Dabei werden die Attribute in einer vorgegebenen Reihenfolge in Tabellen mit fixen Namen abgelegt. Der Test des Container-Managed EntityBeans `ObjektBean` verlief erfolgreich, so sich die Portierungsprobleme nur auf das Datenbankmapping beschränken.

2. Vorgabe eines anderen Datenbankmappings. Auch diese Lösung kommt nicht in Frage, da das Datenbankmapping durch die Beispielapplikation fest vorgegeben ist.

Zusammenfassend läßt sich festhalten, daß die Portierung der Beispielapplikation auf den WebSphere nicht wie gewünscht durchgeführt werden konnte. Die SessionBeans und Bean-Managed EntityBeans konnten portiert und erfolgreich getestet werden. Der Test der gesamten Anwendung ließ sich aufgrund des fehlenden Datenbankmappings der Container-Managed EntityBeans nicht durchführen. Diese konnten zwar portiert aber nicht in die Anwendung integriert werden. Die Ursache ist dabei eindeutig auf Seite der Entwicklungsumgebung *Visual Age für Java* zu suchen. Nur mit Hilfe dieser Entwicklungsumgebung ist es möglich, Container-Managed EntityBeans auf eine Datenbank abzubilden, deren Schema nicht dem WebSphere-Standard entspricht. Wenn allerdings im Zusammenhang mit diesem Tool zusätzlich noch Datenbankprobleme entstehen, so ist der Portierungsaufwand der Container-Managed EntityBeans bereits im Bereich der Neuentwicklung einzuordnen. Dieses Verhalten kann laut Kapitel 4.4 nicht als portabel gewertet werden.

8.6 Portierung des Client

Die Portierung des Client konnte analog zur Portierung auf den PowerTier durchgeführt werden (Kapitel 7.14). Lediglich die Methode `getInitialContext()` mußte an die in Kapitel 8.1.2 aufgeführten Werte angepaßt werden.

8.7 Zusammenfassung

Auf den ersten Blick erscheint die Portierung auf den WebSphere weniger problematisch als die Portierung auf den PowerTier. Dieser Eindruck ergibt sich durch die Tatsache, daß bestimmte Portierungsprobleme wie zum Beispiel die Anbindung an die Datenbank oder die Auflösung der „komponentenübergreifenden Vererbung“ bereits bekannt waren. Die entsprechenden Lösungen konnten in

diesen Fällen meist unverändert übernommen werden. Insgesamt gesehen verlief die Portierung ebenso problematisch wie die Portierung auf den PowerTier. Die Probleme ergaben sich lediglich durch andere Umstände. Es fällt auf, daß auch bei der zweiten Portierung die Container-Managed EntityBeans für die größten Probleme sorgten. Auch hier bestand eine Lösung nur in der Verwendung eines weiteren Tools. Nachfolgend sind die Probleme und Lösungen dieser Portierung kurz zusammengefaßt:

Die Anbindung an die Datenbank brachte auch im Rahmen dieser Portierung einige Probleme mit sich. Diese erforderten die Anpassung einiger Parameter in den DeploymentDescriptoren, sowie die Erweiterung der Methode `getConnection()` einer jeden Bean um die Zeile für das Laden des Oracle-Treibers. Im DeploymentDescriptor wurde der WebLogic-spezifische Wert für die JDBC-Url durch einen serverunabhängigen Wert der Oracle-Datenbank ersetzt. Diese Lösung konnte von der Portierung auf den PowerTier übernommen werden, da dieses Problem bereits von dort bekannt war.

Die Hilfsklasse vendor wurde komplett überarbeitet, indem die Methode `getInitialContext()` entfernt und durch die neue Methode `findHome(String name)` ersetzt wurde. Der Vorteil liegt darin, daß nicht in jeder Bean das Ermitteln des InitialContext und das Aufsuchen der benötigten HomeInterfaces separat an den neuen Server angepaßt werden muß. Auch in diesem Zusammenhang waren Problem und Lösung durch die Portierung auf den PowerTier bekannt.

Die komponentenübergreifenden Vererbungen der Beispielapplikation konnten im Verlauf der Portierung nicht übernommen werden. Analog zur Portierung auf den PowerTier wurde die Vererbung aufgelöst, indem die vererbende Superklasse jeweils in den erbenden Klassen implementiert wurde.

Die komponenteninternen Vererbungen der Komponente `merkmal` ergaben keine Probleme.

Die Container-Managed EntityBeans konnten nicht problemlos portiert werden. Der Hauptgrund hierfür ist im Mapping der Beanattribute auf die Datenbank zu suchen. Nur durch die Verwendung des Tools *Visual Age für Java* ist es dem Anwender möglich, eine bestehende Datenbankstruktur zu übernehmen und anzugeben, wie die Beans auf die Datenbank abgebildet werden sollen. Ohne dieses Tool steht dem Anwender nur die Administrationskonsole zur Verfügung. Über diese kann der Anwender jedoch keinerlei Einfluß auf das Datenbankmapping ausüben. In diesem Fall bildet der Server die Attribute intern auf Tabellen ab, die nach einem vom WebSphere vorgegebenen Schema benannt sind. Dieses Schema

entsprach jedoch nicht dem bereits existierenden Schema der verwendeten Oracle-Datenbank. Hinzukommt, daß auch für die einzelnen Tabellenspalten vom WebSphere eine bestimmte Reihenfolge festgelegt wird, die ebenfalls nicht der existierenden Reihenfolge entspricht. Eine erfolgreiche Portierung setzt also das Tool *Visual Age für Java* voraus, welches jedoch die Datenbank nicht problemlos einlesen konnte. Aufgrund fehlender Alternativen zur Lösung des beschriebenen Mapping-Problems konnten die Container-Managed EntityBeans also portiert aber nicht eingesetzt werden.

Kapitel 9

Theorie und Praxis - Ein Vergleich

In den vorhergehenden Kapiteln wurden sowohl theoretische, als auch praktische Aspekte der Portierung einer EJB-basierten Anwendung betrachtet. Die Ergebnisse, die sich im Verlauf der Diplomarbeit ergeben haben, werden in diesem Kapitel zusammengetragen, um schließlich die Rahmenbedingungen abzuleiten, die eine Aussage über die Portabilität von EJB-basierten Anwendungen zulassen.

Zunächst werden die in Kapitel 6 aufgestellten Thesen hinsichtlich eventueller Portierungsprobleme mit den praktischen Ergebnissen der beiden Portierungen in den Kapiteln 7 und 8 verglichen. Das Ergebnis dieses Vergleichs spiegelt die Ursachen der verschiedenen Portierungsprobleme wieder und bildet somit in Verbindung mit den entsprechenden Lösungsansätzen die Grundlage für die zu erarbeitenden Rahmenbedingungen.

9.1 Theoretische und Praktische Portierungsprobleme

In Kapitel 6 wurden aufgrund der Beschreibung der verwendeten Applikationsserver Thesen aufgestellt, die sich mit eventuellen Problemen bei der Portierung in den Kapiteln 7 und 8 beschäftigen. In den folgenden Abschnitten wird nun untersucht:

1. Haben sich diese Thesen bei den Portierungen bestätigt?

2. Sind eventuell Portierungsprobleme aufgetreten, die in Kapitel 6 noch nicht abzusehen waren?

3. Welche Ursachen sind den einzelnen Portierungsproblemen zuzuordnen?

9.1.1 Portierungsprobleme PowerTier

Nicht alle der in Kapitel 6.3.9 aufgestellten Thesen konnten im Zuge der Portierung der Beispielapplikation auf den PowerTier bestätigt werden:

These 1 , die Portierung sei vermutlich unabhängig von den Tools des PowerTier, konnte nicht bestätigt werden. Für die Portierung der Container-Managed EntityBeans wurde der *PowerTier Builder* benötigt. Der Einsatz dieses Tools führt jedoch dazu, daß die Applikation nach der Portierung nicht mehr auf andere Server portiert werden kann (vgl. Kapitel 6.3.4). Es entsteht somit eine Abhängigkeit, die ursprünglich vermieden werden sollte. Die Ursachen liegen in diesem Fall bei der EJB-Spezifikation. Diese überläßt die Realisierung des *Container-Managed Persistence* den Anbietern der Applikationsserver und gibt diesen dazu keine konkreten Vorgaben. Die daraus resultierenden Unterschiede zwischen den Servern führen zu den beschriebenen Problemen.

Die Verwendung des PowerTier Builder selbst verlief weitestgehend problemlos. Um das `ObjektBean` der Komponente `vertrag` modellieren zu können, mußte für die Historisierungsattribute der Datentyp `java.util.Date` verwendet werden. Der eigentlich benötigte Datentyp `java.sql.Timestamp` der Beispielapplikation stand nicht zur Verfügung. Dem generierten Quellcode wurde aus diesem Grund eine Wrapperklasse hinzugefügt (vgl. Kapitel 7.8.2).

These 2 , der Wechsel der EJB-Spezifikation von Version 1.0 zu Version 1.1 führt zu Portierungsproblemen, konnte durch die Portierung bestätigt werden. Aufgrund der zahlreichen Unterschiede zwischen den Spezifikationen waren Änderungen am Quellcode nicht zu vermeiden. Durch den Einsatz des PowerTier Builder (s.o.) war es allerdings unerheblich, ob die Container-Managed EntityBeans an die neue Spezifikation angepaßt wurden.

These 3 , die in der Beispielapplikation verwendeten Vererbungen könnten unter Umständen zu Problemen führen, konnte ebenfalls bestätigt werden. In diesem Zusammenhang muß allerdings zwischen der „komponenteninternen“ und der „komponentenübergreifenden“ Vererbung unterschieden werden.

Die „komponenteninterne Vererbung“ bereitete keine Probleme. Die Container-Managed EntityBeans der betroffenen Komponente `merkmal` mußten aufgrund der mit dem PowerTier Builder nachmodelliert werden. Da sich jedoch Vererbungen dieser Art im PowerTier Builder darstellen lassen, verlief die Übernahme dieser Vererbung

Die „komponentenübergreifende Vererbung“, welche die Komponenten `historisierung`, `partner`, `police` und `vertrag` betrifft, mußte aufgelöst

werden. Ein derartiger Vererbungsmechanismus wird vom PowerTier nicht unterstützt wird.

Auch für dieses Problem ist die EJB-Spezifikation verantwortlich. Das Thema „Vererbung“ wird in der EJB-Spezifikation nicht spezifiziert. Somit bleibt es den Anbietern der Applikationsserver überlassen, ob und wie Vererbungen realisiert werden können.

Insgesamt ergaben sich zwei Probleme, die im Rahmen der theoretischen Betrachtung des PowerTier nicht abzusehen waren:

1. Die Namensgebung der RemoteInterfaces konnte nicht übernommen werden. Der PowerTier akzeptiert nur RemoteInterfaces, die mit `<BeanName>` bezeichnet werden. Für die Arbeit mit dem WebLogic wurde jedoch die Bezeichnung `<BeanName>Remote` verwendet. Diese Änderungen betrafen sowohl die Quellcodedateien als auch die Referenzen im Quellcode selbst. Die Ursache dieses Problem besteht auch in diesem Fall in der ungenauen EJB-Spezifikation, die keine Bezeichnung der Beanbestandteile keine vorgibt.

2. Beim Komplilieren des Quellcode wurden die unterschiedlichen Signaturen der Methoden `create` und `ejbCreate` vom PowerTier nicht akzeptiert (vgl. Kapitel 7.11), da die EJB-Spezifikation identische Signaturen fordert. Vermieden werden kann ein solches Problem nur, wenn sowohl der Applikationsserver (z.B. beim Kompilieren) als auch der Entwickler (beim Entwickeln) sich streng an die Vorgaben der Spezifikation halten.

9.1.2 Portierungsprobleme WebSphere

Auch für den WebSphere konnten nicht alle Thesen aus Kapitel 6.4.8 bestätigt werden:

These 1 , der Einsatz des Tools *Visual Age für Java* sei vermeidbar, hat sich für diese Portierung als falsch erwiesen. Nur für den Fall, daß die Datenbankstruktur den internen Vorgaben des WebSphere entspricht (Kapitel 6.4.5), wird dieses Tool nicht benötigt. Da jedoch in der existierenden Datenbank eine abweichende Struktur vorliegt, mußte *Visual Age* zwangsläufig verwendet werden. Andernfalls wäre ein Mapping der Container-Managed EntityBeans auf die Datenbank nicht möglich gewesen. Der Grund für diese Abhängigkeit zu einem Tool besteht also wie schon beim PowerTier in der ungenügenden Spezifikation des *Container-Managed Persistence*.

These 2 , die Vererbungen innerhalb der Beispielapplikation könnten zu Problemen führen, hat sich bestätigt. Analog zur Portierung auf den PowerTier mußte die „komponentenübergreifende Vererbung“ aufgelöst werden,

während die „komponenteninterne Vererbung“ unverändert übernommen werden konnte.

These 3 , für die Find-Methoden der Container-Managed EntityBeans müßten noch zusätzliche Interfaces implementiert werden, hat sich bestätigt. Dieses Problem ist ein weiteres Resultat des ungenügend spezifizierten Themas *Container-Managed Persistence*.

Im Gegensatz zur Portierung auf den PowerTier ergaben sich nur Probleme, die bereits aufgrund der theoretischen Betrachtung des Servers abzusehen waren.

9.2 Ermittlung der Rahmenbedingungen

Aufgrund der Ergebnisse aus Kapitel 9.1 lassen sich nun Rahmenbedingungen ableiten, die eine Aussage über die Portabilität von entsprechenden Anwendungen zulassen. Der Ursprung der einzelnen Rahmenbedingungen findet sich jeweils in den Ursachen der beobachteten Portierungsprobleme. Dabei ist zu beachten, daß der Einfluß der einzelnen Bedingungen auf die Portabilität unterschiedlich stark ausgeprägt ist. In einigen Fällen ist es durchaus möglich, daß die Verletzung einer Rahmenbedingung dazu führt, daß die Anwendung nicht mehr zu portieren ist. Die Verletzung einer anderen Rahmenbedingung wiederum führt dazu, daß die Portabilität nur minimal beeinflußt wird. Bei der Beschreibung der einzelnen Bedingungen wird auf diesen Aspekt jeweils individuell eingegangen.

9.2.1 Die zugrundeliegende EJB-Spezifikation

Vergleicht man die beiden durchgeführten Portierungen, so läßt sich leicht feststellen, daß ein grundlegendes Kriterium für die Portabilität einer EJB-basierten Anwendung die zugrundeliegende Spezifikationsversion ist. Die Gründe dafür sind:

1. In den bisher veröffentlichten Spezifikationen werden den einzelnen Rollen noch zuviele Freiräume durch ungenügend spezifizierte Aufgaben gelassen (z.B. *Container-Managed Persistence, Vererbung oder Namensgebung*). Besonders die Versionen 1.0 und 1.1 sind davon betroffen. Ob mit der EJB-Spezifikation 2.0 eine Verbesserung zu erwarten ist wird in Kapitel 10.2 diskutiert.

2. Aufgrund der zahlreichen Änderungen, die bisher mit der Veröffentlichung einer neuen Spezifikation einhergegangen sind, ist die Portabilität zwischen Applikationsservern, die auf verschiedenen Spezifikationen aufsetzen, stark eingeschränkt. Änderungen am Quellcode sind in diesem Zusammenhang nicht zu vermeiden (z.B. *InitialContext* und *Rückgabewerte von Container-Managed EntityBeans*).

Insgesamt läßt sich diese Abhängigkeit der Portabilität zur verwendeten Spezifikation wie folgt beschreiben:

> „*Die zugrundeliegende EJB-Spezifikation einer entsprechenden Anwendung erlaubt Aussagen über die Portabilität dieser Anwendung. Bei Portierungen zwischen Applikationsservern, die auf verschiedenen Versionen der EJB-Spezifikation aufsetzen, sind Portierungsprobleme und die Bearbeitung des Quellcodes nicht zu vermeiden. Des weiteren ist mit einer Einschränkung der Portabilität genau dann zu rechnen, wenn in der Anwendung Bereiche angesprochen werden, die von der EJB-Spezifikation nicht oder nur unklar spezifiziert sind (z.B. Vererbung, CMP). Im Kontext der Entwicklung EJB-basierter Anwendung sollten diese Bereiche möglichst ausgelassen und durch alternative Lösungen realisiert werden.*“

9.2.2 Serverspezifische APIs

Bei der Entwicklung der Beispielapplikation wurden von Malte Hülder bewußt keine serverspezifischen APIs eingesetzt, um eine Abhängigkeit zum WebLogic zu vermeiden. Für die Portierung war dies von Vorteil, da Abhängigkeiten jeder Art im Verlauf einer Portierung wieder aufgelöst werden müssen. Im Gegensatz dazu wurde der Quellcode der Beispielapplikation durch die Verwendung des PowerTier Builder um zahlreiche PowerTier-spezifische Quellcodefragmente erweitert. Diese realisieren in den Container-Managed EntityBeans die gewünschte Funktionalität bezüglich des *Container-Managed Persistence*. Dazu werden PowerTier-spezifische APIs benötigt, die in Kapitel 6.3.8 beschrieben wurden. Gleichzeitig sind die betroffenen Beans durch die Verwendung dieser PowerTier-Spezifika zukünftig nicht mehr portabel (Kapitel 6.3.4).

Dieses Beispiel zeigt, daß sich für die Verwendung von Serverspezifika folgende Rahmenbedingung ableiten läßt:

> „*Um eine EJB-basierte Anwendung portabel zu entwickeln sollte auf die Verwendung Serverspezifischer APIs verzichtet werden. Andernfalls müssen die sich ergebenden Abhängigkeiten im Verlauf einer Portierung wieder aufgelöst werden. Je mehr Serverspezifische APIs verwendet werden, desto mehr nimmt die Portabilität der Anwendung ab. Dies kann unter Umständen dazu führen, daß eine Portierung ganz ausgeschlossen werden muß.*“

9.2.3 Die Bündelung spezieller Funktionalitäten

Im Verlauf der Diplomarbeit wurde bereits Brown erwähnt, der in seinem Kapitel „Techniques for Enhancing Portability“ ([Bro77, S.119 ff.]) die Aufsplittung der

Anwendung in spezifische und unspezifische Module bzw. Komponenten vorschlägt. Dieser Vorschlag wurde in der Beispielapplikation durch die Hilfsklasse `vendor` ansatzweise umgesetzt. Es hat sich jedoch bei den Portierungen gezeigt, daß eine noch konsequentere Umsetzung der Technik von Brown in der Beispielapplikation den Portierungsaufwand erheblich vermindert hätte.

In der Beispielapplikation von Malte Hülder greift lediglich der Client auf die Klasse `vendor` zu, um den InitialContext zu erhalten. Die einzelnen Beans hingegen nutzen diese Möglichkeit nicht und implementieren die Methoden zum Ermitteln des InitialContext und zum Auffinden der HomeInterfaces stets selbst. Dies erfordert im Zuge einer Portierung die Quellcodebearbeitung einer jeden Bean, da die entsprechenden Methoden an den neuen Server angepaßt werden müssen. Für die Portierung wurde die Klasse `vendor` so umgeschrieben, daß in jeder Bean ein benötigtes HomeInterface über eine einzige Methode der Klasse `vendor` ermittelt werden kann. Der dafür notwendige InitialContext wird ebenfalls in dieser Methode ermittelt. Um auch noch die Klasse `vendor` möglichst portabel zu gestalten, wurden in einem weiteren Schritt die notwendigen serverspezifischen Parameter (z.B. `Context.INITIAL_CONTEXT_FACTORY`) in einer Textdatei abgelegt, die von den Methoden der Klasse `vendor` eingelesen werden.

Die Vorteile dieses Vorgehens sind:

1. Die serverspezifische Methoden zum Ermitteln des InitialContext und zum Auffinden eines HomeInterface sind in einer Methode einer Klasse gebündelt. Eventuelle Änderungen (z.B. Anpassungen an einen neuen Server) können somit zentral an einer Stelle vorgenommen werden und betreffen nicht jede einzelne Bean. Somit wird also nicht nur die Portabilität, sondern auch die Wartbarkeit der Anwendung deutlich verbessert.

2. Die Klasse `vendor` stellt keine Komponente dar. Sollten Änderungen an der Klasse notwendig werden, so muß nicht jedesmal ein neues JAR-Archiv erstellt und im Server installiert werden. Das Kompilieren der geänderten Klasse ist völlig ausreichend.

3. Durch die Verlagerung der serverspezifischen Werte in einfache Textdateien wird die Portabilität der Anwendung noch erhöht. Findet ein Serverwechsel statt, so reicht es unter Umständen aus lediglich die Textdatei anzupassen. Ein Kompilieren der Klasse `vendor` wird dann nur noch notwendig, falls der Quellcode geändert werden muß.

Ein Nachteil dieses Vorgehens besteht sicherlich in der Tatsache, daß eine Abhängigkeit der Komponenten zur Klasse `vendor` entsteht. Wägt man jedoch die Vor- und Nachteile gegeneinander ab, so ist deutlich zu erkennen, daß die beschriebenen Vorteile diese Abhängigkeit durchaus rechtfertigen.

Um die Portabilität der Anwendung noch weiter zu erhöhen wäre es auch möglich gewesen, die Anbindung an die Datenbank ebenfalls in der

Klasse `vendor` zu realisieren. In der Beispielapplikation baut abgesehen von den Container-Managed EntityBeans jede Bean, die mit der Datenbank kommuniziert, eine eigene JDBC-Verbindung auf. Die in diesem Zusammenhang notwendige Methode `getConnection()` (Kapitel 7.3) hätte ebenfalls in die Klasse `vendor` ausgelagert werden können. Somit wäre die Anpassung der Datenbankparameter (User, Passwort und JDBC-URL) ebenfalls zentral in einer Klasse der Anwendung möglich gewesen.

Zusammenfassend läßt sich also folgende Rahmenbedingung formulieren:

> „*Um eine EJB-basierte Anwendung portabel zu entwickeln sollte eine Aufsplittung in spezifische und unspezifische Teile vorgenommen werden. Der Ausdruck spezifisch ist an dieser Stelle vorwiegend als serverspezifisch zu interpretieren. Es können allerdings auch andere spezifische Bereiche wie zum Beispiel die Datenbankanbindung (s.o.) abgegrenzt werden.*“

Einen positiven Seiteneffekt übt diese Rahmenbedingung auch hinsichtlich der Wartbarkeit der Anwendung aus.

9.2.4 Der Persistenzmechanismus

Die Portierungen der Beispielapplikation haben gezeigt, daß der verwendete Persistenzmechanismus einen großen Einfluß auf die Portabilität einer EJB-basierten Anwendung ausübt. Im Verlauf der Diplomarbeit konnten die Bean-Managed EntityBeans ohne Probleme auf den neuen Server portiert werden. Die dazu notwendigen Änderungen am Quellcode betrafen höchstens die Anbindung an die Datenbank. Im Gegensatz dazu ergaben sich bei der Portierung der Container-Managed EntityBeans weitaus größere Probleme.

Da die EJB-Spezifikation den Anbietern der Applikationsserver nicht vorschreibt wie der Persistenzmechanismus *Container-Managed Persistence* umzusetzen ist, gibt es an dieser Stelle große Unterschiede zwischen den einzelnen Servern. Diese Unterschiede betreffen hauptsächlich die Art und Weise wie die entsprechenden Beans zu implementieren sind. Die Container-Managed EntityBeans konnten nur mit Hilfe der Tools *PowerTier Builder* (PowerTier) und *Visual Age für Java* (WebSphere) portiert werden. Die Verwendung solcher Tools ist jedoch äußerst kritisch zu beurteilen:

1. Es entsteht eine Abhängigkeit zu den verwendeten Tools, da die Portierung ohne diese nicht durchzuführen ist.

2. Werden diese Tools bereits bei der Entwicklung der Anwendung verwendet, so kann dies dazu führen, daß eine spätere Portierung nicht möglich ist (vgl. Einschränkung des PowerTier in Kapitel 6.3.4).

3. Eine Bearbeitung des Quellcodes kann nicht vermieden werden. Teilweise ist es zwingend notwendig, die Beans erneut zu implementieren (z.B. über den PowerTier Builder).

Die Portierung einer EJB-basierten Anwendung mit Container-Managed EntityBeans ist somit ohne zusätzliche Tools und ohne Bearbeitung des Quellcodes zur Zeit nicht möglich. Die grundlegende Idee nur über den DeploymentDescriptor Einfluß auf die Portierung zu nehmen wird dadurch unterbunden. Des weiteren ist die Bearbeitung des Quellcodes grundsätzlich nicht gewünscht, da zum Beispiel der Käufer einer EJB-Komponente den Quellcode nicht kennen, sondern lediglich nutzen soll (Kapitel 1.1).

Die den Persistenzmechanismus betreffende Rahmenbedingung lautet wie folgt:

> „*Bei der Entwicklung einer EJB-basierten Anwendung ist darauf zu achten, daß die EntityBeans als Bean-Managed EntityBeans realisiert werden. Nur so kann zur Zeit die Portabilität der Anwendung sichergestellt werden. Die Verwendung von Container-Managed EntityBeans erfordert für die Entwicklung oder Portierung meist zusätzliche Tools. Da die Abhängigkeit zu Tools jeder Art nicht gewünscht ist, empfiehlt es sich zu warten bis die EJB-Spezifikation den Bereich Container-Managed Persistence hinreichend spezifiziert hat.*“

9.2.5 Der Einfluß der DeploymentDescriptoren

Die DeploymentDescriptoren sind für EJB-basierte Anwendungen von großer Bedeutung. Sie beeinflussen nicht nur maßgeblich das Verhalten der Beans zur Laufzeit sondern enthalten teilweise auch Umgebungsvariablen (Properties). Diese können von den Beans über einen `Context` abgerufen und verwendet werden.

Die Beispielapplikation kann in diesem Zusammenhang als Beispiel für eine optimale Ausnutzung der DeploymentDescriptoren herangezogen werden. Diese beeinflußen nicht nur das Verhalten der einzelnen Beans sondern enthalten gleichzeitig Umgebungsvariablen mit deren Hilfe innerhalb der Anwendung zum Beispiel eine Datenbankverbindung aufgebaut wird (User, Passwort, etc.). Eventuelle Änderungen dieser Umgebungsvariablen (z.B. aufgrund eines neuen Passworts) betreffen somit lediglich die DeploymentDescriptoren und nicht den Quellcode. Dies wiederum reduziert den Portierungsaufwand nicht unerheblich, da die DeploymentDescriptoren im Gegensatz zum Quellcode für jeden Server angepaßt bzw. neu erstellt werden müssen. Daraus läßt sich folgende Rahmenbedingung ableiten:

> *„Die optimale Verwendung der DeploymentDescriptoren einer EJB-basierten Anwendung kann den Aufwand für eine Portierung deutlich vermindern. Der Begriff „optimal“ ist in diesem Zusammenhang so zu verstehen, daß Parameter, die zur Laufzeit von den Beans benötigt werden, möglichst im DeploymentDescriptor eingetragen und zur Laufzeit über den Bean-Kontext abgerufen werden. Dies bezieht sich vor allem auf spezifische Parameter aller Art (Serverspezifisch, Datenbankspezifisch, etc.). Somit beschränken sich notwendige Änderungen der Parameter auf den DeploymentDescriptor und eine Bearbeitung des Quellcodes wird vermieden.“*

9.2.6 Der Einsatz zusätzlicher Tools

Die beiden durchgeführten Portierungen haben gezeigt, daß die Verwendung zusätzlicher Tools in vielen Fällen zu Portierungsproblemen führt. Dabei kann es sich zum Beispiel um Tools handeln, die Einfluß auf den Quellcode nehmen oder die für das Datenbankmapping zuständig sind. Oftmals sind diese im Leistungsumfang des Applikationsservers enthalten oder werden von Drittanbietern hinzugekauft. Im Kontext der Diplomarbeit finden sich für beide Fälle Beispiele:

- Bei der Portierung auf den PowerTier wurde das mitgelieferte Tool *PowerTier Builder* verwendet, ohne welches die Portierung der Container-Managed EntityBeans nicht durchgeführt werden konnte. Aus der Verwendung des PowerTier Builder resultiert jedoch ein weiteres Problem, welches sich erst im Verlauf einer weiteren Portierung bemerkbar macht. Die Dokumentation des PowerTier schließt eine mögliche Portierung von Container-Managed EntityBeans aus, die auf Basis des PowerTier Builder erstellt wurden (Kapitel 6.3.4).

- Bei der Portierung auf den WebSphere muß der Entwickler die Entwicklungsumgebung *Visual Age für Java Enterprise Edition* verwenden um die Container-Managed EntityBeans auf eine existierende Datenbankstruktur abbilden zu können. Dieses Werkzeug wird zwar auch von der Firma IBM entwickelt, ist jedoch nicht im Leistungsumfang des Applikationsservers enthalten, so daß es erst noch hinzugekauft werden muß. Eine mögliche Alternative könnte in der Konvertierung der Beans in Bean-Managed EntityBeans bestehen, das heißt der Entwickler implementiert die Datenbankkommunikation selbst. Dies entspricht jedoch einer Neuimplementierung und kann somit nicht empfohlen werden.

Insgesamt läßt sich also sagen, daß sich die Verwendung von bestimmten Tools negativ auf die Portabilität von EJB-basierten Anwendungen auswirkt. Dies gilt besonders für Tools, die Einfluß auf den Quellcode ausüben.

Aus diesen Erkenntnissen läßt sich folgende Rahmenbedingung ableiten:

> „*Der Einsatz zusätzlicher Tools vermindert unter bestimmten Voraussetzungen die Portabilität EJB-basierter Anwendungen. Beispiele hierfür sind:*
>
> 1. *Der Quellcode wird derart beeinflußt, daß eine Abhängigkeit zum Tool entsteht.*
> 2. *Aufgrund bestimmter Umstände wird das Tool zwangsläufig benötigt (z.B. für das Datenbankmapping der Container-Managed EntityBeans)*
>
> *Um negative Einflüsse wie die beschriebenen zu vermeiden dürfen entsprechende Tools nicht verwendet werden. Statt dessen sollten alternative Lösungen umgesetzt werden.* “

9.2.7 Die Umsetzung von Vorgaben

Es gibt für die Arbeit mit EJBs zwei grundlegende Dokumente an denen sich die Entwicklung entsprechender Anwendungen orientiert:

1. Die EJB-Spezifikation. Diese spezifiziert die EJB-Technologie unter anderem anhand der in Kapitel 4.2.1 beschriebenen Rollen. Jede dieser Rollen übernimmt bestimmte Aufgaben und hat sich dabei grundsätzlich an die Vorgaben der EJB-Spezifikation zu halten.

2. *Java Coding Conventions* ([Con]). Sun stellt dieses Dokument zur Verfügung um für die Entwicklung von Java-basierten Anwendungen einen einheitlichen Programmierstil vorzugeben. So wird zum Beispiel spezifiziert, daß Klassennamen groß zu schreiben sind oder gibt an, wie Klammern im Quellcode am Besten zu setzen sind. Einige Teile dieser Conventions sind für die Entwicklung verpflichtend, andere wiederum sind lediglich als Empfehlung zu verstehen.

Der Einfluß der EJB-Spezifikation auf die Portabilität von EJB-basierten Anwendungen ist wesentlich größer als der Einfluß der *Java Coding Conventions*. Die Portierung der Beispielapplikation hat hinsichtlich dieser beiden Dokumente folgende Ergebnisse gebracht:

- Die EJB-Spezifikation läßt den einzelnen Rollen bei der Realisierung zahlreiche Freiräume. Dies kann zu Portierungsproblemen führen. Ein Beispiel hierfür sind die fehlenden Angaben der EJB-Spezifikation zum Thema Vererbung. Nur der WebLogic konnte mit der implementierten „komponentenübergreifenden Vererbung“ umgehen. Von den beiden

anderen Servern hingegen konnte dieser Vererbungsmechanismus nicht umgesetzt werden. Solange die EJB-Spezifikation also keine konkreten Aussagen zum Thema Vererbung macht sollte dieser Vererbungsmechanismus nach Möglichkeit vermieden werden.

Stellen an denen keine Unklarheiten vorliegen sollten jedoch der Vorgabe entsprechend umgesetzt werden. Im `TarifBean` der Beispielapplikation kam es zu Problemen mit dem PowerTier, da die Signatur der `create()`-Methode nicht mit der Signatur der `ejbCreate()`-Methode übereinstimmte. Dieses Problem hätte vermieden werden können, wenn die Vorgaben der EJB-Spezifikation exakt umgesetzt worden wären.

- Auch wenn der Einfluß der *Java Coding Conventions* auf die Portabilität als wesentlich geringer zu bewerten ist als der Einfluß der EJB-Spezifikation, so empfiehlt es sich doch, diese Coding Conventions umzusetzen. Im Rahmen der Portierungen fiel zum Beispiel auf, daß der Name einige Attribute der Beispielapplikation mit einem Großbuchstaben beginnen. Wie das nachfolgende Beispiel zeigt, sind in diesem Zusammenhang weitere Portierungsprobleme mit dem PowerTier denkbar:

  Die Quellcodezeile

  ```
  PartnerRemote Partner = null;
  ```

  hätte aufgrund des Problems mit der Namensgebung des PowerTier wie folgt umgeschrieben werden müssen:

  ```
  Partner Partner = null;
  ```

  Das Attribut `Partner` entspräche dann ebenfalls der Klasse `Partner`, so daß Fehler an dieser Stelle nicht zu vermeiden gewesen wären. Eine vergleichbare Situation hat sich allerdings nicht ergeben. Um solche Probleme jedoch von vorne herein auszuschließen, sollte der Entwickler sich streng an die Java Coding Conventions halten.

Eine weitere Rahmenbedingung bezüglich der Portabilität von EJB-basierten Anwendungen kann also wie folgt formuliert werden:

> „*Um eine EJB-basierte Anwendung möglichst portabel zu entwickeln ist es unbedingt erforderlich die Vorgaben der EJB-Spezifikation zu beachten. Ist dies aufgrund fehlender oder unzureichend beschriebener Vorgaben nicht möglich so ist nach alternativen Lösungen zu suchen. Neben der EJB-Spezifikation sollte auch auf eine korrekte Umsetzung der Java Coding Conventions geachtet werden.*“

Kapitel 10

Fazit und Ausblick

10.1 Fazit

Die Analyse der betrachteten Applikationsserver und die anschließende Portierung der Beispielapplikation haben gezeigt, daß die „Portierung von EJB-basierten Anwendungen“ grundsätzlich möglich ist. Dabei kann den Aussagen von Monson-Haefel ([MH99b, S. 2]) und Thomas (Kapitel 1 bzw. [Tho98, S. 5]), EJB-basierte Anwendungen seien so portabel, daß sie unabhängig vom verwendeten Applikationsserver eingesetzt werden können, so nicht zugestimmt werden. Die Diplomarbeit hat in diesem Zusammenhang gezeigt, daß die Portabilität EJB-basierter Anwendungen von bestimmten Rahmenbedingungen abhängig ist, die in der Literatur bisher nicht zu finden sind. Diese Rahmenbedingungen wurden im Verlauf der Arbeit hergeleitet und in Kapitel 9 aufgeführt.

Insgesamt lassen sich die Rahmenbedingungen auf zwei große Bereiche anwenden:

1. Sie lassen eine Aussage über die Portabilität bereits existierender Anwendungen zu. Im Falle einer geplanten Portierung können die Rahmenbedingungen wie eine Art Check-Liste verwendet werden an der festgemacht werden kann, in welchen Bereichen es zu Portierungsproblemen kommen kann bzw. wird. Anhand dieser Analyse ist es anschließend möglich zu beurteilen, mit welchem Aufwand bei der Portierung voraussichtlich zu rechnen ist. Sollte sich dieser Aufwand dem Kosten- und Zeitaufwand einer Neuentwicklung nähern, so muß unter Umständen an Alternativen zur geplanten Portierung gedacht werden. Diese Alternativen könnten zum Beispiel die erwähnte Neuentwicklung oder auch der Kauf einer EJB-basierten Komponente mit entsprechender Funktionalität sein.

2. Sie können bei der Entwicklung von EJB-basierten Anwendungen hinzugezogen werden. Werden die Rahmenbedingungen vom Entwickler

beachtet und umgesetzt, so kann die Portabilität einer Anwendung deutlich verbessert werden. Dabei kann es allerdings vorkommen, daß die Vernachlässigung einer bestimmten Rahmenbedingung sich weniger kritisch auf die Portabilität der Anwendung auswirkt als die Vernachlässigung einer anderen Rahmenbedingung. Wichtig ist auch, daß nicht nur eine, sondern alle Rahmenbedingungen bei der Entwicklung beachtet werden.

Die Umsetzung dieser Rahmenbedingungen schränkt den Entwickler allerdings in zahlreichen Bereichen ein. So fordert zum Beispiel die Rahmenbedingung aus Kapitel 9.2.4, daß die Verwendung von Bean-Managed EntityBeans die Portabilität einer Anwendung im Gegensatz zur Verwendung von Container-Managed EntityBeans deutlich erhöht. Gleichzeitig nimmt der Entwickler jedoch auch einige negative Begleiterscheinungen in Kauf:

- Die Kommunikation mit der Datenbank muß „von Hand“ programmiert werden. Dieses Verfahren ist weitaus fehleranfälliger und zeitaufwendiger als der Einsatz von Container-Managed EntityBeans und lenkt vom eigentlichen Hintergrund EJB-basierter Anwendungen ab: Der Konzentration auf die Entwicklung der Geschäftslogik. Je nach Komplexität der benötigten Datenbankabfragen kann es also dazu kommen, daß sich der Fokus der Entwicklung nicht mehr nur auf die Umsetzung der Geschäftslogik in den Beans richtet, sondern auch auf die Kommunikation mit der Datenbank.

- Container-Managed EntityBeans sind meist performanter, da die Kommunikation mit der Datenbank von den Anbietern der Applikationsserver optimal abgestimmt ist.

Ein weiteres Beispiel für eine Einschränkung stellen die zahlreichen Tools dar. So liefert zum Beispiel Persistence den PowerTier Builder mit aus, mit dessen Hilfe Container-Managed EntityBeans modelliert werden können, aus denen anschließend Rahmenquellcode erzeugt wird. Aber nicht nur Anbieter von Applikationsservern stellen Tools dieser Art zur Verfügung. Auch über Drittanbieter können ähnliche Hilfsmittel erworben werden. Für den Entwickler stellt sich somit stets die Frage, ob er auf diese Tools zurückgreift oder nicht. Um die Anwendung möglichst portabel zu entwickeln, dürfen Hilfsmittel wie zum Beispiel der PowerTier Builder nicht verwendet werden. Laut Persistence ist der generierte Quellcode nicht portabel und kann somit auch nicht auf anderen Servern eingesetzt werden (Kapitel 6.3.4). In diesem Zusammenhang stellt sich jedoch die Frage ob Sinn es macht Geld für einen Server auszugeben, dessen Leistungspotential nicht ausgeschöpft wird? Gibt es unter Umständen günstigere Server, die im Endeffekt die selbe Funktionalität (nur ohne entsprechende Tools) bieten? Diese Fragen gelten nicht nur für den PowerTier, sondern für jeden Server, der Tools zur Verfügung stellt, auf deren Verwendung zugunsten der Portabilität verzichtet werden muß.

Der Entwickler verfügt also über zwei Möglichkeiten:

1. Die Anwendung wird mit Hilfe der Rahmenbedingungen portabel entwickelt. Dafür wird auf zahlreiche Möglichkeiten des Servers verzichtet und vermutlich längere Entwicklungszeiten, höhere Entwicklungskosten und eine schlechte Performanz in Kauf genommen. Eine absolut problemlose Portabilität kann zur Zeit allerdings nicht zugesichert werden, da bei der Portierung der Beispielapplikation auf andere Server durchaus neue individuelle Probleme auftreten können. Erst wenn die EJB-Spezifikation ausgereifter ist und die verschiedenen Rollen nicht mehr so unvollständig bzw. fehlerhaft beschrieben werden, kann von einer Verbesserung des Themas „Portabilität“ ausgegangen werden. Bis dahin wird die Umsetzung der Rahmenbedingungen empfohlen.

2. Der Fokus liegt auf der optimalen Zusammenarbeit der Anwendung mit dem Server. Dafür ist die Anwendung nur schwer auf andere Server zu portieren (im schlimmsten Fall gar nicht). Bei Anwendungen von denen bekannt ist, daß sie nicht auf anderen Servern eingesetzt werden sollen, kann die Portabilität durchaus vernachlässigt werden. Problematisch wird es allerdings für Entwickler, die ihre EJBs zum Verkauf anbieten (Kapitel 1). Einerseits müssen diese EJBs absolut portabel sein, da der Entwickler nicht für jeden verfügbaren Server eine eigene Version entwickeln kann. Andererseits sollen die EJBs zum Beispiel auch performant und unabhängig von jeglicher Datenbank sein. Zur Zeit schließen sich diese Punkte jedoch gegenseitig aus, so daß ein großangelegter Handel mit EJBs wohl noch etwas auf sich warten lassen dürfte.

10.2 Ausblick

Im Rahmen dieser Diplomarbeit wurden zahlreiche Probleme der Portierung EJB-basierter Anwendungen behandelt. Bei einem Großteil dieser Probleme konnte als Ursache die EJB-Spezifikation ermittelt werden. Die weiteren Probleme ergaben sich dann als Folgefehler. Die EJB-Spezifikation spezifiziert die EJB-Technologie in den Versionen 1.0 und 1.1 teilweise sehr ungenau bzw. lückenhaft. Mit der im Oktober des letzten Jahres als „Proposed Final Draft“ veröffentlichten Version 2.0 sollen jedoch viele Probleme behoben werden.

Die meisten Änderungen betreffen das in dieser Arbeit oft kritisierte *Container-Managed Persistence*. Das beschriebene Problem des Datenbankmapping soll zukünftig die neue Rolle des *Persistence-Manager-Provider* übernehmen. Dieser soll die Unabhängigkeit zur Datenbank erweitern, indem er das Mapping zu verschiedenen Datenbanksystemen (O/R, OO, etc.) übernimmt und dabei aus Sicht der Beans und des Containers stets gleich angesprochen wird. Des weiteren kann im DeploymentDescriptor „*eine abstrakte Schema-*

Beschreibung angegeben werden, die beschreibt, wie das objekt-relationale Mapping durchgeführt werden soll. Diese enthält die persistenten Felder der Bean und der von ihr abhängigen Klassen sowie von Relationen. ([Haw00])“.

Ein Zugewinn an Portabilität ist auch hinsichtlich der Find-Methoden zu erwarten. In der EJB-Spezifikation 2.0 wird die *EJB Query Language* (EJB QL) definiert. Mit Hilfe dieser sollen auch die Find-Methoden für Container-Managed EntityBeans einheitlich spezifiziert werden, so daß proprietäre Ansätze wie die Finder-Interfaces des WebSphere entfallen.

Zur Zeit gibt es nur einige wenige Applikationsserver, die bereits auf der neuen Spezifikation aufsetzen. Dazu zählt unter anderem der WebLogic 6.1. Die meisten Anbieter sind erst noch dabei, eine vollständige Unterstützung der EJB-Spezifikation 1.1 aufzubauen. Dazu zählen auch die Nachfolger des WebSphere (Version 4.0) und des PowerTier (Version 6.5).

Anhang A

Quellcode des PowerTier Builder

A.1 MyClassHome.java

```
// ============================================================================
// File: MyClassHome.java
// Generated by Persistence(TM) Object Builder from
// Persistence Software, Inc.
// ============================================================================
// The Persistence Object Builder software is the property of Persistence
// Software, Inc. Any rights to use are granted under the license agreement.
// Copyright 1998, all rights reserved.
// ============================================================================

package prototyp;

import java.math.BigDecimal;
import java.util.Date;
import java.util.Enumeration;
import java.rmi.RemoteException;
import javax.ejb.*;
import com.persistence.container.*;

//BEGIN ----------------- PS(Prototyp,MyClass,home_imports)
// Custom imports.
//END   ----------------- PS(Prototyp,MyClass,home_imports)


public interface MyClassHome
    extends EntityHome
    //BEGIN ----------------- PS(Prototyp,MyClass,home_parents)
    // Custom parent interfaces of MyClassHome <preceded by a comma>.
```

```
    //END   ----------------- PS(Prototyp,MyClass,home_parents)

{
    //BEGIN ----------------- PS(Prototyp,MyClass,home_methods)
    // Custom MyClassHome methods.
    //END   ----------------- PS(Prototyp,MyClass,home_methods)

    public abstract MyClass create()
        throws RemoteException, CreateException;
    public abstract MyClass createWithOid(byte[] id)
        throws RemoteException, CreateException;
    public abstract MyClass create(MyClassCreationState info)
        throws RemoteException, CreateException;
    public abstract Enumeration create(MyClassCreationState info[])
        throws RemoteException, CreateException;
    public abstract void setState(Enumeration states)
        throws RemoteException, FinderException;
    public abstract MyClass findByPrimaryKey(MyClassKey pkey)
        throws RemoteException, FinderException;
    public abstract MyClass findByPrimaryKey(MyClassKey pkey, int findSource)
        throws RemoteException, FinderException;
    public abstract MyClassState findStateByPrimaryKey(MyClassKey pkey, int findSourc
       throws RemoteException, FinderException;
}
```

A.2 MyClass.java

```
// ============================================================================
// File: MyClass.java
// Generated by the Persistence(TM) Object Builder from
// Persistence Software, Inc.
// ============================================================================
// The Persistence Object Builder software is the property of Persistence
// Software, Inc. Any rights to use are granted under the license agreement.
// Copyright 1998, all rights reserved.
// ============================================================================

package prototyp;

import java.math.BigDecimal;
import java.util.Date;
```

```
import java.util.Enumeration;
import java.rmi.RemoteException;
import javax.ejb.*;
import com.persistence.container.*;

// ============================================================================
// Stored procedure parameters
// ============================================================================
//
//     Attribute values will be supplied to the k_insert stored procedure
//     in the following order:
//         id (primary key)
//
//     Attribute values will be supplied to the k_read stored procedure
//     in the following order:
//         id (primary key)
//
//     Attribute values will be supplied to the k_update stored procedure
//     in the following order:
//         id (primary key)
//
//     Attribute values will be supplied to the k_remove stored procedure
//     in the following order:
//         id (primary key)
//
// ============================================================================

//BEGIN ----------------- PS(Prototyp,MyClass,imports)
// Custom imports.
//END   ----------------- PS(Prototyp,MyClass,imports)


public interface MyClass
    extends EntityObject
    //BEGIN ----------------- PS(Prototyp,MyClass,parents)
    // Custom parent interfaces of MyClass <preceded by a comma>.
    //END   ----------------- PS(Prototyp,MyClass,parents)

{
    //BEGIN ----------------- PS(Prototyp,MyClass,methods)
    // Custom MyClass methods.
    //END   ----------------- PS(Prototyp,MyClass,methods)
```

```
    public abstract byte[] getId() throws RemoteException;

    public abstract boolean getIdNull() throws RemoteException;
    public abstract void setState(EntityState state) throws RemoteException;
}
```

A.3 MyClassBean.java

```
// ====================================================================
// File: MyClassBean.java
// Generated by the Persistence(TM) Object Builder from
// Persistence Software, Inc.
// ====================================================================
// The Persistence Object Builder software is the property of Persistence
// Software, Inc. Any rights to use are granted under the license agreement.
// Copyright 1998, all rights reserved.
// ====================================================================

package prototyp;

import java.math.BigDecimal;
import java.util.Date;
import java.util.Enumeration;
import java.rmi.RemoteException;
import javax.ejb.*;
import com.persistence.container.*;
import com.persistence.container.internal.*;

//BEGIN ----------------- PS(Prototyp,MyClass,bean_imports)
// Custom imports.
//END ----------------- PS(Prototyp,MyClass,bean_imports)


public class MyClassBean
    extends EntityBeanImpl
    //BEGIN ----------------- PS(Prototyp,MyClass,bean_parents)
    // Custom parent interfaces of MyClassBean <preceded by 'implements'>.
    //END ----------------- PS(Prototyp,MyClass,bean_parents)

{
    //BEGIN ----------------- PS(Prototyp,MyClass,bean_methods)
    // Custom MyClassBean methods.
```

```
//END ----------------- PS(Prototyp,MyClass,bean_methods)


public static EntityBean getInstance() throws RemoteException
{
    return new MyClassBean();
}

public static EntityBean getInstance(NativePObject pobj) throws RemoteEx
{
    return new MyClassBean(pobj);
}

public void ejbCreate()
    throws RemoteException, CreateException
{
    //BEGIN ----------------- PS(Prototyp,MyClass,bean_create)
    // Custom create code.
    //END ----------------- PS(Prototyp,MyClass,bean_create)

}

public void ejbPostCreate()
    throws RemoteException, CreateException
{
    //BEGIN ----------------- PS(Prototyp,MyClass,bean_postcreate)
    // Custom post-create code.
    //END ----------------- PS(Prototyp,MyClass,bean_postcreate)

}

public void ejbCreate(MyClassCreationState info)
    throws RemoteException, CreateException
{
    //BEGIN ----------------- PS(Prototyp,MyClass,bean_state_create)
    // Custom state create code.
    //END   ----------------- PS(Prototyp,MyClass,bean_state_create)

}

public void ejbPostCreate(MyClassCreationState info)
    throws RemoteException, CreateException
{
```

```
        //BEGIN ----------------- PS(Prototyp,MyClass,bean_state_postcreate)
        // Custom state post-create code.
        //END   ----------------- PS(Prototyp,MyClass,bean_state_postcreate)

    }

    public void ejbActivate() throws RemoteException
    {
        super.ejbActivate();

        //BEGIN ----------------- PS(Prototyp,MyClass,bean_activate)
        // Custom activate code.
        //END   ----------------- PS(Prototyp,MyClass,bean_activate)

    }

    public void ejbPassivate() throws RemoteException
    {
        //BEGIN ----------------- PS(Prototyp,MyClass,bean_passivate)
        // Custom passivate code.
        //END   ----------------- PS(Prototyp,MyClass,bean_passivate)

        super.ejbPassivate();
    }

    public void ejbRemove() throws RemoteException, RemoveException
    {
        //BEGIN ----------------- PS(Prototyp,MyClass,bean_remove)
        // Custom remove code.
        //END   ----------------- PS(Prototyp,MyClass,bean_remove)

        super.ejbRemove();
    }

    public void ejbLoad() throws RemoteException
    {
        super.ejbLoad();

        //BEGIN ----------------- PS(Prototyp,MyClass,bean_load)
        // Custom load code.
        //END   ----------------- PS(Prototyp,MyClass,bean_load)

    }
```

```
    public void ejbStore() throws RemoteException
    {
        //BEGIN ----------------- PS(Prototyp,MyClass,bean_store)
        // Custom store code.
        //END   ----------------- PS(Prototyp,MyClass,bean_store)

        super.ejbStore();
    }

    public byte[] getId() throws RemoteException
    {
        AssertionException.assert(m_id);

        if (m_nativeObj == null)
        {
            throw new RemoteException(''MyClassBean is no longer valid.'');
        }

        try
        {
            return m_nativeObj.getOid(m_id);
        }
        catch(ContainerException exc)
        {
            throw new RemoteException(exc.getMessage());
        }
    }

    public boolean getIdNull() throws RemoteException
    {
        AssertionException.assert(m_id);

        if (m_nativeObj == null)
        {
            throw new RemoteException(''MyClassBean is no longer valid.'');
        }

        try
        {
            return m_nativeObj.getNull(m_id);
        }
        catch(ContainerException exc)
```

```
{
    throw new RemoteException(exc.getMessage());
}
}

public Object getPrimaryKey() throws RemoteException
{
    return new MyClassKey(getId());
}

public EntityState toState() throws RemoteException
{
    MyClassState ret = null;

    try
    {
        ret = new MyClassState(getId(), getIdNull(), m_context.getEJBObject());
    }
    catch(RemoteException exc) {}

    return ret;
}

public void setState(EntityState state) throws RemoteException
{
    this._setState(state);
}

protected void _setState(EntityState state) throws RemoteException
{
    if (state instanceof MyClassState)
    {
        MyClassState tmpState = (MyClassState) state;

        if(tmpState.getIdNull())
        {
            if(!getIdNull())
            {
                throw new RemoteException(''Required attribute is null.'');
            }
        }
        else
        {
```

```
                if(getIdNull())
                {
                    throw new RemoteException(''Key attribute mismatch.'');
                }
                else
                {
                    if (!(ByteArrayUtil.isEquivalent(tmpState.getId(), getId
                    {
                        throw new RemoteException(''Key attribute mismatch''
                    }
                }
            }
        }
        else
        {
            throw new ClassCastException(''Invalid state object type.'');
        }
    }

    protected MyClassBean() throws RemoteException
    {
    }

    protected MyClassBean(NativePObject pobj) throws RemoteException
    {
        super(pobj);
    }

    static NativeClassDesc m_myClass;
    static NativeAttrDesc m_id;

    static
    {
        m_myClass = null;
        m_id = null;
    }

}
```

A.4 MyClassKey.java

```
// ============================================================================
// File: MyClassKey.java
// Generated by the Persistence(TM) Object Builder from
// Persistence Software, Inc.
// ============================================================================
// The Persistence Object Builder software is the property of Persistence
// Software, Inc. Any rights to use are granted under the license agreement.
// Copyright 1998, all rights reserved.
// ============================================================================

package prototyp;

import java.math.BigDecimal;
import java.util.Date;
import com.persistence.container.internal.ByteArrayUtil;


public class MyClassKey
    implements java.io.Serializable
{
    /**
    @deprecated constructor - should use constructor with parameters
    */
    public MyClassKey() {}

    public MyClassKey(byte[] id)
    {
        if(id == null)
        {
            throw new NullPointerException(''Null attribute parameter.'');
        }
        m_id = id;
    }

    public byte[] getId() { return m_id; }

    public void setId(byte[] id)
    {
        if(id == null)
        {
            throw new NullPointerException(''Null attribute parameter.'');
```

```
        }
        m_id = id;
    }

    public int hashCode()
    {
        int ret = 0;

        ret ≙ ByteArrayUtil.getOIDHashCode(m_id);

        return ret;
    }

    public boolean equals(Object obj)
    {
        boolean ret = false;

        if (this == obj)
        {
            ret = true;
        }
        else if (obj != null)
        {
            if (obj.getClass() == getClass())
            {
                MyClassKey that = (MyClassKey)obj;
                ret = ((ByteArrayUtil.isEquivalent(m_id, that.m_id)));
            }
        }

        return ret;
    }

    protected byte[] m_id;
}
```

A.5 MyClass.des

```
// ============================================================================
// File: MyClass.des
// Generated by the Persistence(TM) Object Builder from
```

```
// Persistence Software, Inc.
// ============================================================================
// The Persistence Object Builder software is the property of Persistence
// Software, Inc. Any rights to use are granted under the license agreement.
// Copyright 1998, all rights reserved.
// ============================================================================

//BEGIN ----------------- PS(Prototyp,MyClass,deployment_descriptor)
// Custom deployment descriptor.
//END   ----------------- PS(Prototyp,MyClass,deployment_descriptor)


PSEntity
{
    ProjectName
    {
        Prototyp
    }
    EnterpriseBeanClassName
    {
        prototyp.MyClassBean
    }
    Attribute
    {
        AttributeName
        {
            Id
        }
        MemberName
        {
            prototyp.MyClassBean.m_id
        }
        ColumnName
        {
            id
        }
        PrimaryKey
        {
            true
        }
        Type
        {
            OIDVAL
```

```
        }
        RequiredAttribute
        {
            false
        }
        NullAllowed
        {
            false
        }
        OptimisticControl
        {
            false
        }
    }
    OidClass
    {
        true
    }
    TableName
    {
        MyClass
    }
    CreateParameters
    {
    }
    GeneratedMethods
    {
        getId void
        getIdNull void
        setState com.persistence.container.EntityState
    }
}
```

Anhang B

Inhaltsverzeichnis der CD

Die beigefügte CD enthält folgende Dateien/Verzeichnisse:

- DA.pdf
  Die Diplomarbeit als PDF-Dokument
- /AcrobatReader
  Software, mit der PDF-Dokumente gelesen werden können
- /Original
  Originalquellen der Beispielapplikation von Malte Hülder
- /PowerTier/config
  Konfigurationsdateien für den PowerTier
- /PowerTier/jars
  JAR-Archive der portierten Komponenten
- /PowerTier/PowerTier Builder
  Modelle der Container-Managed EntityBeans
- /PowerTier/source
  Quellcode der auf den PowerTier portierten Anwendung
- /WebSphere/jars
  JAR-Archive der portierten Komponenten
- /WebSphere/source
  Quellcode der auf den WebSphere portierten Anwendung

Literaturverzeichnis

[Bal96] Balzert, Helmut: *Lehrbuch der Software-Technik*. Spektrum Akademischer Verlag GmbH, 1996

[BK88] Buschhorn, Michael ; Kuchnowski, Hans J.: *Entwicklung eines Portabilitätsmaßes und Entwurf eines Tools zur Unterstützung der Portierung*, Universität Dortmund, Fachbereich Informatik, Diplomarbeit, 1988

[BM00] Bauer, Nikolai ; Mandl, Peter: Rollenspiel. Chancen und Risiken der EJB-Komponententechnik. In: *iX* (2000), Jan., S. 108. – Verlag Heinz Heise

[Bro77] Brown, Peter J.: *Software Portability, An Advanced Course*. Cambridge University Press, 1977

[Con] Java Coding Conventions. – http://java.sun.com/docs/codeconv/html/CodeConv (30.08.2001)

[Con00] Converting Powertier EJBs To WebLogic. 2000. – 2-Seitiges Dokument des GHA-Projekts der Zürich Agrippina (31.05.2000)

[DeM00] DeMichiel, Yalcinalp: Enterprise JavaBeans, Spezifikation 2.0. 2000. – Sun Microsystems, Proposed Final Draft, http://java.sun.com/products/ejb/docs.html (18.02.2001)

[Fie00] Fields, Duane K.: Looking Into Enterprise JavaBeans. 2000. – wsyiwyg://237/http://developer.netscape.com/viewsource/fields_beans.html (31.05.2000)

[Fla97] Flanagan, David: *Java in a Nutshell, 2nd Edition*. O'Reilly, 1997

[Fla00] Application Server Comparison Matrix. Dez. 2000. – http://www.flashline.com/components/appservermatrix.jsp (12/2000)

[Gla00] Glahn, Kay: Stets zu Diensten. In: *Java Magazin* (2000), Dez., S. 30. – Software & Support Verlag GmbH

[Gra97a] Grand, Mark: *Java Fundamental Classes Reference*. O'Reilly, 1997

[Gra97b] Grand, Mark: *Java Language Reference, 2nd Edition*. O'Reilly, 1997

[Gru99] Gruhn, Volker: *Softwaretechnologie III. Komponentenbasierte Software-Entwicklung*, Universität Dortmund, Fachbereich Informatik, Vorlesungsunterlagen, 1999

[Har00] Harmon, Paul: Enterprise Application Servers. Jan. 2000. – Cutter Consortium, http://www.cutter.com/cds/cds0001.html (01/2000)

[Haw00] Hawlitzek, Florian: Im Wandel der Zeit. In: *Java Spektrum* (2000), Nov., S. 51. – Verlag 101communications Deutschland GmbH

[HM98] Hapner, Mark ; Matena, Vlada: Enterprise JavaBeans, Spezifikation 1.0. 1998. – http://java.sun.com/products/ejb/docs.html (18.02.2001)

[HM99] Hapner, Mark ; Matena, Vlada: Enterprise JavaBeans, Spezifikation 1.1. 1999. – http://java.sun.com/products/ejb/docs.html (18.02.2001)

[Hue00] Huelder, Malte: *Erfassung der zeitabhängigen Entwicklung von Geschäftsobjekten im Kontext der Historisierung von Versicherungsverträgen mit Enterprise JavaBeans*, Universität Dortmund, Fachbereich Informatik, Diplomarbeit, 2000

[Jee01] J2EE Glossar (Sun Microsystems, Inc.). 2001. – http://www.java.sun.com/j2ee/glossary.html (14.06.2001)

[jGu00] Enterprise JavaBeans Technology Fundamentals Short Course. 2000. – jGuru, http://developer.java.sun.com/developer/onlineTraining/EJBIntro/EJBIntro.html (25.02.2000)

[Kle00] Klein, Manuel: Bohnen-Power in jeder Schicht. In: *Java Magazin* (2000), Dez., S. 57. – Software & Support Verlag GmbH

[Mer00a] Merkle, Bernhard: Aufgefrischt. In: *iX* (2000), Feb., S. 120. – Verlag Heinz Heise

[Mer00b] Merkle, Bernhard: Geschäftsbohnen. In: *iX* (2000), Dez., S. 68. – Verlag Heinz Heise

[Mer00c] Merkle, Bernhard: Große Sprünge. EJB 2.0: Portable Beans und neue Query Language. In: *iX* (2000), Aug., S. 135. – Verlag Heinz Heise

[MH99a] Monson-Haefel, Richard: Create Forward-Compatible Beans In EJB, Part 1. 1999. – http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-ejb1_p.html (31.05.2000)

[MH99b] Monson-Haefel, Richard: *Enterprise JavaBeans*. O'Reilly, 1999

[MH00] Monson-Haefel, Richard: Create Forward-Compatible Beans In EJB, Part 2. 2000. – http://www.javaworld.com/javaworld/jw-01-2000/jw-01-ssj-ejb2_p.html (31.05.2000)

[Nag90] Nagl, Manfred: *Softwaretechnik: Methodisches Programmieren im Großen*. Springer-Verlag, 1990

[Per00] Powertier for J2EE Release Notes - VERSION 6.53X. Dez. 2000

[Per01] Dokumentation zum Applikationsserver "PowerTier". 2001. – http://www.persistence.com/powertier/papers/index.html#EJBPAPERS (15.02.2001)

[Roe90] Roeing, Ernst F.: *Verbesserung der Software-Portabilität durch Standardisierung: Ein Fallbeispiel zur Benutzerschnittstelle SAA*, Universität Dortmund, Fachbereich Informatik, Diplomarbeit, 1990

[Rom99] Roman, Ed: Mastering Enterprise JavaBeans and the Java 2 Platform, Enterprise Edition. 1999. – http://theserverside.com/resources/book.jsp (10.03.2001)

[Sch80] Schneider, Hans J. (Hrsg.): *Portable Software (Berichte des German Chapter of the ACM)*. Bd. 4. 1980

[Som89] Sommerville, Ian: *Software Engineering, Third Edition*. Addison-Wesley, 1989

[S&SV00] Software & Support Verlag, GmbH (Hrsg.): *Java Magazin* Software & Support Verlag GmbH, Dez. 2000

[Sun99] Sundsted, Todd: Application Servers - An Introduction. 1999. – http://java.sun.com/events/jbe/98/features/appservers.html (16.02.2001)

[Tho98] Thomas, Anne: Enterprise JavaBeans Technology. Server Component Model for the Java Platform. 1998. – Patricia Seybold Group, http://www.java.sun.com/products/ejb/pdf/white_paper.pdf (20.12.2000)

[Wal82] Wallis, Peter J.: *Portable Programming*. The Macmillan Press Ltd., 1982

[Web01a] Dokumentation zum Applikationsserver "WebLogic". 2001. – http://www.weblogic.com/docs/resources.html (15.02.2001)

[Web01b] Dokumentation zum Applikationsserver "WebSphere". 2001. – http://www-4.ibm.com/software/webservers/appserv/support.html (15.02.2001)

[Zuk99] Zukorwski, John: Welcome To The Server-Side Java Series. 1999. – http://www.javaworld.com/javaworld/jw-12-1999/jw-12-ssj-welcome_p.html (31.05.2000)

Abbildungsverzeichnis

Index