Diplomarbeit

Generierung serverseitiger Komponenten basierend auf UML Profiles

bearbeitet von

Sean Eikenberg

geboren am 10. Juni 1976 in Hannover

an der

Technischen Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Betreuer: Dipl. Inform. Mike Fischer
Hochschullehrer: Prof. Dr. rer. nat. habil. Heinrich Hußmann

Eingereicht am 30. April 2002

[Original oder Kopie der Aufgabenstellung]

Vorwort

Lieber Leser,

hinter mir liegen nun sechs äußerst anstregende, ärgerliche, aber auch ermutigende, abwechslungsreiche, produktive, erfolgreiche und machmal sogar lustige Monate, in denen die nachfolgende Diplomarbeit entstanden ist.

Ich will nicht leugnen, dass ich mir in der zurückliegenden Zeit manchmal gewünscht habe, das Projekt „Diplomarbeit“ gar nicht erst angefangen zu haben - und doch möchte ich im Nachhinein diesen geistigen Prozess nicht missen!

Bevor ich mich nun aber im Detail über die mir eigenen Beweggründe auslasse, will ich das Vorwort lieber zur Danksagung an all diejenigen nutzen, die mir bei der Fertigstellung der Diplomarbeit geholfen haben.

Mein Dank gilt insbesondere meinem Vater, Herrn Stefan Rahn sowie Frau Kerstin Zeuss, die alle die undankbare Arbeit des Korrektors übernommen haben, um mir bei der Wahrung eines hohen Rechtschreibniveaus zu helfen.

Auch möchte ich mich bei Herrn Norbert Graf bedanken, der mir einige wichtige Denkanstöße bezüglich der Abbildung von ‚Container-Managed-Relationships’ gegeben hat.

Dank auch an all diejenigen, die mir allein durch ihren Beistand über manch schwere Zeit hinweg geholfen haben.

Ganz besonders möchte ich mich aber auch bei meinem Betreuer, Herrn Mike Fischer, bedanken, der mir durch seine unzähligen Ratschläge sowie der enormen Geduld, mit der er mir in den Konsultationen begegnet ist, maßgeblich bei der Erstellung der Diplomarbeit geholfen hat!

Abschließend bleibt mir nur noch übrig, dem Leser viel Spaß beim Studium der nachfolgenden Diplomarbeit zu wünschen!

Dresden, den 30. April 2002

Inhaltsverzeichnis

Kapitel 1
Einleitung

Gemäß der Präambel des “UML Profiles for EJB” [RAT01, Seite 8] und gemäß [Dejo01, WWW08, WWW09] ist die Unified Modeling Language (UML) eine weit verbreitete Modellierungs- und Beschreibungssprache für unternehmensbezogene Software. In diesem Zusammenhang zeichnet sich marktseitig auch ein großer Anteil von EJB-basierter Software im Unternehmensbereich ab [META01, WWW06, WWW07]. Daher ist es naheliegend, die UML als Modellierungs- und Beschreibungssprache für EJB-basierte Softwareprojekte zu benutzen.

Bei der genaueren Betrachtung der Mächtigkeit der UML als Modellierungs- und Beschreibungssprache ist zwar eine ausreichende Unterstützung des objektorientierten Programmierparadigmas festzustellen, dennoch fehlt eine spezifische Einbindung bestimmter Programmierstandards. Dieses ist jedoch nicht vermeidbar, da sich ein breit akzeptierter Industriestandard wie die UML gerade durch seine Generalität auszeichnet. In diesem Zusammenhang spielt aber auch die standardisierte Erweiterungsmöglichkeit der UML eine wichtige Rolle. Sie ermöglicht damit eine flexible Anpassung an spezielle Bereiche der Softwaretechnologie, wie z.B. an bestimmte Komponentenmodelle, sichert aber dennoch die Einheitlichkeit des Standards.

Auf diesen Erweiterungsstandard, der als UML-Profile in der Literatur geführt wird, geht der Große Beleg mit dem Titel „Werkzeugunterstützung für UML-Profiles“ von Herrn Andreas Pleuß detailliert ein [Pleu01].

Basierend auf diesem Erweiterungsstandard wurde laut [RAT01, Seite 8] in der Vergangenheit verschiedentlich versucht, die UML an das EJB-Komponentenmodell anzupassen. Da dies aber in der Regel Versuche einzelner Firmen waren, deren UML-Erweiterungen sich am Markt nicht durchsetzten, beschloss ein Firmenkonsortium unter der Führung der Rational Software Corporation [WWW01] einen einheitlichen Industriestandard für die Modellierung und Beschreibung von EJB-basierter Software einzuführen. Dieser Standard, der folglich unter dem Titel „UML Profile for EJB“ geführt wird, soll, vereinfacht dargestellt, den kompletten Inhalt eines beliebigen EJB-Jars

beschreiben und mit ihm auch die dort enthaltenen EJB-Komponenten sowie ihre Beziehungen untereinander.

1.1 Problemstellung

Im Rahmen eines softwaretechnologisch korrekten Entwicklungsprozesses ist eine detaillierte Modellierung von komponentenbasierten Anwendungen sowohl auf Analyse- als auch auf Entwurfsebene wünschenswert. Diese Modellierung sollte insbesondere die präzise Beschreibung der Komponentenbeziehungen umfassen.

Nun wurde jedoch gerade der Aspekt der Komponentenbeziehungen im Rahmen des „UML Profiles for EJB“ stark vernachlässigt [RAT01, Seite 63].

Als Begründung hierfür wird die mangelnde Unterstützung des Erstellens, Löschens und Verwaltens von Beziehungen zwischen EJBs durch die EJB-1.1-Spezifikation [SUN99] angegeben. Dadurch sei es nicht möglich, die UML-Semantik in Bezug auf Assoziationen, z.B. im konkreten Fall der Komposition, im Rahmen des Profiles umzusetzen.

Ein weiterer Kritikpunkt am „UML Profile for EJB“ ist die nicht vorhandene Unterstützung der aktuellen EJB-2.0-Spezifikation [SUN01]. Auch werden die verbesserten Erweiterungsmechanismen der UML 1.4 [OMG01, Seite „2-74“ff.] nicht berücksichtigt.

1.2 Aufgaben

Aufgrund der im vorhergehenden Abschnitt formulierten Problemstellung sollen zunächst allgemeine Anforderungen an die Beschreibung der Kombination von serverseitigen Komponenten gesammelt und spezifiziert werden. In diesem Zusammenhang sind auch die aktuellen Standardisierungsbemühungen der Object Management Group (OMG) bezüglich der Modellierung von komponentenbasierter Software (Stichwort EDOC) [OMG02] zu untersuchen und gegebenenfalls zu berücksichtigen.

Die zunächst allgemein spezifizierten Anforderungen sind dann konkret an die aktuelle EJB-2.0-Spezifikation anzupassen und dienen somit auch als Verifikationsgrundlage für die aktuelle Version des „UML Profile for EJB“.

Aufbauend auf den Verifikationsergebnissen sollen im Anschluss die notwendigen Verbesserungen bzw. Erweiterungen am Profile vorgenommen werden. Diese beziehen sich ausdrücklich auch auf die bisher fehlende EJB-2.0-Unterstützung sowie auf die nicht genutzten UML-1.4-Erweiterungsmechanismen.

In einem späteren Schritt sind dann, basierend auf der Semantik des bis dahin überarbeiteten Profiles, Abbildungsvorschriften zu definieren, die eine automatische Generierung von EJB-spezifischen Codeartefakten auf Basis von Profile-basierten Entwurfsmodellen ermöglichen.

Im Rahmen des praktischen Teils dieser Arbeit ist eine prototypische Implementierung eines Generators zu erstellen, der auf Grundlage der zuvor definierten Abbildungsvorschriften, die dort definierte Transformation von EJB-Entwurfsmodellen in Codeartefakte umsetzt. Die als Datenbasis zugrunde liegenden Modelle sollen auf dem Metadatenaustauschformat XMI in der Version 1.1 [OMG00b] basieren.

Um die prinzipielle Funktionsfähigkeit des Generators nachzuweisen, sind einige einfache Beispielmodelle zu erstellen, die dann demonstrativ in EJB-Codeartefakte umgewandelt werden sollen.

1.3 Kapitelübersicht

Es folgt nun eine kurze inhaltliche Übersicht der weiteren Kapitel dieser Arbeit.

In Kapitel 2 werden gemäß der Aufgabenstellung zunächst allgemeine Anforderungen an die Modellierung von komponentenbasierter Software spezifiziert. Aktuelle Standardisierungbemühungen der OMG, wie die Model-Driven-Architecture (MDA) und das „UML Profile for EDOC (Enterprise Distributed Object Computing)“, werden dabei, so weit wie möglich, berücksichtigt.

Kapitel 3 dient der Vorstellung der momentan aktuellen Version des „UML Profile for EJB“.

Kapitel 4 untersucht auf Basis der in Kapitel 2 spezifzierten Anforderungen, ob die aktuelle Version des Profiles für die EJB-basierte Komponentenmodellierung verwendet

werden kann. Dort, wo das Profile den Anforderungen nicht genügt, werden Verbesserungsvorschläge erläutert, die dann den beiden folgenden Kapiteln als Arbeitsgrundlage dienen.

In Kapitel 5 wird die Profile-Struktur optimiert. Dies geschieht zunächst einmal unter dem Aspekt der fehlenden UML-1.4-Konformität. Zum anderen geht es um das Entfernen von im Profile vorhandenen Redundanzen.

In Kapitel 6 wird das Profile auf Basis der vorangegangenen Optimierung um die Neuerungen der EJB-2.0-Spezifikation erweitert.

Kapitel 7 definiert die, für die Generierung der EJB-Codeartefakte notwendigen, Abbildungsvorschriften.

Im Rahmen des 8. Kapitels werden die erarbeiteten Abbildungsvorschriften mittels eines prototypischen Generators, dessen prinzipiell korrekte Funktionsweise anhand zweier einfacher Beispiele nachgewiesen wird, umgesetzt.

In Kapitel 9 erfolgt schließlich eine Zusammenfassung der in dieser Arbeit erzielten Ergebnisse. Dort, wo im Rahmen dieser Arbeit offene Punkte entstanden oder geblieben sind, soll versucht werden, mögliche Lösungsansätze im Sinne eines Ausblicks zu skizzieren.

Dem Anhang dieser Arbeit können sowohl die Originalfassung des Profiles (Anhang A), die im Rahmen von Kapitel 6 erweiterte Endfassung (Anhang C) als auch das in Kapitel 5 nach der Optimierung erzielte Zwischenergebnis (Anhand B) entnommen werden. Die Notationsweise folgt weitestgehend (mit Ausnahme der Constraints-Notationsweise) der Vorgabe der UML-1.4-Spezifikation.

Kapitel 2
Anforderungen an die Komponentenmodellierung

Dieses Kapitel dient der Spezifizierung von Anfoderungen an die Komponentenmodellierung im Allgemeinen sowie bezogen auf die EJB-2.0-Technologie im Speziellen. Die spezifizierten Anforderungen umfassen sowohl die Beschreibung der einzelnen Komponenten eines Modells im Sinne ihrer Struktur, d.h. die jeweilige Komponentenspezifikation samt der in ihr enthaltenen Schnittstellen und Realisierungsartefakte, als auch die Beschreibung der in diesem Modell untereinander bestehenden Komponentenbeziehungen, die auch unter dem Begriff der Komponentenrelationen zusammengefasst werden können.

Um die Spezifizierung der Anforderungen nachvollziehbar zu gestalten, folgt der Aufbau dieses Kapitels einem inhaltlich klar strukturierten Konzept. So wird zunächst eine allgemeine Kontextbestimmung vorgenommen, indem schon existierende oder sich in der Entwicklung befindliche Ansätze bzw. Standardisierungsbemühungen bezüglich der Komponentenmodellierung vorgestellt werden. Als nächstes erfolgt die Begriffsbestimmung des Wortes Komponentenstruktur, an der sich die Formulierung der komponentenstrukturabhängigen Anforderungen anschließt. Schließlich erfolgt in den letzten beiden Abschnitten dieses Kapitels die Begriffsbestimmung des Wortes Komponentenrelationen sowie, analog zur Komponentenstruktur, die Formulierung der komponentenrelationsabhängigen Anforderungen. In beiden Begriffsbestimmungen wird nochmals zwischen der allgemeinen (Überschrift: **Allgemeine Begriffsbestimmung**) und der EJB-bezogenen (Überschrift: **EJB-bezogene Begriffsbestimmung**) Definition unterschieden; die Formulierung der sich daraus ergebenden Anforderungen bezieht sich jedoch immer konkret auf die EJB-2.0-Technologie.

2.1 Kontextbestimmung

Dass die Beschreibung und Modellierung von komponentenbasierter Software kein absolut neues Thema im Rahmen der Softwaretechnologie ist, zeigen die aktuellen Standardisierungsbemühungen der OMG. Mit der „Model-Driven Architecture“ (MDA) sowie dem in ihrem Kontext angesiedelten „Profile for Enterprise Distributed Object

Computing“ (EDOC) existiert insgesamt eine Architektur, die die effektive und nachhaltige Modellierbarkeit von komponentenbasierter Unternehmenssoftware verspricht.

2.1.1 „Model-Driven Architecture“ (MDA)

Die “Model-Driven Architecture”, oder auch MDA, definiert im Kern einen Ansatz, der die Trennung der jeweils spezifischen Systemfunktionalität eines Modells von dessen konkreten Implementierung im Rahmen einer bestimmten Architekturplattform ermöglicht. Anders formuliert bedeutet das, dass die MDA die nötige Infrastruktur zur Verfügung stellt, die zur Spezifizierung von programmiersprachen-, anbieter- und Middleware-unabhängigen Systemmodellen notwendig ist. Die MDA erhebt dabei den Anspruch, für solche Systemmodelle den kompletten Lebenszyklus abzudecken, d.h. deren Analyse-, Entwurfs- und Implementierungsphase, aber auch das sich anschließende „Deployment“ sowie Management des dann installierten Systems. Im Rahmen des ihr eigenen Architekturmodells greift die MDA ihrerseits auf die schon bestehenden Standards „Unified Modeling Language“ (UML), „Meta Object Facility“ (MOF), „XML Metadata Interchange“ (XMI) sowie „Common Warehouse Metamodel“ (CWM) zurück.

Der Prozess zur Erstellung von MDA-basierten Applikationen unterteilt sich, grob gesprochen, in folgende drei Phasen:

1. Erstellung eines plattformunabhängigen Modells (PIM).
2. Standardisierte Abbildung eines PIM auf eine plattformspezifische Architektur; dadurch Entstehung eines plattformspezifischen Modells (PSM).
3. Implementierung des PSM.

Zusammenfassend kann festgestellt werden, dass die MDA durch zwei wesentliche Merkmale gekennzeichnet ist:

1. Durch die Unabhängigkeit der Systemmodelle (PIMs) von konkreten Architekturplattformen und damit auch Implementierungsmodell len (PSMs).
2. Durch die Möglichkeit der formalspezifizierbaren Abbildung der PIMs (über den Zwischenschritt eines PSMs) auf verschiedene konkrete Architekturplattformen, wie z.B. CORBA, EJB oder .NET.

Weitergehende Informationen bezüglich der MDA und ihrer Anwendung sind [NT01, OMG00c, OMG01b, OMG01c, Poo01] zu entnehmen.

2.1.2 „Profile for Enterprise Distributed Object Computing“ (EDOC)

Das sich momentan noch in der Entwicklung befindliche “Profile for Enteprise Distributed Object Computing” (EDOC) dient dem Ziel, die Entwicklung von komponentenbasierter verteilter Unternehmenssoftware zu vereinfachen, indem es eine Modellierungsplattform zur Verfügung stellt, die auf der UML-1.4-Spezifikation basiert und die dem Architekturkonzept der MDA (siehe Abschnitt 2.1.1) entspricht.

Die EDOC-Modellierungsplattform besteht im Prinzip aus folgenden vier Teilen:

1. Aus einem Profile, das den Titel „Enterprise Collaboration Architecture“ (ECA) trägt und das die Modellierung von technologieunabhängigen komponentenbasierten Systemen ermöglicht, entsprechend dem PIM-Ansatz der MDA.
2. Aus einem Profile, dass Entwurfsmuster definiert, die im Rahmen der ECA verwendet werden können.
3. Aus einer Menge von vordefinierten MOF-Metamodellen, die, nachdem sie mittels der im „UML Profile for MOF“ definierten Abbildungsregeln (in Form von Mustern) auf technologiespezifische Profiles abgebildet wurden, zur Modellierung von plattformspezifischen Modellen gemäß des MDA-PSM-Ansatzes benutzt werden können.
4. Aus einer Struktur, die die MDA-konforme Anwendung des EDOC ermöglicht.

Deutlich wird auch hier die konsequente Trennung in plattformunabhängige und plattformspezifische Modelle, was aber aufgrund des gewollten Zusammenhangs zur MDA nur eine logische Folge des dort vorgestellten Architekturmodells ist.

Im Rahmen der EDOC-Spezifikation wird auch der Zusammenhang zum „UML Profile for EJB“ deutlich. Laut Spezifikation soll das dort definierte EJB-Metamodell die zukünftige Grundlage für das „UML Profile for EJB“ sein.

Allerdings scheint es noch keine Einigung diesbezüglich zu geben, so dass die aktuelle Version des „UML Profile for EJB“ weiterhin die Gundlage dieser Arbeit bildet.

Weitergehende Informationen bezüglich des EDOC sind [OMG02] zu entnehmen.

2.1.3 Eigenes (vereinfachtes) Architekturmodell

Um im weiteren Verlauf die nötige Begriffsbestimmung so einfach wie möglich zu halten, werden zwar die wesentlichen Architekturkonzepte (PIM und PSM) der beiden vorhergehenden Standardisierungsbemühungen (MDA und EDOC) übernommen, sie werden jedoch in eine weitestgehend vereinfachte Darstellungsweise neu eingebettet.

Dazu wird im Rahmen dieser Arbeit eine Unterteilung der Entwurfsphase in zwei sogenannte Modellierungsebenen vorgenommen. Diese Unterteilung findet sich in der folgenden Definition wieder, die in enger Anlehnung an die in [CD00, Seite 22f.] eingeführten Modellebenen steht.

Definition

Das fachliche Modell, dass in der Analysephase angesiedelt ist, stellt eine softwareunabhängige Beschreibung von verwendbaren Konzepten einer bestimmten Domäne dar.

Das technologieunabhängige Modell ist in der Entwurfsphase angesiedelt und definiert die Spezifikation von komponentenbasierter Software, d.h. es modelliert die jeweiligen Schnittstellen der einzelnen Komponenten. Dabei bezieht es sich ausdrücklich nicht auf ein bestimmtes Komponentenmodell.

Das technologieabhängige Modell ist ebenfalls in der Entwurfsphase angesiedelt und beschreibt die Implementierungsdetails der komponentenbasierten Software. Die Implementierungsdetails fundieren dabei auf einem bestimmten Komponentenmodell.

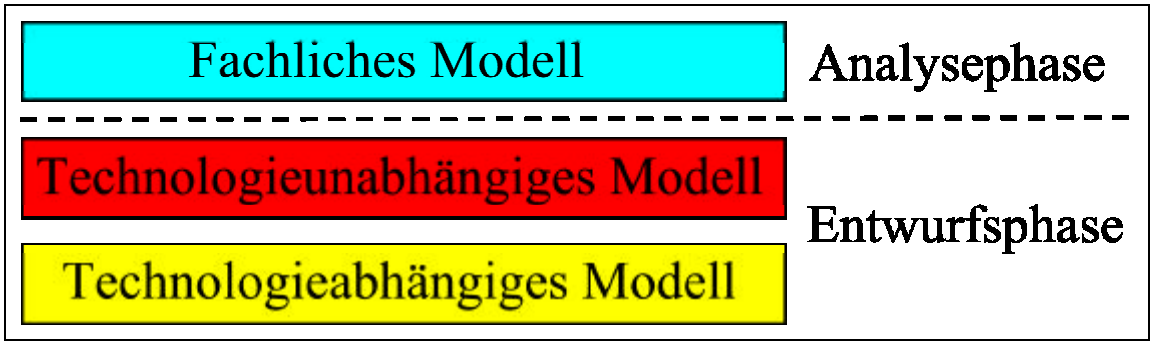


Abbildung 1- Eingeführte Modellierungsebenen

Im weiteren Verlauf werden gemäß der Aufgabenstellung nur EJB-spezifische Anforderungen an die Komponentenmodellierung spezifiziert, die dann entsprechend der vorhergehenden Information zur vollständigen Beschreibung von technologieabhängigen Modellen führen.

2.2 Komponentenstruktur

Allgemeine Begriffsbestimmung:

Gemäß [CD00, Seite 7] gehören zu einer Komponente mehrere Bestandteile:

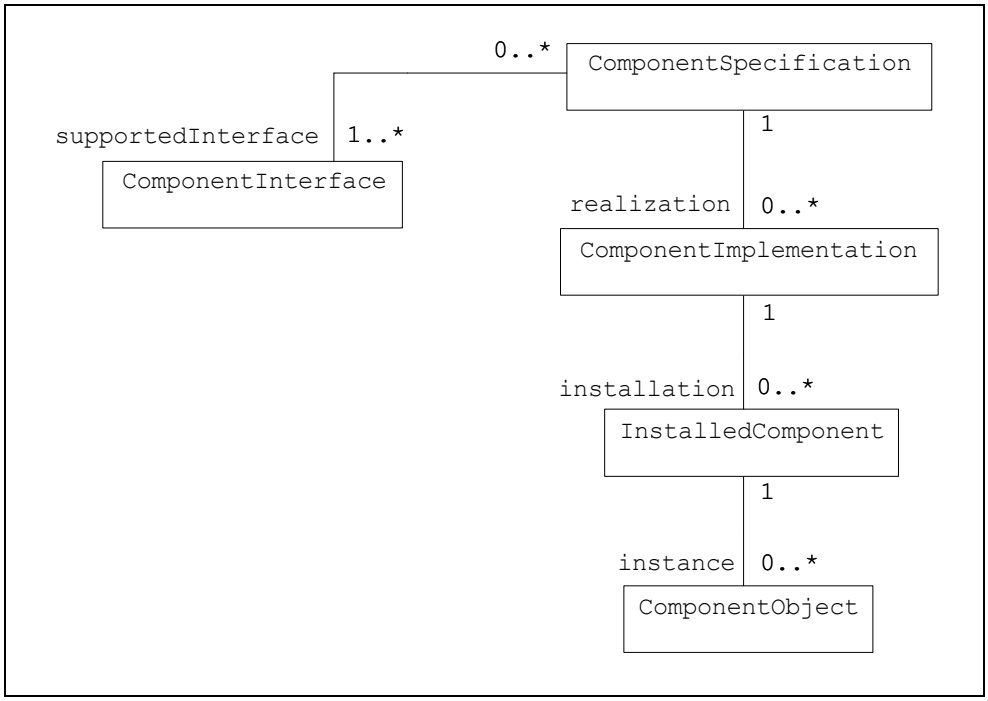


Abbildung 2 - Bestandteile einer Komponente [CD00, Abbildung 1.2]

Der Ausgangspunkt einer Komponente ist ihre Komponentenspezifikation. Diese ist dadurch gekennzeichnet, dass sie mindestens eine Komponentenschnittstelle unterstützt. Damit eine Komponente auch installiert und genutzt werden kann, benötigt sie eine Komponentenimplementierung. Diese kann wiederum als installierte Komponente innerhalb einer Laufzeitumgebung benutzt werden. Schließlich können zur Laufzeit mehrere Komponenteninstanzen durch die Laufzeitumgebung zur Verfügung gestellt werden.

Um einzelne Komponenten innerhalb des Entwurfs hinreichend zu spezifizieren, muss deren Struktur abgebildet werden. Diese besteht im Kern aus den angebotenen

Komponentenschnittstellen und den zur Implementierung gehörenden Artefakten (Komponentenimplementierung).

EJB-bezogene Begriffsbestimmung:

Die konkrete Komponentenstruktur einer EJB ist in [SUN01] festgelegt und kann wie folgt zusammengefasst werden:

Eine „Enterprise JavaBeans"-Komponente (EJB) besitzt

- keine (Message-Driven-Bean), eine oder zwei Home-Schnittstellen;
- in Abhängigkeit von der Anzahl der Home-Schnittstellen keine, eine oder zwei Component-Schnittstellen;
- eine Implementierungsklasse, die über einen internen Zustand verfügen kann (zustandsbehaftete Session-Bean);
- in Abhängigkeit vom Persistenztyp eventuell eine Primärschlüsselklasse (Entity-Bean);
- einen Deployment-Descriptor, der als Rahmen für die Komponentenspezifikation dient;
- unter Umständen Hilfsklassen, die zur Komponentenimplementierung benötigt werden.

Gemäß der allgemeinen Definition des Begriffs Komponentenstruktur wird folgende konkrete Zuordnung zum EJB-Komponentenmodell verwendet:

Tabelle 1 - Begriffszuordnung: Komponentenstruktur <-> EJB

Allgemeine Komponentenstruktur	„Enterprise JavaBeans"
Komponentenspezifikation	Deployment-Descriptor, Primärschlüsselklasse
Komponentenschnittstelle	Home- und Component-Schnittstelle
Komponentenimplementierung	Implementierungsklasse, Hilfsklassen

2.3 Komponentenstrukturabhängige Anforderungen

Zur vollständigen Beschreibung komponentenbasierter Modelle gehört auch die jeweilige Komponentenstruktur. Somit sind in diesem Bereich auch Anforderungen an die Komponentenmodellierung zu suchen.

Allgemein gesprochen besteht zunächst die Anforderung, dass die Komponentenstruktur der einzelnen Komponenten im jeweiligen Entwurf vollständig zu modellieren, also abzubilden ist.

Bezogen auf das EJB-Komponentenmodell bedeutet das, dass zunächst die Abbildung

- der Home- und Component-Schnittstellen,
- der Implementierungsklasse und
- des Deployment-Descriptors

erfolgen muss. Falls vorhanden, sind auch

- die Primärschlüsselklasse und
- die Hilfsklassen

im Entwurfsmodell abzubilden.

2.4 Komponentenrelationen

Bevor im einzelnen die Modellierbarkeit von Relationen zwischen Komponenten diskutiert wird, gilt es zunächst, das mögliche Spektrum derselben zu untersuchen.

Da gemäß der Aufgabenstellung die UML als Modellierungssprache zu wählen ist, sind zunächst alle dort definierten Relationen auch als Komponentenrelationen denkbar.

Das Metamodell der UML führt die vier Elemente ***Flow***, ***Generalization***, ***Association*** und ***Dependency*** als mögliche Beschreibungselemente für Relationen ein.

Es gilt nun im Einzelfall die Verwendbarkeit dieser Relationen bezogen auf das EJB-Komponentenmodell zu prüfen. Darauf aufbauend sind dann Anforderungen an die Modellierung von Komponentenrelationen aufzustellen.

2.4.1 Flow

Allgemeine Begriffsbestimmung:

Das Metamodellelement ***Flow*** stellt Beziehungen zwischen unterschiedlichen Versionen einer Instanz des Metamodelelements ***ModelElement*** dar. Dabei kann unter dem Begriff „unterschiedliche Version“ auch eine Kopie einer bestimmten Instanz, also ein sogenannter „Klon“ verstanden werden.

Im Rahmen der UML-Spezifikation nimmt das Metamodellelement ***Flow*** insofern eine Sonderstellung ein, als dass es nur im Zusammenhang mit dem Metamodellelement ***Object*** Verwendung findet. Dieses dient der Darstellung von Klasseninstanzen (Objekten).

Da der UML-Spezifikation keine weiteren Hinweise bezüglich. der Anwendung dieser Relation zu entnehmen sind, wird auf sie im Rahmen der Komponetenmodellierung verzichtet.

2.4.2 Generalisierung

Allgemeine Begriffsbestimmung:

Nach [OMG01, Seite „2-70“f.] und nach [HK99] stellt eine Generalisierung eine taxonomische Beziehung zwischen einem spezialisierten Element und einem allgemeinen Element dar, wobei das Subelement alle Merkmale des Superelements erbt und außerdem noch weitere Merkmale hinzufügen kann.

Um Generalisierung innerhalb von UML-Modellen zu beschreiben, wird das Metamodellelement ***Generalization*** verwendet. Es findet hauptsächlich Verwendung in Bezug auf Klassen, ist semantisch jedoch auch auf Instanzen anderer Metamodellelemente anwendbar, wie z.B. auf ***Component*** oder ***Subsystem***.

Nun stellt sich jedoch die Frage, wie die Generalisierung, oder auch Vererbungsrelation, in Bezug auf Softwarekomponenten im Allgemeinen definiert ist. Da sich in verschiedenen, für die Komponentenmodellierung relevanten Literaturquellen, wie z.B. in [GT00] oder in [DW99], keine Angaben zu der Vererbungsrelation finden lassen bzw. in [WBGP01] explizit auf das Nichtvorhandensein der Vererbungsrelation bei der Komponentenmodellierung hingewiesen wird, soll in dieser Arbeit ein eigener Ansatz verfolgt werden.

Die Grundlage der Vererbungsrelation ist die Komponentenstruktur, die sich in die drei Bestandteile Komponentenspezifikation, Komponentenschnittstellen und Komponentenimplementierung unterteilt. Da aus Sicht eines potentiellen Clients zunächst nur die für die Nutzung einer Komponente relevanten Merkmale, also ihre Spezifikation mit den dort offerierten Schnittstellen, von Relevanz sind, wird die Vererbungsrelation auch nur auf diese Eigenschaften angewandt.

Definition:

Wenn eine Komponente eine andere spezialisiert, also in einer Vererbungsrelation zu dieser steht, so erbt sie die Schnittstellen der Superkomponente sowie die in der Komponentenspezifikation festgelegten Eigenschaften. Sie kann sowohl die Schnittstellen als auch die geerbten Spezifikationseigenschaften um eigene Merkmale erweitern.

EJB-bezogene Begriffsbestimmung:

Bezogen auf die EJB-Technologie bedeutet das, dass bei einer Generalisierung einer EJB deren Schnittstellen sowie Teile des Deployment-Descriptors (sofern sie die Eigenschaften der jeweiligen EJB beschreiben) berücksichtigt werden müssen.

Allerdings wird in der Spezifikation des EJB-Komponentenmodells [SUN01, Seite 544] ausdrücklich darauf hingewiesen, dass das Konzept der Komponentengeneralisierung nicht spezifiziert ist. Es findet sich dort zwar ein Hinweis auf die Nutzung der Schnittstellenvererbung, ein genaue Anleitung zur Umsetzung der eigenen Definition (siehe weiter oben) läßt sich daraus aber nicht ableiten.

Zwar ließe sich die Component-Schnittstelle einer EJB relativ problemlos spezialisieren, bei der Home-Schnittstelle stößt das Konzept der Schnittstellenvererbung jedoch relativ schnell an seine Grenzen [WWW05]. So müssten z.B. im konkreten Einzelfall die `Create`-Methoden und die eventuell vorhandenen `Find`-Methoden so überladen werden, dass ihre jeweilige Signatur gleich bliebe. Da diese Methoden aber die zugehörige Remote-Schnittstelle als Rückgabetyp besitzen müssen, wäre hier eine Anpassung in der Subkomponente notwendig. Allerdings ist eine Methodenüberladung, die nur aus der Änderung des Rückgabetyps bestünde, unter Java nicht möglich.

Auch gäbe es bei Entity-Beans eventuell Probleme mit der Primärschlüsselklasse, da diese zu der Spezifikation der jeweiligen EJB gehört und dementsprechend mitvererbt werden müsste. Hier wäre dann die freie Wahl eines Primärschlüssels bei den Subkomponenten nicht mehr möglich, was die Verwendbarkeit einer Vererbungsrelation in Bezug auf persistente EJBs stark einschränken würde.

Aus diesen Gründen wird im Rahmen dieser Arbeit auf die Spezifizierung und Anwendung der Vererbungsrelation verzichtet.

2.4.3 Assoziation

Allgemeine Begriffsbestimmung:

Eine Assoziation stellt nach der UML-Spezifikation [OMG01, Seite „2-65“ff.] eine zur Laufzeit bestehende Verknüpfung zwischen den betreffenden Elementen dar, die im Falle einer binären Assoziation einseitig oder auch beidseitig navigierbar sein kann.

Nach [HK99] beschreibt die allgemeine Assoziation „die gemeinsame Struktur einer Menge von in der Regel statischen Beziehungen zwischen Objekten“.

Zusammenfassend handelt es sich also bei einer Assoziation, oder auch Assoziationsrelation, um eine statische Eigenschaft von Objekten, die diese miteinander verknüpft. Diese Eigenschaft ist, abhängig von der Art der Verknüpfung (unär, binär oder mehrstellig), navigierbar.

Die Assoziationsrelation läßt sich durch zusätzliche Angaben genauer beschreiben. So kann durch die Angabe einer Assoziationsrolle das jeweilige Assoziationsende einen aussagekräftigen Bezeichner bekommen. Des Weiteren kann zusätzlich zur Assoziationsrolle deren Multiplizität angegeben werden. Diese besagt, mit wie vielen Objekten, die die jeweilige Assoziationsrolle einnehmen, zur Laufzeit eine Verknüpfung besteht oder maximal bestehen kann (je nach Aussagekraft der Multiplizität).

Auch können Assoziationen geordnet oder nach einem bestimmten Kriterium sortiert sein, oder durch ein qualifizierendes Attribut in disjunkte Teilmengen zerlegt werden. Für den Fall, dass einer Assoziationsrelation eigene Attribute zugeordnet werden sollen, ist eine Assoziationsklasse zu verwenden.

Die UML erlaubt jedoch auch eine semantisch weitergehende Verfeinerung von Assoziationen. So stehen neben der allgemeinen Assoziation auch die Aggregation sowie die Komposition zur Verfügung.

Die Aggregation wird dann als Beschreibungsmittel eingesetzt, wenn semantisch eine Enthaltenseinbeziehung beschrieben werden soll. Gemäß [HK99] handelt es sich in einem solchen Fall „um eine asymmetrische Beziehung zwischen nicht gleichwertigen Partnern“, und zwar zwischen dem Aggregat und seinen Teilen. Dabei umfasst die Aggregation allerdings keine zusätzliche Semantik bezüglich der an der Aggregation teilnehmenden

Teile. Diese können im Rahmen der Aggregation (im Gegensatz zur Komposition) auch mehreren Aggregaten gleichzeitig zugeordnet sein.

Mittels der Komposition wird eine strenge Enthaltenseinbeziehung beschrieben. Sie besagt, dass die Teile nun mehr maximal einem Aggregat, dem Komposit, zur gleichen Zeit zugeordnet sein dürfen (Multiplizität [0..1]) und das Komposit auch für das Erstellen bzw. Löschen der ihm zugeordneten Teile zuständig ist.

Um den Bereich der Modellierung von Komponentenbeziehungen im Rahmen dieser Arbeit in vertretbarem Aufwand abhandeln zu können, muss eine Abgrenzung erfolgen. So wird im Folgenden zunächst die Untersuchung auf die allgemeine (binäre) Assoziation zwischen Komponenten beschränkt. Im Rahmen dieser vereinfachten Assoziationsrelation sind sowohl die unidirektionale als auch die bidirektionale Navigierbarkeit zugelassen.

Auf die Aggregation als mögliche Relation zwischen Komponenten wird verzichtet, da sie im Hinblick auf die später zu erarbeitenden Abbildungsvorschriften, technisch der allgemeinen Assoziation entspricht [HK99, Seite 277].

Auch auf die Komposition wird verzichtet, da sich insbesondere die Umsetzung der eindeutigen Zuordnung eines Elements zu einem Komposit im Rahmen der EJB-Technologie als schwierig darstellt. So steht es EJBs frei, jederzeit andere EJBs über deren Home-Schnittstelle aufzurufen und damit auch zu referenzieren. Auch fehlt hier die Unterstützung der Kompositionsrelation durch die EJB-Spezifikation – im Rahmen des Komponentenmodells COM+ ist z.B. die Containment- bzw. Aggregationsrelation vorgesehen [GT00].

Weitere zusätzliche semantische Verfeinerungen, wie z.B. die Angabe von Rollennamen, Ordnungen, Sortierungen oder qualifizierten Attributen, werden im Rahmen dieser Arbeit ebenfalls nicht behandelt.

EJB-bezogene Begriffsbestimmung:

Innerhalb des EJB-Komponentenmodells ist die Assoziationsrelation am ehesten mit der Referenzierung von Komponentenschnittstellen über sogenannte EJB-Referenzen gleichzusetzen [DW99, Seite 435f.].

Im Rahmen von Entity-Beans sind aber auch die sogenannten Container-Managed-Relationships (CMRs) von Interesse, da sie im Sinne des EJB-Datenmodells persistente

Relationen zwischen einzelnen Entitäten darstellen. Da die Realisierung von CMRs im Gegensatz zur der Realisierung Bean-Managed-Relationen vorgegeben sind, werden sie ebenfalls als Grundlage für Assoziationsrelationen angesehen.

Um die einzelnen Anforderungen an die Modellierung von Komponentenbeziehungen so einfach wie möglich zu gestalten, werden in den folgenden Abschnitten diejenigen Assoziationsformen zwischen EJBs ausgeschlossen, die technisch zwar möglich, aber von der Architektur des Komponentenmodells eigentlich nicht vorgesehen sind.

Dazu sollen jeweils, ausgehend von den drei EJB-Komponententypen, die möglichen „Assoziationspartner“ (Supplier) untersucht werden. Grundlage hierfür bilden die in der EJB-Spezifikation [SUN01] verwendeten Beispiele, sowie die gängige Fachliteratur [Mon99, MS01, RAJ02] und vorhandene EJB-Entwurfsmuster [Klin02].

2.4.3.1 Allgemeine Einschränkungen

Bevor nun im Einzelnen auf die jeweils möglichen Assoziationen aus Sicht der unterschiedlichen EJB-Komponententypen eingegangen wird, werden zunächst zwei allgemein gültige Einschränkungen (die für alle EJB-Typen gelten) formuliert:

1. Grundsätzlich können nur dann Multiplizitäten größer ‚1’ im Rahmen eines Assoziationsendes verwendet werden, wenn die assoziierte EJB über einen Zustand verfügt. Anderenfalls kann keine Unterscheidung zwischen den einzelnen assoziierten EJB stattfinden.
2. Bidirektionale Assoziationen können immer nur zwischen zustandsbehafteten EJBs gebildet werden. Ohne einen vorhandenen Zustand kann anderenfalls die eindeutige Identifikation des jeweils anderen Assoziationspartners nicht gewährleistet werden.

2.4.3.2 Session-Bean

Session-Beans werden zur Modellierung von Geschäftsprozessen eingesetzt. Sie stellen serverseitig eine zugreifende Client-Rolle dar und verwalten in diesem Rahmen die im jeweiligen Prozess anfallenden Aufgaben und Aktivitäten. Da Geschäftsprozesse einen Zustand besitzen können, werden die Session-Beans in zustandslose und zustandsbehaftete Session-Beans unterteilt.

Zustandsbehaftete Session-Beans werden dann eingesetzt, wenn sich die Implementierung der Aufgaben oder Aktivitäten eines Prozesses auf mehrere Methoden einer Session-Bean

verteilen. In diesem Fall ist es notwendig, den jeweiligen Zustand über den einzelnen Methodenaufruf hinweg zu erhalten.

Zustandslose Session-Beans hingegen umfassen Aufgaben oder Aktivitäten, die vollständig im Rahmen von einzelnen Methoden implementiert werden können. D.h., dass zum einen die zur Bearbeitung der jeweiligen Aktivität oder Aufgabe notwendigen Daten als Parameter vollständig der jeweiligen Methode übergeben werden und zum anderen, dass das Ergebnis dieses Prozesses als Rückgabewert der betreffenden Methode zur Verfügung steht.

Session-Beans werden in der Regel zur Manipulation von Geschäftsdaten verwendet, die mit Hilfe von Entity-Beans (siehe Abschnitt 2.4.3.3) modelliert werden. Daher können Session-Beans grundsätzlich über Assoziationen zu Entity-Beans verfügen. Diese sind trotz des auf beiden Seiten vorhandenen Zustands unidirektional, also nur in Richtung der Entity-Bean navigierbar, da Entity-Beans in der Regel keine zustandsbehafteten Session-Beans assoziieren. Die Multiplizität auf Seiten der assoziierten Entity-Bean ist dabei beliebig, also im [0..*].

Darüber hinaus können Session-Beans aber auch andere Session-Beans assoziieren, um z.B. deren Dienste für den eigenen Geschäftsprozess zu nutzen. Ein typisches Beispiel hierfür ist das Entwurfsmuster „Session Facade“, in welchem eine bestimmte Session-Bean die Dienste der anderen Session-Beans des betreffenden Entwurfs für alle zugreifenden Client-Anwendungen kapselt. Dabei ist jedoch zwischen zustandslosen und zustandsbehafteten Session-Beans zu unterscheiden.

Eine bidirektionale Assoziation mit beliebigen Multiplizitäten auf beiden Seiten ist gemäß Abschnitt 2.4.3.1 nur zwischen zwei zustandsbehafteten Session-Beans möglich.

In einem solchen Fall sollten jedoch sogenannte Ketten von zustandsbehafteten Session-Beans („stateful Session Bean Chain“) vermieden werden. Würde in einem solchen Beispiel eine einzelne Session-Bean dieser Kette, z.B. durch eine auftretende Exception, ungültig werden, ist gleich die ganze Verkettung der EJBs ungültig [Mon99, Seite 283].

Zustandsbehaftete Session-Beans können darüber hinaus zustandslose Session-Beans unidirektional mit einer maximalen Multiplizität von ‚1’ assoziieren.

In der Regel assoziieren zustandslose Session-Beans nur andere zustandslose Session-Beans. Dies liegt daran, dass im Falle einer zustandsbehafteten Session-Bean als Partner, deren Zustand auf Seiten der zustandslosen Session-Bean gar nicht über den Rahmen der einzelnen Methoden hinweg erhalten werden könnte [Mon99, Seite 283] und somit der Verzug eines Zustands gar nicht genutzt werden könnte.

Assoziationen zu zustandslosen Session-Beans sind gemäß Abschnitt 2.4.3.1 unidirektional und besitzten am navigierbaren Assoziationsende eine maximale Multiplizität von ‚1'.

Die Assoziierung von Message-Driven-Beans ist im herkömmlichen Sinne zunächst nicht möglich, da diese über keinerlei Komponentenschnittstellen verfügen, die im Rahmen der Implementierungsklasse referenziert werden könnten.

Dennoch kann das asynchrone Abschicken von JMS-Nachrichten als ‚schwache' Assoziierung einer Message-Driven-Bean angesehen werden. In [RAJ02] wird sogar die Möglichkeit der Implementierung eines Rückgabewerts bei der Nutzung von Message-Driven-Beans diskutiert.

Aus diesem Grund soll eine unidirektionale Assoziation von Mesage-Driven-Beans durch Session-Beans vorgesehen werden. Da Message-Driven-Beans analog zu zustandslosen Session-Beans ebenfalls über keinen Zustand verfügen, wird auch hier eine maximale Multiplizität von ‚1' zugelassen.

2.4.3.3 Entity-Bean

Entity-Beans werden zur Modellierung von persistenten Geschäftsdaten eingesetzt. Sie werden dabei im Rahmen der Container-Managed-Persistence (CMP) auf Datenbanken oder datenbankähnlichen Speichermedien abgebildet und verwalten selber keine Prozesse oder Aktivitäten [RAJ02, Seite 111].

Da zwischen persistenten Geschäftsdaten in der Regel persistente Beziehungen bestehen können, sind diese durch das jeweilige Komponentenmodell zu berücksichtigen.

Dies geschieht im Rahmen der EJB-Spezifikation mit Hilfe der CMR, die zwischen Entity-Beans existierende persistente Assoziationen darstellt.

Ein CMR kann über folgende Eigenschaften verfügen:

- Sie ist uni- oder bidirektional.

- Die durch sie ausgedrückten Multiplizitäten entsprechen entweder der „One-to-One“-, „One-to-Many“- oder „Many-to-Many“-Semantik [OMG, Seite 137ff.].

Des Weiteren können Entity-Beans zwecks der Implementierung ihrer eigenen Fachlogik unidirektionale Assoziationen zu zustandslosen Session-Beans oder Message-Driven-Beans mit der maximalen Multiplizität von ‚1’ besitzen (siehe Abschnitt 2.4.3.1).

2.4.3.4 Message-Driven-Bean

Message-Driven-Beans sind spezielle EJB-Komponenten, die JMS-Nachrichten asynchron empfangen und verarbeiten können.

Gemäß der EJB-Spezifikation besitzt eine Message-Driven-Bean folgende Kurzcharakteristik:

Eine Message-Driven-Bean besitzt

- weder eine Home- noch eine Component-Schnittstelle,
- keinen nach außen sichtbaren Zustand und
- auch keine Identität.

Da Message-Driven-Beans jedoch ebenfalls über eine Implementierungsklasse verfügen, können dort für eigene Zwecke Referenzen auf andere EJBs gehalten werden.

Durch ihren nicht vorhandenen Zustand lassen sie sich am besten mit zustandslosen Session-Beans vergleichen, wodurch auch ihre möglichen Assoziationspartner feststehen:

- Unäre Assoziationen mit zustandslosen Session-Beans und Message-Driven-Beans mit einer maximalen Multiplizität von maximal ‚1’.
- Unäre Assoziationen mit Entity-Beans, wobei die Multiplizität beliebig, also im allgemeinsten Fall [0..*] sei kann.

2.4.4 Abhängigkeit

Das Metamodellelement Abhängigkeit (***Dependency***) beschreibt gemäß [HK99] eine allgemeine Kopplungsbeziehung zwischen zwei UML-Modellelementen.

Unter allgemeiner Kopplung ist zu verstehen, dass Änderungen in einem Element möglicherweise zu Änderungen in dem davon abhängigen Element führen können.

Im Rahmen der UML werden vier Arten von möglichen Abhängigkeiten unterschieden:

1. Eine zwischen zwei Elementen bestehende Abstraktionsabhängigkeit (***Abstraction***) verdeutlicht, dass das selbe Konzept auf zwei unterschiedlichen Abstraktionsebenen beschrieben wird.
2. Gemäß [HK99] modelliert die Bindungsabhängigkeit (***Binding***) eine Schabloneninstantiierung durch Parameterbindung.
3. Die Genehmigungsabhängigkeit (***Permission***) drückt eine Zugriffsgenehmigung zwischen zwei Elementen aus.
4. Die Benutzungsabhängigkeit (***Usage***) beschreibt schließlich, dass ein Element ein anderes für die korrekte Funktionsweise benötigt.

In Bezug auf die Komponentenmodellierung ist sowohl die Abstraktions- als auch die Benutzungsabhängigkeit von Interesse.

Die Genehmigungsabhängigkeit wäre zwar im Rahmen des Sicherheitskonzept der EJB-Spezifikation von Interesse, sie wird im Rahmen dieser Arbeit jedoch mit Hilfe von Tagged-Values, die in diesem Fall Eigenschaften von Komponenten darstellen, modelliert werden.

Die Bindungsabhängigkeit ist in diesem Zusammenhang von keinem Interesse.

2.5 Komponentenrelationsabhängige Anforderungen

Basierend auf den in Abschnitt 2.4 erarbeiteten Ergebnissen erfolgt nun die Formulierung der für die Modellierung von Komponentenbeziehungen relevanten Anforderungen.

In der folgenden Tabelle werden nochmals diejenigen Assoziationen aufgezeigt, die allgemein im Rahmen von EJB-basierten Systemen zwischen den einzelnen Komponententypen modelliert werden können und somit auch als Anforderungen an die EJB-basierte Modellierung anzusehen sind.

Die Tabelle ist dabei so strukturiert, dass immer die aus Sicht eines Clients möglichen Assoziationsformen beschrieben werden.

Die Zelle, die sich an der Position (1, 1) (Zeile, Spalte) befindet, besagt z.B., dass aus Sicht einer zustandsbehafteten Session-Bean (Client) grundsätzlich bidirektionale Assoziationen zu anderen zustandsbehafteten Session-Beans (Supplier) möglich sind. Die beiden Multiplizitäten sind in einem solchen Fall beliebig.

Tabelle 2- Assoziationen zwischen EJBs (n beliebig, fest Þ "n: 1 < n <= *)

Supplier / Client	Stateful Session Bean	Stateless Session Bean	Entity Bean	Message-Driven Bean
Stateful Session Bean	[0..n] ⟷ [0..n]	→ [0..1]	→ [0..n]	→ [0..1]
Stateless Session Bean	×	→ [0..1]	→ [0..n]	→ [0..1]
Entity Bean	×	→ [0..1]	[0..n] ⟷ [0..n]	→ [0..1]
Message-Driven Bean	×	→ [0..1]	→ [0..n]	→ [0..1]

Weitere Anforderungen bezüglich der Modellierung von Komponentenbeziehungen zwischen EJBs sind nur noch in Bezug auf die Modellierbarkeit von Abhängigkeiten zu finden.

So sollte es grundsätzlich möglich sein, Abstraktionsabhängigkeiten und Benutzungsabhängigkeiten zwischen EJBs zu modellieren.

Dies sollte aber gerade aufgrund der dort nicht vorhandenen Auswirkungen auf die spätere Codegenerierung nicht weiter schwierig umzusetzen sein.

Kapitel 3
„UML Profile for EJB“

Im Rahmen dieses Kapitels findet eine Einführung in die momentan aktuelle Version des „UML Profile for EJB“ statt.

Dazu werden zunächst allgemeine Hintergrundinformationen zur Entstehungsgeschichte des Profiles präsentiert, die auch eine Erläuterung des Entwicklungsprozesses, in den das Profile eingebettet ist, beinhalten. Als nächstes erfolgt eine Zusammenfassung der in der Präambel des Profile definierten Zielsetzung sowie der in der Profile-Spezifikation vorkommenden Definitionen. Der wesentliche Aspekt dieses Kapitels, die Einführung in die Profile-Architektur und in die dort definierten UML-Erweiterungselemente, findet in den letzten drei Abschnitten statt.

3.1 Hintergründe

Das „UML Profile for EJB“ ist eine Standardisierungsbemühung im Rahmen des „Java Community Process“ (JSR 26) mit dem Ziel, die Beschreibungssprache UML und ihre Modellelemente um die dezidierte Unterstützung der serverseitigen Komponententechnologie Enterprise JavaBeans zu erweitern.

Gemäß [WWW03] dient der „Java Community Process“ dabei folgenden Zielen:

- Förderung der Entwicklung von qualitativ hochwertigen Spezifikationen in kürzester Zeit.
- Ermöglichung der Partizipation von unabhängigem Publikum an diesem Prozess, somit auch Emanzipation von den SUN-Entwicklungsabteilungen.
- Sicherstellung von hohen Standards, z.B. durch Vorgabe von Referenzimplementierungen.

Der Standardisierungsbemühung steht ein Firmenkonsortium unter der Führung von Rational Software Corporation [WWW01] vor. An dem Firmenkonsortium sind außerdem die Unternehmen Forte Software Inc., Fujitsu Limited, IBM, IONA Technologies PLC, Open Cloud, Oracle, Sun Microsystems Inc. und Unisys beteiligt.

Seit dem 21. Juni 2001 ist die sogenannte „public review“–Phase abgeschlossen. Allerdings steht der „public draft“, also der aktuelle Entwurf der Spezifikation, der Öffentlichkeit nach wie vor zur Verfügung [WWW02] und kann somit auch als Grundlage für diese Arbeit verwendet werden.

Das Profile selbst basiert auf der UML 1.3 Spezifikation [OMG99] und führt dabei Erweiterungen bezüglich der EJB 1.1 Spezifikation [SUN99] ein.

3.2 Ziele

Die wesentliche Zielstellung, die im Rahmen der Profile-Spezifikation definiert wurde [RAT01, Seite8], lässt sich in drei Punkten zusammenfassen:

1. Definition einer vollständigen, korrekten und einfachen Repräsentationssyntax von EJB-Artefakten und ihrer Semantik für Modellierungszwecke, die eine Vereinfachung des Entwurfs von EJB-basierter Software zur Folge haben soll und damit außerdem einen, von bestimmten Modellierungswerkzeugen unabhängigen, Austausch von EJB-Entwurfsmodellen ermöglicht.
2. Abdeckung der Entwicklungs-, Installations- und Laufzeitphase des EJB-Lebenszyklus.
3. Kompatibilität mit der UML 1.3 Spezifikation, mit allen involvierten Java-APIs und –Standards sowie mit allen UML-Profiles, die im Zusammenhang mit der EJB-Technologie stehen (z.B. mit dem „UML Profile for Corba“).

3.3 Inhalt

Der Inhalt der Profile-Spezifikation beschäftigt sich neben der eigentlichen Einführung der einzelnen UML-Erweiterungselemente, wie z.B. Stereotypes, Tag-Definitions und Constraints, auch mit der Entwicklungsgeschichte dieser Erweiterungen. Im weiteren Verlauf dieser Kapitels wird jedoch nicht weiter auf die Entstehungsgeschichte eingegangen.

3.4 Grundlegende Architektur

Das „UML Profile for EJB“ führt im Wesentlichen Erweiterungselemente für die Modellierung und Beschreibung von EJB-basierter Software ein, definiert jedoch auch einige Erweiterungselemente, die zur Beschreibung der Programmiersprache Java nötig sind. Dies geschieht überall dort, wo die EJB-Erweiterungselemente direkt oder indirekt auf Java-Sprachkonstrukten aufbauen. Als Beispiel sei hier das Prinzip der Java-Schnittstellen erwähnt, die als direkte Grundlage für die EJB-Komponentenschnittstellen dienen.

Innerhalb der EJB-Erweiterungselemente wird ebenfalls eine Differenzierung eingeführt, zwischen den Elementen, die für die Beschreibung der jeweiligen EJB-Komponentenschnittstellen benötigt werden, und damit der Sicht eines potenziellen Clients genügen, und denen, die zur vollständigen Beschreibung der Komponentenimplementierung nötig sind. Die erste Gruppe wird unter dem Begriff „externe Sicht“ zusammengefasst, die zweite unter dem Begriff „interne Sicht“.

Typischerweise wird bei wiederverwendeten Komponenten, also Komponenten die z.B. von einem Dritthersteller stammen, nur die externe Sicht angegeben, da der realisierende Anteil der Komponente, z.B. die Implementierungsklasse, häufig entweder nicht verfügbar oder für das jeweilige Modell nicht relevant ist.

Die dadurch enstehende Unterteilung des Profiles wurde durch die Einführung einer Package-Struktur berücksichtigt, wobei sich alle Java-spezifischen Erweiterungselemente im Package ***java*** und alle EJB-spezifischen Erweiterungselemente im Package ***javax.ejb*** befinden:

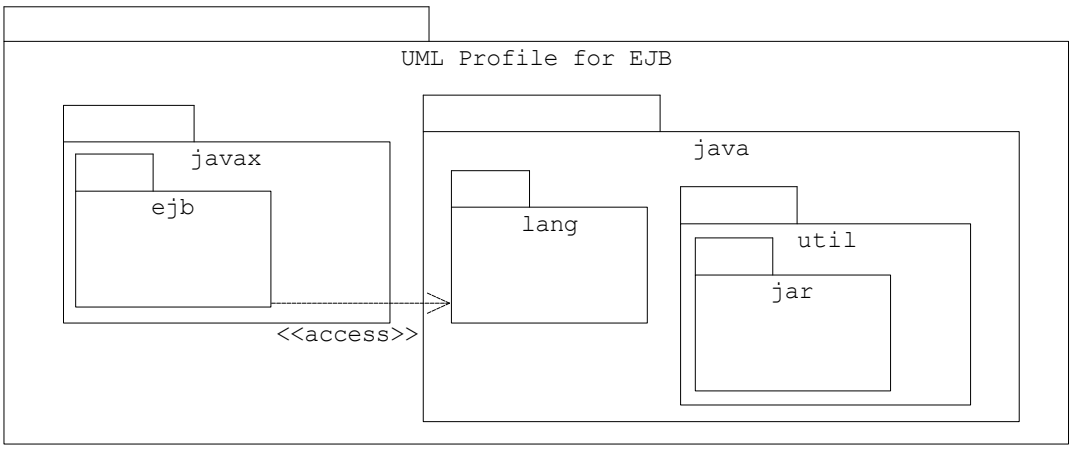


Abbildung 3 - Package-Struktur des Profiles [RAT01, Seite 55]

Des Weiteren wird gemäß der UML-Dialektik zwischen Entwurfsmodellen und Implementierungsmodell len unterschieden. Entwurfsmodelle kommen überall dort zum Einsatz, wo es um die detaillierte Modellierung und Beschreibung von Softwarelösungen geht. Entwurfsmodelle können zwar schon an Programmiersprachen gebunden sein, sind jedoch in der Regel immer noch unabhängig von bestimmten Hardwarekonfigurationen. Die Implementierungsmodell le hingegen verdeutlichen die verschiedenen Aspekte der physikalischen Implementierung, wie z.B. die verwendeten Komponenten eines Systems, deren Beziehungen untereinander sowie deren physische Verteilung.

Die Unterscheidung zwischen Entwurfs- und Implementierungsmodell len wurde innerhalb der Package-Struktur nicht explizit berücksichtigt. Es befinden sich sowohl alle dem Entwurfsmodell als auch alle dem Implementierungsmodell l zugeordneten Erweiterungselemente innerhalb des jeweils selben Packages.

Aus diesem Grund wird auch in den folgenden Abschnitten nicht mehr explizit zwischen den Erweiterungselementen des Entwurfsmodells und denen des Implementierungsmodell ls getrennt.

3.5 Java-Erweiterungselemente

Die Java-Erweiterungselemente dienen primär der Abbildung von Java-Sprachkonstrukten auf die UML. Sie bilden damit die Grundlage für die EJB-Erweiterungselemente, da diese auf der Sprache Java basieren.

Im Rahmen des „UML Profiles for EJB“ wird ein pragmatischer Ansatz im Hinblick auf die Abbildung der Java-Sprachkonstrukte verfolgt. Dabei ist es nicht Ziel dieses Profiles, die Programmiersprache vollständig auf die UML abzubilden. Vielmehr soll nur der für die Beschreibung der EJB-Komponenten notwendige Teil umgesetzt werden.

Da es nicht Ziel dieser Arbeit ist, die Java-Erweiterungselemente zu untersuchen, wird hier auf eine detaillierte Beschreibung verzichtet.

3.6 EJB-Erweiterungselemente

Die EJB-Erweiterungselemente bilden den eigentlichen Kern des Profiles. Sie dienen der detaillierten Beschreibung von EJB-Bestandteilen und unterteilen sich in Elemente, die die externe Sicht beschreiben, und in Elemente, die die interne Sicht beschreiben.

Im folgenden werden nun die einzelnen Erweiterungselemente vorgestellt, wobei die Unterscheidung zwischen externer und interner Sicht beibehalten wird:

3.6.1 Externe Sicht

3.6.1.1 EJB-Methoden

EJB-Methoden werden innerhalb der beiden Komponentenschnittstellen einer EJB deklariert und dabei innerhalb der UML grundsätzlich auf das Metamodellelement ***Operation*** abgebildet. Bei der Einführung von Stereotypes durch das Profile wird dabei zwischen drei Methodentypen unterschieden:

1. Remote-Methoden werden innerhalb der Remote-Schnittstelle der EJB deklariert und mit dem Stereotyp «EJBRemoteMethod» gekennzeichnet.
2. Create-Methoden dienen der Erstellung einzelner Komponenteninstanzen und werden innerhalb der Home-Schnittstelle der EJB deklariert. Zur Kennzeichnung dieser Methoden führt das Profile das Stereotype «EJBCreateMethod» ein.
3. Find-Methoden dienen dem Auffinden existierender Komponenteninstanzen und werden innerhalb der Home-Schnittstelle der Entity-Bean deklariert. Zur Kennzeichnung dieser Methoden führt das Profile den Stereotype «EJBFinderMethod» ein.

3.6.1.2 Remote-Schnittstelle

Die Remote-Schnittstelle stellt die Geschäftslogik der jeweiligen EJB dar. Sie definiert eine Menge von Remote-Methoden (siehe 3.6.1.1). Einzelne Remote-Schnittstellen werden innerhalb der UML auf das Metamodellelement ***Class*** abgebildet. Zusätzlich wird dieses Element mit dem Stereotype «EJBRemoteInterface» gekennzeichnet, dass das Standardstereotype «type» spezialisiert.

3.6.1.3 Home-Schnittstelle

Die Home-Schnittstelle ermöglicht den Zugriff auf einzelne oder mehrere EJB-Instanzen. Sie stellt abhängig vom EJB-Typ eine Menge von Create-Methoden und Find-Methoden bereit.

Find-Methoden können nur innerhalb von Entity-Beans definiert werden. Grundsätzlich werden einzelne Home-Schnittstellen auf das UML-Metamodellelement ***Class*** abgebildet. Abhängig vom jeweiligen Bean-Typ, ist diese Klasse noch mit einem der folgenden Stereotypes zu versehen:

1. Das Stereotype «EJBSessionHomeInterface» kennzeichnet die Home-Schnittstelle von Session-Beans. Um die Unterscheidung zwischen zustandsbehafteten und zustandslosen Session-Beans zu ermöglichen, wurde das Tagged-Value „EJBSessionType“ eingeführt (siehe Anhang A).
2. Das Stereotype «EJBEntityHomeInterface» kennzeichnet die Home-Schnittstelle von Entity-Beans.

Beide Stereotypes spezialisieren ihrerseits das abstrakte Stereotype «EJBHomeInterface», das die gemeinsamen Eigenschaften von Home-Schnittstellen bündelt.

3.6.1.4 Primärschlüsselklasse

Die Primärschlüsselklasse dient der eindeutigen Kennzeichnung verschiedener Entity-Bean-Instanzen. Im Rahmen der externen Sicht wird die Primärschlüsselklasse innerhalb der UML auf eine Abhängigkeit (***Usage***) abgebildet, die mit dem Stereotype «EJBPrimaryClass» versehen ist. Der Client dieser Abhängigkeit ist die jeweilige Home-Schnittstelle der Entity-Bean, der Supplier ist die für den Primärschlüsseltyp verwendete Klasse.

3.6.1.5 EJB-Jar

Das EJB-Jar, das zum Implementierungsmodell gehört, wird innerhalb der UML auf das Metamodellelement ***Package*** abgebildet. Dieses ist dann mit dem Stereotype «EJB-JAR» zu kennzeichnen, das seinerseits das Stereotype «JavaArchiveFile» spezialisiert (siehe Anhang A).

Soll ein expliziter Verweis auf ein EJB-Client-Jar aus Sicht eines EJB-Jars beschrieben werden, so wird dafür eine Abhängigkeit (***Usage***) verwendet, die das Stereotype «EJBClientJar» trägt. Der Supplier dieser Abhängigkeit ist dann das jeweilige EJB-Client-Jar.

3.6.1.6 Deployment-Descriptor

Der Deployment-Descriptor, der ebenfalls zum Implementierungsmodell gehört, wird innerhalb der UML auf das Metamodellelement ***Component*** (UML 1.3) abgebildet. Dieses ist mit dem Stereotype «EJBDescriptor» zu kennzeichnen, das seinerseits das Standardstereotype «file» spezialisiert.

3.6.2 Interne Sicht

3.6.2.1 EJB-Methoden (Ergänzung)

Ergänzend zu denen für EJB-Methoden in der externen Sicht festgelegten Erweiterungselementen, sind innerhalb der internen Sicht zunächst die jeweiligen Gegenstücke innerhalb der Implementierungsklasse (siehe Abschnitt 3.6.2.4) vorgesehen. Diese werden mittels des UML-Metamodellelements ***Operation*** modelliert. Um die möglicherweise vorgesehenen Transaktionsattribute und Sicherheitsrollen, die an einzelne EJB-Methoden geknüpft werden können, berücksichtigen zu können, wurden die beiden Tagged-Values „EJBRoleNames“ und „EJBTransAttribute“ eingeführt (siehe Anhang A). Beide sind innerhalb der Implementierungsklasse der jeweiligen Methode hinzuzufügen.

3.6.2.2 Remote-Schnittstelle (Ergänzung)

Im Rahmen der internen Sicht gilt es die Verknüpfung zwischen der Implementierungsklasse einer EJB und der verwendeten Remote-Schnittstelle herzustellen. Dieses geschieht mittels einer Abhängigkeit (***Abstraction***), die das Stereotype «EJBRealizeRemote» trägt. Der Supplier dieser Abhängigkeit stellt dementsprechend die Remote-Schnittstelle dar, der Client ist die jeweilige Implementierungsklasse.

3.6.2.3 Home-Schnittstelle (Ergänzung)

Ergänzend zu der Verknüpfung zwischen der Implementierungsklasse und der zugehörigen Remote-Schnittstelle sieht die EJB-Spezifikation innerhalb des jeweiligen Deployment-Descriptors auch eine Verknüpfung zur verwendeten Home-Schnittstelle vor. Diese wird ebenfalls mittels einer Abhängigkeit (***Abstraction***) modelliert, die das Stereotype «EJBRealizeHome» trägt.

3.6.2.4 Implementierungsklasse

Eine EJB umfasst neben den beiden Schnittstellen auch einen realisierenden Teil, die Implementierungsklasse. Diese wird innerhalb der UML mit Hilfe des Metamodellelements ***Class*** abgebildet, das das Stereotype «EJBImplementation» trägt.

3.6.2.5 Enterprise JavaBean (EJB)

Die EJB an sich (im Sinne einer Komponentenspezifikation) wird mit Hilfe des Metamodellelements ***Subsystem*** auf die UML abgebildet. Dieses Element dient gewissermaßen als Container für die Schnittstellen, die Implementierungsklasse und gegebenenfalls den Primärschlüssel sowie weiteren Hilfsklassen. Generell wurde zur Kennzeichnung eines solchen Subsystems das abstrakte Stereotype «EJBEnterpriseBean» eingeführt.

Eine EJB umfasst neben den zuvor erwähnten Bestandteilen auch bestimmte Eigenschaften. Dazu gehören der eindeutige Jar-Bezeichner, eventuelle Umgebungsvariablen („environment entries“), eventuelle EJB-Referenzen, eventuell verwendete Ressourcen-Factories sowie die verwendeten Sicherheitsrollen. Diese Eigenschaften können mit Hilfe der Tagged-Values „EJBNameInJAR“, „EJBEnvEntries“, „EJBReferences“, „EJBResources“ sowie „EJBSecurityRoles“ beschrieben werden (siehe Anhang A).

Da aber zwei unterschiedliche EJB-Typen innerhalb der EJB 1.1-Spezifikation verwendet werden, erfolgt innerhalb des Profiles eine weitere Spezialisierung des Stereotypes «EJBEnterpriseBean»:

1. Session-Bean-Spezifikationen werden mit Hilfe des Stereotypes «EJBSessionBean» gekennzeichnet. Mit Hilfe des Tagged-Values „EJBTransType“ kann in diesem Fall

zusätzlich festgelegt werden, ob die Transaktionsverwaltung in den Händen der jeweiligen Session-Bean oder des Containers liegt.

2. Entity-Bean-Spezifikationen werden mit Hilfe des Stereotypes «EJBEntityBean» gekennzeichnet. Um innerhalb der Entity-Bean persistente Java-Felder zu kennzeichnen, die vom Container verwaltet werden, muss dem jeweiligen ***Attribut*** innerhalb der Implementierungsklasse das Stereotype «EJBCmpField» zugewiesen werden. Dasjenige Java-Feld der Implementierungsklasse von Entity-Beans, das den Primärschlüssel beinhaltet, wird mit dem Stereotype «EJBPrimaryKeyField» gekennzeichnet.

3.6.2.6 EJB-Primärschlüsselklasse (Ergänzung)

Im Rahmen der internen Sicht ist das UML-Element, das die Primärschlüsselklasse repräsentiert, innerhalb des Spezifikationsabschnitts des jeweiligen Subsystems zu platzieren oder zu importieren.

Kapitel 4
Bewertung des Profiles

Bevor die eigentliche Erweiterung des Profiles (im Sinne der Aufgabenstellung) vorgenommen werden kann, erfolgt im Rahmen dieses Kapitels zunächst eine Bewertung der ursprünglichen Fassung. Das wesentliche Ergebnis dieser Bewertung wird die Formulierung von Anforderungen sein, die die Grundlage der später vorzunehmenden Erweiterung bilden.

Die Bewertung erfolgt anhand der vier folgenden Kriterien:

1. Umsetzung der in der Profile-Spezifikation definierten Zielstellung.
2. Semantisch korrekte Nutzung des UML-Metamodells.
3. Die zur Erstellung des Profiles verwendete Strategie.
4. Erfüllung der im zweiten Kapitel definierten Anforderungen bezüglich der Komponentenmodellierung.

Am Ende des Kapitels erfolgt schließlich eine Zusammenfassung der einzelnen Bewertungsergebnisse und die schon angesprochene Formulierung der für die Erweiterung grundlegenden Anforderungen.

4.1 Umsetzung der Zielstellung

Hinweis:
Die Bewertung des Profiles, anhand der in der Spezifikation definierten Zielstellung, ist unter der Voraussetzung, dass die Profile-Spezifikation noch keinen finalen Zustand erreicht hat, prinzipiell als vorläufig anzusehen.

So ist zunächst der Anspruch, eine vollständige, korrekte und einfache Repräsentationssyntax von EJB-Artefakten und ihrer Semantik für Modellierungszwecke zu definieren, bezogen auf die ersten beiden Prädikate – Vollständigkeit und Korrektheit - zu verneinen.

Zur Erfüllung dieser Prädikate fehlt hauptsächlich die im Rahmen der UML-Spezifikation vorgesehene Definition von Constraints (Stichwort: „Well-Formedness rules“); daher kann von ‚Vollständigkeit’ auch keine Rede sein.

Des Weiteren existieren eine Fülle von Inkonsistenzen bzw. syntaktische und semantische Fehler innerhalb der Profile-Spezifikation, die das Prädikat ‚korrekt’ ebenfalls ausschließen.

Beispielhaft sei an dieser Stelle der Hinweis, dass es zwei unterschiedliche und sich dabei widersprechende Definitionen des Stereotypes «EJBReference» gibt. Die erste Definition ordnet diesem Stereotype die Base-Class ***Association*** zu [RAT01, Seite 14], die zweite spricht hingegen von ***Usage*** als Base-Class [RAT01, Seite 58].

Die Überprüfung des Prädikats ‚Einfachheit’ ist hingegen ohne einen Vergleich mit anderen, konkurrierenden Profiles bzw. einem einschlägigen Erfahrungsbericht von möglichen Nutzergruppen an dieser Stelle nicht durchführbar.

Die Überprüfung der in der Profile-Spezifikation angegebene Zielsetzung, dass die modellierungswerkzeugunabhängige Austauschbarkeit von EJB-Entwurfsmodellen zu erreichen ist, ist ohne die dafür nötige Werkzeugunterstützung nicht möglich. Dafür werden Werkzeuge benötigt, die über eine UML-konforme Profileunterstützung verfügen und die darüber hinaus das Austauschformat XMI vollständig (bezogen auf das Metamodell) unterstützen.

Die Überprüfung der hier noch nicht erwähnten Ziele macht erst nach der Freigabe der Final-Release-Version der Spezifikation einen Sinn.

Zwischenfazit:
Somit steht fest, dass die in der Profile-Spezifikation festgelegte Zielsetzung in einigen Punkten nicht erreicht wurde bzw. in den verbliebenen Punkten aufgrund des vorläufigen Charakters der Spezifikation nicht verifizierbar ist.

4.2 Nutzung des UML-Metamodells

Bei der Überprüfung der semantisch korrekten Nutzung des ULM-Metamodells durch das Profile geht es um die prinzipielle Frage, ob das Basiselement des Profiles, das ***Subsystem***,

das für die Darstellung der EJBs (im Sinne einer Komponentenspezifikation) zuständig ist, grundsätzlich richtig ausgewählt wurde.

Eine falsche Auswahl würde die Anwendbarkeit des Profiles, im Hinblick auf die semantisch korrekte Modellierung von EJB-basierten Systemen sowie im Hinblick auf die partielle Codeerzeugung erschweren, da die Abbildung des jeweiligen Modells nur mit Inkaufnahme von Kompromissen möglich wäre.

Außerdem wäre die richtige Auswahl auch ein gewisser Garant für die Beständigkeit des Profiles hinsichtlich zukünftiger Versionen der UML, da gerade in Bezug auf die Komponentenmodellierung eine Überarbeitung des Metamodells zu erwarten ist [Kobr99, Seite 35, OMG00, Seite 24]. Diese Überarbeitung wird aber wahrscheinlich nur Rücksicht auf die bisher übliche Form der Komponentenmodellierung nehmen, indem sie bei dem betreffenden Metamodellelement eine eventuelle Abwärtskompatibilität bewahrt.

Um eine Überprüfung der semantisch korrekten Nutzung des UML-Metamodells durchführen zu können, muss zunächst einmal festgelegt werden, welches Metamodellelement überhaupt den Ausgangspunkt dieser Diskussion bildet.

Den Ausführungen von [OMG01, Seite „2-28“] folgend („A classifier is an element that describes behavioral and structural features; [...]“), stünde demnach das Metamodellelement ***Classifier*** als Ausgangspunkt fest, da es über alle die, bezüglich der Komponentenmodellierung in Kapitel 2 geforderten, Eigenschaften verfügt.

Da dieses Element jedoch als abstrakt deklariert ist, müssen alle von ***Classifier*** erbenden Elemente, die nicht als abstrakt deklariert sind, als mögliche Kandidaten für die Darstellung von Komponenten in Betracht gezogen werden.

Dies sind die Elemente ***Actor***, ***Artifact***, ***Class***, ***Component***, ***DataType***, ***Interface***, ***Signal***, ***Subsystem*** und ***UseCase***.

Nach einem ausführlichen Studium (das an dieser Stelle nicht weiter vertieft werden soll) der diesen Elementen im Rahmen der Spezifikation zugeordneten Rollen, bleiben die Elemente ***Class***, ***Component*** und ***Subsystem*** als mögliche Darstellungsmittel von Komponenten übrig.

Nun gilt es nur noch, anhand dafür geeigneter Kriterien, den für die Komponentenmodellierung geeignetesten Kandidaten zu bestimmen.

Dazu wird die in [Kobr00, Seiten 33-34] erfolgte Untersuchung (samt der ihr eigenen Kriterien) übernommen, und im Rahmen der nachfolgenden Tabelle dargestellt.

Es sollte noch erwähnt werden, dass einerseits die Vererbungsrelation zusätzlich Berücksichtigung im Kriterienkatalog fand und andererseits eine Anpassung der Ergebnisse an die momentan geltende UML-1.4-Spezifikation erfolgte.

Tabelle 3 - Kriterien für die Verwendung als Komponentendarstellung

Criterials	*Component*	*Subsystem*	*Class*
Is a classifier?	✓	✓	✓
Can have operations?	✓	✓	✓
Can have methods?	×	×	✓
Can have attributes?	✓	×	✓
Can have interfaces?	✓	✓	✓
Can be associated?	✓	✓	✓
Can be extended?	✓	✓	✓
Can have thread of control?	×	×	✓
Can be nested?	✓	✓	✓
Is a grouping construct?	✓	✓	✓
Can import/access?	×	✓	×
Represents a unit in a physical system?	✓	✓	×
Contains implementation of model elements?	✓	×	×
Can create instances?	✓	✓	✓
Instances typically reside on nodes?	✓	×	×
Tend to be coarse-grained?	✓	✓	×
Typically modeled during analysis?	×	✓	✓
Typically modeled during design?	✓[1]	✓	✓
Typically modeled during implementation?	✓	×	×

Bei der abschließenden Auswahl des zur Darstellung von Komponenten geeignetesten Elements, werden nur die kontroversen Punkte, also diejenigen Eigenschaften, die nicht für alle drei Metamodellelemente gleichermaßen gelten, diskutiert und damit für die Ergebnisfindung berücksichtigt:

- Die fehlende Möglichkeit, den beiden Metamodellelementen ***Component*** und ***Subsystem*** Methoden zuordnen zu können, fällt nicht weiter ins Gewicht, da es sich im konkreten Fall um die Abbildung der EJB-Komponentenspezifikation und nicht um die der Implementierungsklasse handelt und somit auf dieser Ebene kein Bedarf für Methoden besteht.
- Das Fehlen von Attributen bei dem Metamodellelement ***Subsystem*** kann z.B. durch die Anwendung von Tagged-Values kompensiert werden.

[1] Ist in der UML-1.4-Spezifikation ausdrücklich vorgesehen [OMG01, Seite „2-31“].

- Auch die Eigenschaft „Can have thread of control?“ ist für die EJB-Komponentenspezifikation nicht relevant und somit fällt das Fehlen dieser Eigenschaft bei den Metamodellelementen ***Component*** und ***Subsystem*** nicht ins Gewicht.
- Die fehlende Möglichkeit, andere Elemente mittels der Anweisungen ***import*** bzw. ***access*** in den eigenen Namensraum einzubinden, ist für die beiden Metamodellelemente ***Component*** und ***Class*** im Vergleich zum Metamodellelement ***Subsystem*** der entscheidende Nachteil. So wird im Rahmen der EJB-Spezifikation [SUN01] die gemeinsame Nutzung von Home- und Component-Schnittstellen durch unterschiedliche EJBs nicht ausgeschlossen. Dies gilt auch für die gemeinsame Nutzung von Implementierungsklassen. Diese Semantik ist durch das Fehlen von ***import***- bzw. ***access***-Anweisungen nur schwer anderweitig umzusetzen.
- Die fehlenden Eigenschaften „Represents a unit in a physical system?“, „Contains implementation of model elements?“, „Instances typically reside on nodes?“ und „Typically modeled during implementation?“ fallen bei den betreffenden Metamodellelementen nicht so stark ins Gewicht, da es hierbei primär um die Darstellungsmöglichkeit innerhalb von Entwurfsmodellen geht. Die aufgezählten Eigenschaften sind aber eher bei Implementierungsmodell len von Belang.
- Dass das Metamodellelement ***Component*** nicht im Rahmen der Analysephase eingesetzt wird, spielt hier ebenfalls keine große Rolle.
- Die eher als fein anzusetzende Granularität des Metamodellelements ***Class*** (fehlende Eigenschaft „Tend to be coarse-grained?“) deutet jedoch eher auf eine Nichteignung für die Komponentenmodellierung hin.

Basierend auf der Gegenüberstellung der den einzelnen Metamodellelementen jeweils zuzuordnenden Eigenschaften, ist das ***Subsystem***, als das wohl geeigneteste Element zur Darstellung von Komponenten anzusehen.

Zwischenfazit:

Als Ergebnis der Bewertung der semantisch korrekten Nutzung des UML-Metamodellelements steht fest, dass die Wahl des Elements ***Subsystem*** zur Darstellung von EJBs richtig war.

4.3 Verwendete Profile-Erstellungsstrategie

Grundsätzlich existieren für die Erstellung von technologiespezifischen Profiles zwei unterschiedliche Ansätze (Strategien):

1. Der Bottom-Up-Ansatz.
2. Der Top-Down-Ansatz.

Bevor die Analyse der zur Erstellung des Profiles verwendeten Strategie erfolgt, werden zunächst die beiden unterschiedlichen Ansätze erläutert und im Anschluß daran verglichen.

4.3.1 Top-Down-Ansatz

Der Top-Down-Ansatz zur Erstellung von technologiespezifischen Profiles ist hauptsächlich dadurch gekennzeichnet, dass er unter dem Einsatz einer möglichst geringen Anzahl von Erweiterungselementen (Stereotypes und Tag-Definitions) semantisch eindeutige Modellbeschreibungen ermöglicht.

Die dann auf Basis eines solchen Profile entstandenen Modelle sind hauptsächlich durch ein sehr hohes Abstraktionsniveau im Vergleich zu ihrer späteren Implementierung (Codeabbild) gekennzeichnet. Dieses hohe Abstraktionsniveau schlägt sich natürlich auch in der hohen Komplexität der für die Codegenerierung notwendigen Abbildungsregeln nieder.

Der Top-Down-Ansatz ist darüber hinaus aufgrund der eher niedrigeren Modellkomplexität auch durch eine, im Vergleich zum Bottom-Up-Ansatz, intuitivere Nutzbarkeit gekennzeichnet.

4.3.2 Bottom-Up-Ansatz

Im Gegensatz zum Top-Down-Ansatz ist der Bottom-Up-Ansatz, dessen Ziel die möglichst vollständige Abbildung aller technologiespezifischen Details mittels UML-Erweiterungselementen ist, durch ein äußerst geringes Abstraktionsniveau seiner Modelle gekennzeichnet.

Dadurch wird zwar die Formulierung sehr einfacher Abbildungsvorschriften ermöglicht, jedoch nur durch Inkaufnahme des Nachteils, dass die dann enstehenden Modelle sehr komplex sind.

Es ist daher fragwürdig, ob eine vernünftige Handhabbarkeit solcher Profiles überhaupt möglich ist. Ein weiterer Nachteil dieses Ansatzes ist das Vorhandensein von Redundanzen in den Modellen, da insbesondere die dort verwendeten Stereotypes keine zusätzliche Semantik kapseln.

4.3.3 Vergleich beider Ansätze

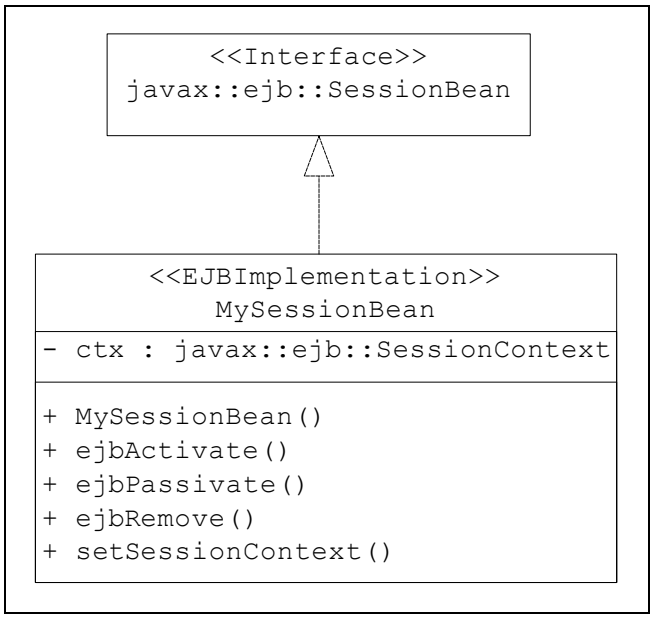
Das folgende einfaches Beispiel soll den jeweils ansatzspezifischen Modellierungsaufwand demonstrieren.

Es gilt das folgende Codefragment mittels eines Modells zu beschreiben:

```
public class MySessionBean implements javax.ejb.SessionBean {

    private javax.ejb.SessionContext ctx;

    public MySessionBean() {
    }

    public void ejbActivate() throws javax.ejb.EJBException {
    }

    public void ejbPassivate() throws javax.ejb.EJBException {
    }

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

    public void setSessionContext(javax.ejb.SessionContext ctx)
            throws javax.ejb.EJBException {

        this.ctx = ctx;
    }

   ...
}
```

Im Rahmen des Top-Down-Ansatzes sähe das dafür notwendige Modell wie folgt aus:

Im Gegensatz dazu, müsste bei der Verwendung des Bottom-Up-Ansatzes das Modell hingegen so beschrieben werden:



Abschließend werden in der folgenden Tabelle nochmals die jeweiligen Vor- und Nachteile der beiden Profile-Erstellungsstrategien zusammengefasst:

Tabelle 4 - Vor- und Nachteile von Profile-Erstellungsstrategien

Kriterium	Top-Down	Bottom-Up
Anzahl der Erweiterungselemente (Komplexität)	Gering	Hoch
Abstraktionsniveau der Modelle	Hoch	Gering
Grad der Redundanz in den Modellen	Gering	Hoch
Intuitive Nutzbarkeit des Profiles	Hoch	Gering
Komplexität der Abbildungsvorschriften (Generator)	Hoch	Gering

4.3.4 Analyse der in der Profile-Spezifikation verwendeten Strategie

Bei einem genauen Studium der Profile-Spezifikation lassen sich zunächst relativ eindeutige Hinweise auf die Verwendung des Bottom-Up-Ansatzes zur Erstellung der Profile-Architektur finden.

Das wird z.B. anhand des folgenden Zitats bezüglich der Erstellung der Java-Erweiterungselemente deutlich: „This section defines a mapping from Java source files to UML model elements.“ [RAT01, Seite 18]. Das Wort „mapping“, dass in diesem Zusammenhang als Abbildung verstanden werden kann, impliziert eine dem Reverse-Engineering gleichzusetzende Erstelllungsstrategie – also den Bottom-Up-Ansatz.

Ebenso deutlich kann das auch für die EJB-Erweiterungselemente nachgewiesen werden: „It (dieser Abschnitt, der Autor) enumerates the logical constructs defined by the EJB Specification, and for each construct defines a mapping to UML model elements, [...].“ [RAT01, Seite 28].

Am deutlichsten zeigt sich die Verwendung des Bottom-Up-Ansatzes bei der Abbildung von EJB-Referenzen innerhalb der Profile-Architektur. Einerseits sind solche Referenzen mittels einer, mit dem Stereotype «EJBReference» gekennzeichneten, Assoziation zwischen zwei EJBs abzubilden, andererseits ist auch das entsprechende Tagged-Value «EJBReferences» mit den entsprechenden Werten zu belegen.

An dieser Stelle wurden sowohl die entsprechenden EJB-Codefragmente als auch die Struktur des Deployment-Descriptors als Grundlage für die Erstellung des Profiles verwendet, wodurch es dann zu dieser Redundanz (als wesentliches Merkmal des Bottom-Up-Ansatzes) gekommen ist.

Allerdings ist bei einer genaueren Analyse der Profile-Architektur (siehe Kapitel 3) an anderer Stelle auch ein höheres Abstraktionsniveau festzustellen.

Bei der exakten Abbildung aller der in der EJB-Spezifikation erwähnten Bestandteile, hätte z.B. auch die `ejbCreate`-Methode als Ausgangspunkt für ein Stereotype berücksichtigt werden müssen. Da dies aber in diesem und auch in anderen Fällen nicht geschehen ist, kann nicht von einem reinen Bottom-Up-Ansatz bei der Erstellung des Profiles gesprochen werden.

Zwischenfazit:
Bei der Erstellung der Profile-Architektur ist keiner der beiden Ansätze (Top-Down bzw. Bottom-Up) durchgehend verfolgt worden.

4.4 Anforderungsüberprüfung

Abschließend gilt es nun das Profile in Bezug auf die Erfüllung der im zweiten Kapitel postulierten Anforderungen an die Komponentenmodellierung zu überprüfen.

4.4.1 EJB-Komponentenstruktur

Die Abbildung der EJB-Komponentenstruktur erfolgt, bezogen auf die EJB-2.0-Spezifikation, nur teilweise.

So fehlt z.B. die korrekte Abbildung der jeweiligen Home- und Component-Schnittstellen. Auch die Neuerungen bezüglich der duch den Deployment-Descriptor beschriebenen EJB-Komponentenspezifikation werden im Rahmen dieser Profile-Architektur nicht

berücksichtigt. Gleiches gilt auch für die implementierungsklassenbetreffenden Erweiterungen.

Zwischenfazit:
Die Abbildung der EJB-Komponentenstruktur erfolgt, bezogen auf die EJB-2.0-Spezifikation, nur mangelhaft.

4.4.2 EJB-Komponentenbeziehungen

Die Berücksichtigung der möglichen EJB-Komponentenbeziehungen erfolgt im Profile gerade im Hinblick auf die Codegenerierung ebenfalls nur mangelhaft.

EJB-Referenzen werden zwar im Rahmen der Erweiterungselemente berücksichtigt (siehe Abschnitt 4.3.4), eine Formulierung von in diesem Fall anzuwendenden Codeabbildungsregel erfolgt jedoch nicht.

Persistene Assoziationen zwischen Entity-Beans werden unter dem Hinweis der Seitens der EJB-1.1-Spezifikation mangelnden Unterstützung des Erstellens, Löschens und Verwaltens von solchen Beziehungen nicht unterstützt [RAT01, Seite 63].

Diese Feststelllung gilt in dieser Tragweite jedoch nicht für die EJB-2.0-Spezifikation, die das Konzept der Container-Managed-Relationships eingeführt hat (siehe dazu Kapitel 6).

Zwischenfazit:
Gerade der für die Komponentenmodellierung besonders interessante Bereich der Komponentenbeziehungen ist im Rahmen des „UML Profile for EJB“ nur mangelhaft umgesetzt.

4.5 Zusammenfassung

Basierend auf den Zwischenergebnissen der vorherigen Abschnitte ergibt sich folgende Zusammenfassung:

- Zunächst einmal ist festzustellen, dass das Basiselement des Profiles, das ***Subsystem***, zur Beschreibung von EJB-Komponentenspezifikationen korrekt gewählt wurde.
- Durch den vorläufigen Charakter des Profiles ist es jedoch nicht gelungen, die von der Profile-Spezifikation gesetzte Zielsetzung vollständig umzusetzen.

- Durch die Nichteinbeziehung der EJB-2.0-Spezifikation erfüllt das Profile sowohl die komponentenstrukturabhängigen- als auch die komponentenbeziehungsabhängigen Anforderungen an die Komponentenmodellierung nur mangelhaft.
- Auch die bisher existierende Profile-Architektur erscheint aufgrund der nicht durchgehenden Verfolgung einer Erstellungsstrategie als optimierungsbedürftig.

Basierend auf diesem Gesamtergebnis erscheint eine grundlegende Übearbeitung des Profiles als unumgänglich. Der Charakter der einzelnen Mängel erlaubt jedoch eine weitere Differenzierung dieses Prozesses in zwei voneinander unabhängige Teilschritte:

1. Optimierung der Profile-Architektur.
2. Erweiterung des Profiles um die EJB-2.0-Neuerungen.

4.5.1 Anforderungen an die Optimierung des Profiles

Das Ergebnis dieser Profile-Optimierung muss folgenden Anforderungen genügen:

- Korrekte Anwendung des duch die aktuelle UML-1.4-Spezifikation vorgegebenen (light-weight) Metamodell-Erweiterungsmechanismus.
- Durchgehende Nutzung einer Profile-Erstellungsstrategie – in diesem Fall des Top-Down-Ansatzes und der damit einhergehenden Redundanzverminderung.

4.5.2 Anforderungen an die Erweiterung des Profiles

Die Erweiterung des Profiles um die Neuerungen der EJB-2.0-Spezifikation muss am Ende folgenden Anforderungen genügen:

- Vollständige Berücksichtigung der im zweiten Kapitel formulierten komponentenstrukturabhängigen Anforderungen.
- Vollständige Berücksichtigung der ebenfalls im zweiten Kapitel formulierten komponentenbeziehungsabhängigen Anforderungen.
- Vorhandensein einer vollständigen Profile-Spezifikation gemäß der in der UML-Spezifikation empfohlenen Art und Weise, d.h. Definition aller Stereotypes, Tag-Definitions und Constraints („Well-Formedness rules“), die für die eindeutige Beschreibung von EJB-basierten Komponentenmodellen benötigt werden.

Kapitel 5
Optimierung des Profiles

Dieses Kapitel hat die Umsetzung und anschließende Bewertung der im vorhergehenden Kapitel erarbeiteten Optimierungsanforderungen zum Gegenstand.

Dabei erfolgt die Optimierung der usprünglichen Fassung des „UML Profiles for EJB“ in zwei Teilschritten:

1. Herstellung der UML-1.4-Konformität.
2. Optimierung der Profile-Architektur im Sinne eines nachträglich konsequent angewandten „Top-Down-Ansatzes“.

Im Anschluß der Optimierung findet eine Bewertung des Ergebnisses anhand einer, die Anzahl der EJB-bezogenen Erweiterungselemente betreffenden, Statistik statt.

Hinweis:
Die Optimierung des Profiles betrifft sowohl die Java-Erweiterung als auch die EJB-Erweiterung. Da der Schwerpunkt dieser Arbeit jedoch auf der EJB-Technologie liegt, wird im Folgenden nur auf die Optimierung des EJB-spezifischen Teils des Profiles eingegangen. Die Optimierungsergebnisse des Java-spezifischen Teils sind dem Anhang B zu entnehmen.

5.1 Herstellung der UML-1.4-Konformität

Die Herstellung der UML-1.4-Konformität erfolgt durch eine Typisierung der bisher untypisierten Tagged-Values mittels sogenannter Tag-Definitions. Im Anschluss werden die Tag-Definitions eindeutig einem Stereotype zugeordnet, da einige der bisher definierten Tagged-Values direkt UML-Metamodellelementen zugeordnet waren.

Außerdem gilt es in Einzelfällen, die sich veränderte UML-Metamodellarchitektur zu berücksichtigen.

5.1.1 Einführung und Zuordnung von Tag-Definitions

Die Einführung und Zuordnung der angesprochenen Tag-Definitions orientiert sich an den ursprünglich definierten Tagged-Values.

Dort, wo eine gültige Zuordnung von Tagged-Values zu Stereotypen bestand, wird sie übernommen. In den Fällen, wo eine solche Zuordnung bisher nicht existierte, wird zunächst geprüft, ob sie im Rahmen der schon definierten Stereotypes nachträglich erfolgen kann. Falls keine geeigneten Stereotypes für eine solche Zuordnung vorhanden sind, müssen sie neu eingeführt werden.

Für die beiden Tagged-Values „EJBRoleNames“ und „EJBTransAttribute“ existierte bisher keine Zuordnung zu einem Stereotype. Im Rahmen der Einführung von Tag-Definitions werden sie daher beide dem Stereotype «EJBRemoteMethod» zugeordnet, der selber aus dem früheren Stereotype «EJBMethod» hervorgegangen ist (siehe Abschnitt 5.2.4). Diese Zuordnung ist insofern semantisch korrekt, da beide Tag-Definitions laut EJB-Spezifikation direkte Eigenschaften von Schnittstellenmethoden darstellen und letztere durch das Stereotype «EJBRemoteMethod» im Profiles eindeutig gekennzeichnet werden. Da die Tag-Definition „EJBRoleNames“ zur Beschreibung aller für einen Methodenaufruf zugelassenen Sicherheitsrollen dient, ist sie vom Typ String (Multiplizität: [0..*]). Die Tag-Definition „EJBTransAttribute“ beschreibt das der jeweiligen Methode zugeordnete Transaktionsattribut und ist daher vom Typ String.

Die Zuordnung der beiden Tagged-Values „EJBEnvEntries“ und „EJBSecurityRoles“ zum Stereotype «EJBEnterpriseBean» wurde bei der Einführung der dafür vorgesehenen Tag-Definitions übernommen. Da beide Tag-Definitions jeweils eine Menge von Eigenschaften beschreiben (im ersten Fall die von der EJB verwendeten Umgebungseinträge; im zweiten Fall die dort ebenfalls definierten Sicherheitsrollen), sind beide vom Typ String (Multiplizität: [0..*]).

Das Tagged-Value „EJBResources“ wird als umbenannte Tag-Definition „EJBResourceRefs“ samt der bisherigen Zuordnung zu «EJBEnterpriseBean» übernommen. Ihr wird der Typ String (Multiplizität: [0..*]) zugeordnet, da sie die Menge aller von der EJB verwendeten Ressourcen beschreibt.

Die beiden Tagged-Values „EJBNameInJAR“ und „EJBReferences“ werden im Rahmen des Optimierungsprozesses hingegen nicht als Tag-Definitions übernommen (siehe Abschnitt 5.2.2).

Die beiden Tagged-Values „EJBReentrant“ und „EJBPersistenceType“ werden samt der Zuordnung zum Stereotype «EJBEntityBean» bei der Einführung der entsprechenden Tag-Definitions übernommen. Dabei ist die Tag-Definition „EJBReentrant“ gemäß der EJB-Spezifikation vom Typ Boolean und „EJBPersistenceType“ vom Typ String, da sie eine Aussage über die jeweils geltende Persistenzverwaltung („Bean“ oder „Container“) macht.

Die Zuordnung des Tagged-Values „EJBSessionType“ zum Stereotype «EJBSessionHomeInterface» wird bei der Einführung der gleichnamigen Tag-Definition nicht übernommen. Dieser Tag-Definition wird hingegen das Stereotype «EJBSessionBean» zugeordnet, da die Festlegung, ob die betreffende Session-Bean über einen Zustand verfügt („Stateful“ versus „Stateless“), zur EJB-Spezifikation und nicht zur Home-Schnittstelle gehört; sie ist dementsprechend auch vom Typ String.

Das Tagged-Value „EJBTransType“ wird schließlich unter dem veränderten Namen „EJBTransactionType“ als Tag-Definition übernommen. Die Zuordnung zum Stereotype «EJBSessionBean» bleibt bestehen. Da diese Tag-Definition eine Aussage über die verwendete Transaktionsverwaltung trifft („Bean“ oder „Container“), ist sie vom Typ String.

5.1.2 Auswirkungen von Änderungen des UML-1.4-Metamodells

Innerhalb der aktuellen UML-1.4-Spezifikation hat das Metamodellelement ***Component*** eine tiefgreifende semantische Änderung erfahren [OMG01, Seite „2-31“f.].

Dessen bisherige Funktion wird durch das neu eingeführte Metamodellelement ***Artifact*** übernommen.

Diese Veränderung hat Auswirkungen auf das EJB-bezogene Stereotype «EJBDescriptor», dessen bisherige Base-Class das Metamodell ***Component*** war.

Um im Rahmen der UML-1.4-Spezifikation weiterhin die ursprüngliche Semantik zum Ausdruck zu bringen, erbt dieses Stereotype nun vom standardisierten Stereotype «file», das wiederum das neu eingeführte Metamodellelement ***Artifact*** spezialisiert.

Die gleiche Anpassung gilt für das Java-bezogene Stereotype «JavaClassFile».

5.2 Optimierung der Profile-Architektur

Die Optimierung der Profile-Architektur basiert im Wesentlichen auf der nachträglich angewandten durchgehenden Nutzung des im vierten Kapitel angesprochenen „Top-Down-Ansatzes".

Als Ergebnis einer konsequenten Anwendung dieser Strategie ist eine im Vergleich zum Bottom-Up-Ansatz" hohe Redundanzminimierung zu erwarten.

Die konkreten Umsetzung des „Top-Down-Ansatzes" im Falle der EJB-Komponentenmodellierung orientiert sich dabei an folgenden drei Anforderungen:

1. Tag-Definitions, die gleichzeitig mehreren Stereotypes zugeordnet werden können, sind durch die Anwendung einer möglichst optimalen (Stereotype-bezogenen) Vererbungsrelation im Profile nur einmalig zu definieren.
2. Es sind diejenigen Redundanzen zu entfernen, die durch die wiederholte Darstellung von in der EJB-1.1-Spezifikation fest vorgegebenen Abbildungsvorschriften im bisherigen Profile entstanden sind.
3. Informationen, die sich aus dem logischen Aufbau der EJB-Komponentenspezifikation ergeben, dürfen nicht explizit modellierbar sein.

5.2.1 EJB-Jar

Die Struktur des Stereotypes «EJBJar» wurde im Rahmen der Optimierung nicht verändert.

5.2.2 Komponentenspezifikation

Die Struktur der komponentenspezifikationsbezogenen Erweiterungselemente hat sich nur minimal Verändert.

Zu den tiefgreifensten Veränderungen gehört die nicht erfolgte Wiedereinführung der beiden Tagged-Values „EJBNameInJAR" und „EJBReferences".

Die Funktion des Tagged-Values „EJBNameInJAR“ wird durch die Interpretation des Namens des jeweiligen EJB-darstellenden Subsystems übernommen.

Da eine zwischen zwei EJB-darstellenden Subsystemen bestehende Assoziation im Modell vorhandene EJB-Referenzen eindeutig beschreibt (siehe Abschnitt 5.2.6), kann das Tagged-Value „EJBReferences“ ebenfalls wegelassen werden.

Diejenigen Änderungen, die zu den neu eingeführten Tag-Definitions „EJBSessionType“ und „EJBTransactionType“ führten, sind in Abschnitt 5.1.1 erläutert.

Schließlich wurde das Stereotype «EJBPrimaryKey» durch das neu eingeführte Stereotype «EJBPrimaryKeyClass» ersetzt, dass zur Kennzeichnung der jeweils verwendeten Primärschlüsselklasse einer Entity-Bean dient. Es gehört, wie auch die durch die EJB realisierten Komponentenschnittstellen, zum Spezifikationsteil des EJB-darstellenden Subsystems.

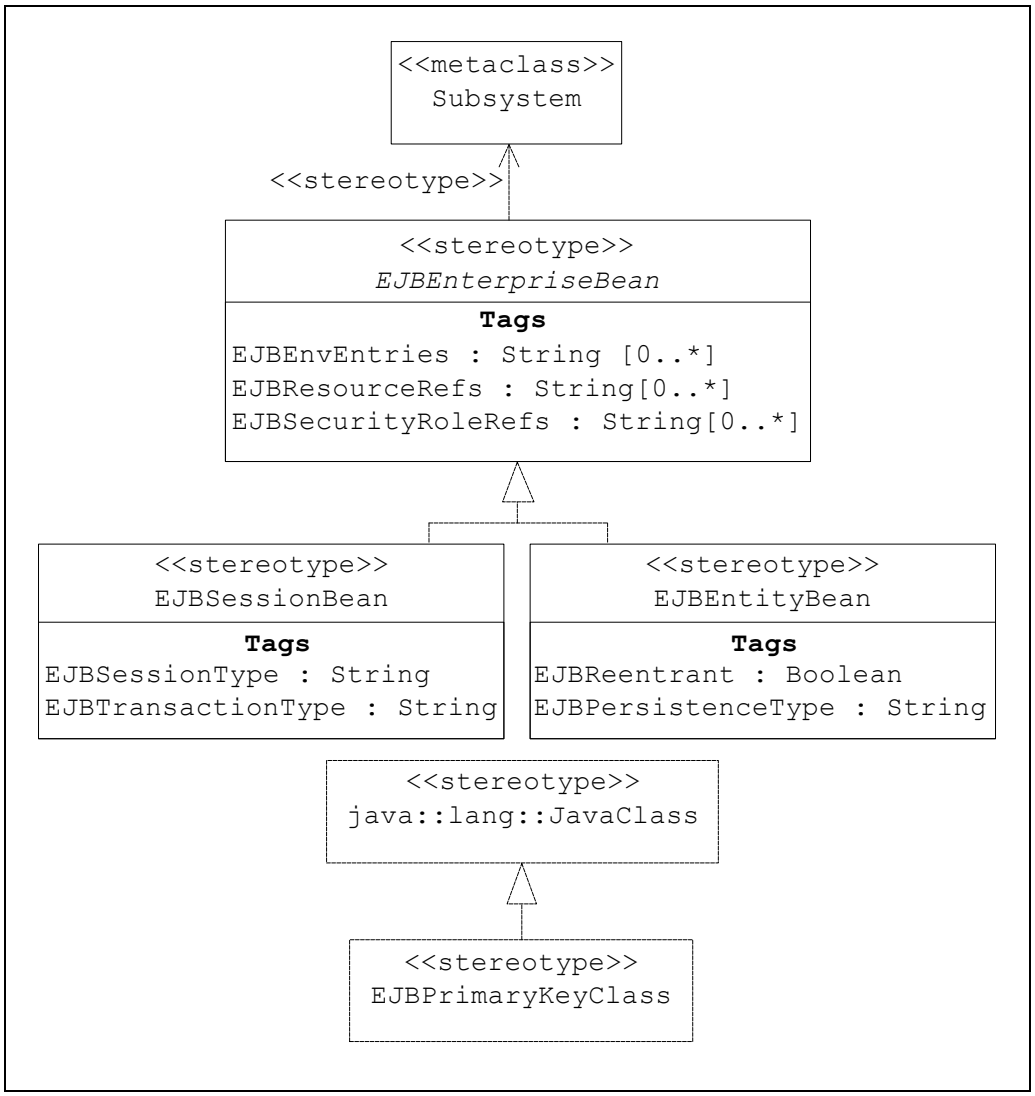


Abbildung 4 – Optimierte Architektur der komponentenspezifikationsbezogenen Streotypes

5.2.3 Komponentenschnittstellen

Die Veränderungen an den komponentenschnittstellenbezogenen Stereotypes sind ebenfalls minimal.

Um die Semantik der Java-bezogenen Erweiterungselemente wiederzuverwenden, spezialisieren die beiden Stereotypes «EJBRemoteInterface» und «EJBHomeInterface» nun nicht mehr das standardisierte Stereotype «type» sondern stattdessen das im Profile definierte Stereotype «JavaInterface».

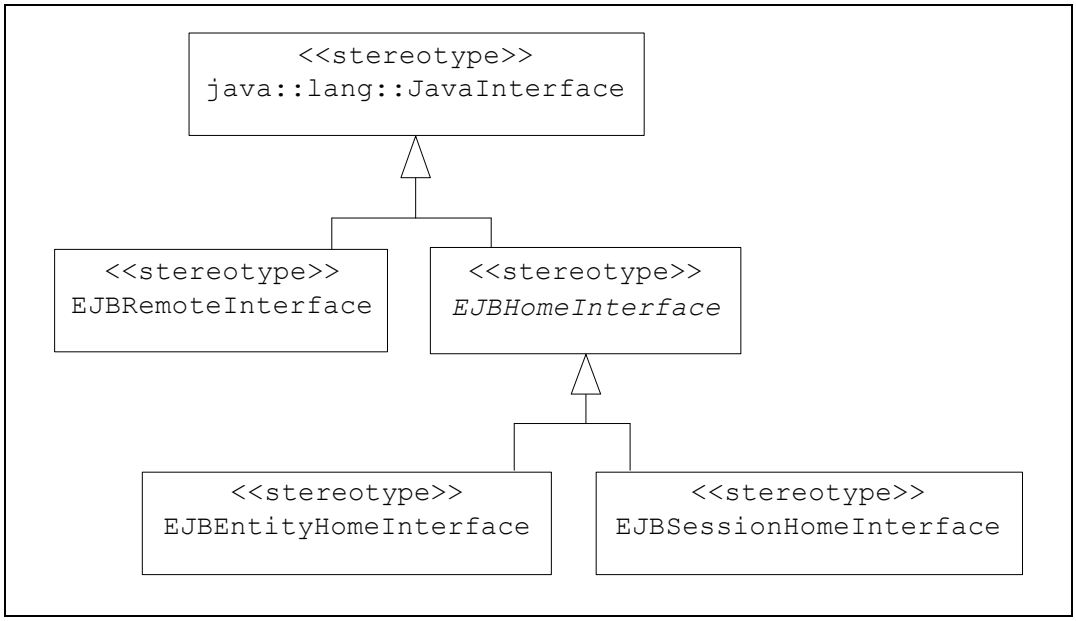


Abbildung 5 – Optimierte Architektur der komponentenschnittstellenbezogenen Stereotypes

5.2.4 Komponentenoperationen

Am einschneidesten sind die Veränderungen an der Struktur der komponentenoperationsbezogenen Stereotypes.

Der neue Ausgangspunkt dieser Erweiterungselemente ist das Java-bezogene Stereotype «JavaMethod», dass die Java-Methoden-spezifische Semantik kapselt.

Von ihm erben sowohl das zuvor schon vorhandene Stereotype «EJBRemoteMethod» als auch das neu eingeführte Stereotype «EJBBeanMethod».

Das Stereotype «EJBRemoteMethod» repräsentiert diejenigen Methoden, die im Rahmen der EJB-Komponentenschnittstellen definiert werden dürfen und die direkte Auswirkungen auf die Methodendefinition der dazugehörigen Implementierungsklasse haben. Diesem Stereotype werden auch die beiden Tag-Definitions „EJBTransAttribute“ und „EJBRoleNames“ zugeordnet (siehe Abschnitt 5.1.1).

Vom Stereotype «EJBRemoteMethod» erben wiederum die beiden Stereotypes «EJBCreateMethod» und «EJBFindMethod».

Das Stereotype «EJBCreateMethod» dient zur Kennzeichnung von Create-Methoden, die nur innerhalb der Home-Schnittstelle einer EJB Verwendung finden dürfen.

Mit dem Stereotype «EJBFindMethod» (früher «EJBFinderMethod») werden hingegen die sogenannten Find-Methoden gekennzeichnet, die ebenfalls nur innerhalb der Home-Schnittsteller einer EJB Verwendung finden dürfen.

Um die spezielle `findByPrimaryKey(...)`-Methode kennzeichnen zu können, wurde die Tag-Definition „EJBPKMethod“ eingeführt und dem Stereotype «EJBFindMethod» zugeordnet. Sie ist ihrer Funktion ensprechend vom Typ Boolean.

Das im Rahmen der Optimierung eingeführte Stereotype «EJBBeanMethod» dient zur Kennzeichnung von selbst definierten Implementierungsklassenmethoden. Die Einführung dieses Stereotypes ist insofern notwendig, da mit Hilfe der ihm (später) zugewiesenen Constraints die Modellkonsistenz gewährleistet werden kann.

So verhindert die, bei der Definition eigener Implementierungsklassenmethoden zwingend vorgeschrieben, Nutzung dieses Stereotypes Namenskonflike mit den automatisch generierten Methoden. Zu diesen gehören z.B. die ejbCreate- oder die ejbFind-Methoden.

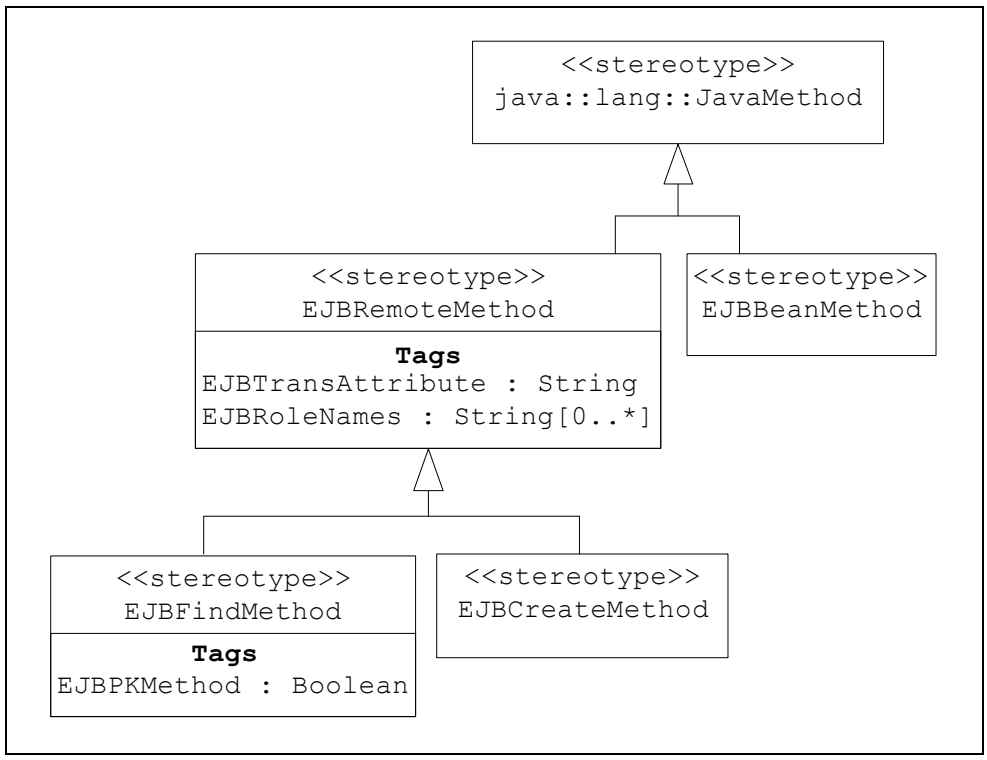


Abbildung 6 – Optimierte Architektur der komponentenoperationsbezogenen Stereotypes

5.2.5 Komponentenrealisierung

Auch bei der Struktur der komponentenrealisierungsbezogenen Stereotypes wurden Veränderungen durchgeführt.

Das Stereotype «EJBImplementation» spezialisiert nun anstelle des standardisierten Stereotypes «implementationClass» das im Profile definierte Java-bezogene Stereotype «JavaClass».

Die Änderungen bezüglich des Stereotypes «EJBCMPField» (früher «EJBCmpField») sind die umfangreichsten im Umfeld der komponentenrealisierungsbezogenen Erweiterungselemente.

Anders als in der ursprünglichen Fassung des Profiles spezialisiert das Stereotype nun das Java-bezogene Stereotype «JavaField» und erbt dadurch auch dessen Java-bezogene Semantik (siehe Anhang B).

Außerdem ist diesem Stereotype die eingeführte Tag-Definition „EJBPKField“ zugeordnet, die die Funktionalität des nicht mehr vorhandenen Stereotypes «EJBPrimaryKeyField» übernimmt. Da diese Tag-Definition anzeigt, ob es sich bei dem betreffenden CMP-Feld um ein Primärschlüsselfeld handelt, ist sie vom Typ Boolean.

Die beiden Stereotypes «EJBRealizeHome» und «EJBRealizeRemote» wurden bei der Optimierung ebenfalls gelöscht, da sich die betreffende, mit Hilfe der beiden Stereotypes ursprünglich zu beschreibende, Information aus dem logischen Aufbau der EJB-Komponentenspezifikation ergibt.

Die jeweiligen Schnittstellen sind entweder im Spezifikationsteil des EJB-darstellenden Subsystems definiert oder sie werden mit Hilfe der «import»-Semantik des Metamodellelements ***Subsystem*** in den betreffenden Namensraum importiert.

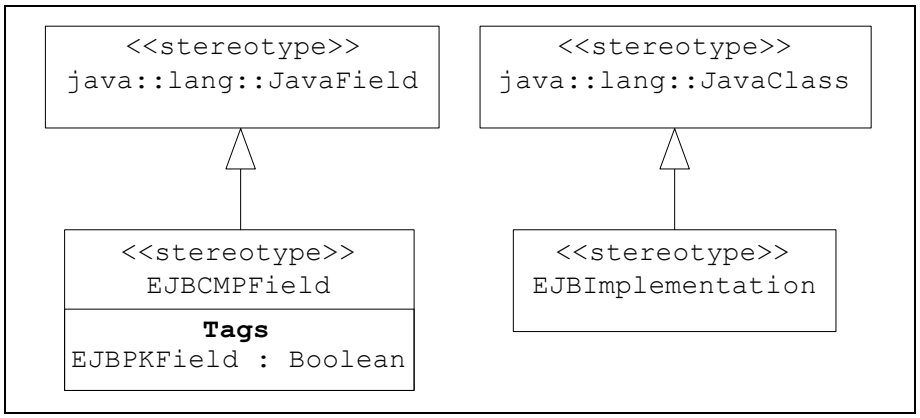


Abbildung 7 – Optimierte Architektur der komponentenrealisierungsbezogenen Stereotypes

5.2.6 Komponentenrelationen

Das Stereotype «EJBReference», das zur Beschreibung von Komponentenrelationen dient, die mittels EJB-Referenzen umgesetzt worden sind, wurde nicht mehr in die optimierte Fassung des Profiles übernommen.

Als Begründung hierfür gilt, dass eine zwischen zwei EJBs exisiterende Assoziation im Rahmen der EJB-1.1-Spezifikation nur durch die Anwendung einer EJB-Referenz umgesetzt werden kann und daher diese zwingend vorgegeben Codeabbildung nicht redundant mittels eines Stereotypes beschrieben werden muss.

Diese Aussage ließe sich zwar, bezogen auf Assoziationen zwischen Entity-Beans, gegebenfalls abschwächen (z.B. Referenzierung über Primärschlüssel und Finder-Methoden), da aber der EJB-1.1-Spezifikation diesbezüglich keine Aussagen zu entnehmen sind, gilt weiterhin die oben genannte Feststellung.

5.3 Bewertung

Abschließend gilt es, die Herstellung der UML-1.4-Konformität und die anschließende Optimierung der Profile-Architektur im Hinblick auf ihr Ergbebnis zu bewerten.

Da, wie im vierten Kapitel postuliert, eine Komplexitätsverminderung mit der Optimierung einhergehen müsste, wird versucht, dieses anhand einer Statistik zu verifizieren.

Im Anschluß daran wird eine Deutung des dort präsentieren Ergebnisses vorgenommen.

5.3.1 Statistik

Basierend auf dem Vergleich der EJB-bezogenen Erweiterungselemente beider Profile-Fassungen, also der Zahl der jeweils definierten Stereotypes und Tag-Definitions (Tagged-Values) sowie der sich dadurch insgesamt ergebenden absoluten Veränderung, kommt folgende Statistik zu stande:

Tabelle 5 - Statistik über die jeweilige Anzahl der EJB-bezogenen Erweiterungselemente

Anzahl der EJB-Erweiterungselemente des Typs...		
…Stereotype…	...Tag-Definition...	...bezogen auf...
21	11	...das ursprüngliches Profile.
18	11	...das optimiertes Profile.
-03	+00	...die absoluten Veränderungen.

5.3.2 Abschließendes Resümee

Das der Tabelle zu entnehmende Ergebnis entspricht in etwas dem nach Kapitel 4 zu erwartenden Resultat.

Dort wird eine Komplexitätsverminderung aufgrund der im Vergleich zum „Bottom-Up-Ansatz“ geringeren Anzahl an Erweiterungselementen in Aussicht gestellt, und diese Verminderung ist auch, zumindest bezogen auf die Zahl der Stereotypes, eingetroffen.

Dass diese Verminderung nicht spektakulär hoch ausgefallen ist, liegt hauptsächlich daran, dass die ursprüngliche Profile-Fassung nicht auf einem reinen „Bottom-Up-Ansatz“ basiert.

Insgesamt kann demnach folgendes Resümee gezogen werden:

- Die Herstellung der UML-1.4-Konformität hat eine grundlegende Verbesserung der Profile-Architektur zur Folge, da nun eine eindeutige Zuordnung von Tag-Definitions zu Stereotypes besteht, und damit sowohl die Lesbarkeit als auch die Erweiterungsfähigkeit des Profiles zugenommen hat.
- Die Optimierung der Profile-Architektur, die hauptsächlich aus der nachträglichen Anwendung des „Top-Down-Ansatzes“ bestand, hatte eine, wenn auch nicht sonderlich große, Komplexitätsverminderung zur Folge. Damit ist eine noch intuitivere Anwendbarkeit des Profiles in Bezug auf die EJB-Modellierung ermöglicht worden.
- Somit ist es nun auch einfacher, das Profile um die Neuerungen der EJB-2.0-Spezifikation zu erweitern, da die nach der Optimierung zugrundeliegende Architektur exakter und eindeutiger als zuvor definiert ist.

In Bezug auf die Vorgabe, dass alle in Abschnitt 4.5.1 postulierten Anforderungen zu erfüllen sind, können die folgenden Ergebnisse festgehalten werden:

- Die korrekte Anwendung des duch die aktuelle UML-1.4-Spezifikation vorgegebenen (light-weight) Metamodell-Erweiterungsmechanismus kann nach der erfolgten Optimierung festgestellt werden.
- Die geforderte durchgehende Nutzung einer Profile-Erstellungsstrategie – in diesem Fall die des Top-Down-Ansatzes – ist, insbesondere bezogen auf die erwartete Komplexitätsverminderung, erfolgreich erbracht worden.

Kapitel 6
Erweiterung des Profiles

Die Aufgabe dieses Kapitels besteht darin, weitere Konsequenzen aus den Bewertungsergebnissen des vierten Kapitels zu ziehen.

Diese Konsequenzen umfassen primär die Erweiterung des zuvor in Kapitel 5 optimierten Profiles und zwar um die Neuerungen der EJB-2.0-Spezifikation. Diese Erweiterung umfasst implizit auch die Verbesserung der Beschreibungsmöglichkeiten von Komponentenbeziehungen. Die angesprochene Verbesserung bezieht sich auf die beiden, nunmehr vorhandenen, Modellierungsmöglichkeiten zur Beschreibung von persistenten Assoziationen zwischen Entity-Beans auf der einen Seite und zur Beschreibung von Assoziationen zu den neu eingeführten Message-Driven-Beans auf der anderen Seite.

Der Erweiterungsprozess folgt in seiner Gliederung der in Kapitel 5 zuvor eingeführten Unterteilung der Optimierungsschritte, d.h. es wird mit dem EJB-Jar-bezogenen Abschnitt begonnen und es endet schließlich bei dem der Komponentenbeziehungen gewidmeten Abschnitt.

Zuvor werden jedoch die einzelnen Neuerungen der EJB-2.0-Spezifikation, nach Themengebieten sortiert, vorgestellt.

Im letzten Abschnitt erfolgt eine abschließende Bewertung der Erweiterung.

6.1 Neuerungen der EJB-2.0-Spezifikation

Die EJB-2.0-Spezifikation hat einige substanzielle Veränderungen bzw. Erweiterungen im Hinblick auf die Vorgängerversion mit sich gebracht. In den folgenden Teilabschnitten werden daher zunächst die einzelnen Veränderungen, nach einzelnen Themenbereichen sortiert, vorgestellt.

6.1.1 Message-Driven-Bean

Die Message-Driven-Bean („Message-Driven Bean“) stellt einen neuen Komponententyp innerhalb der EJB-Spezifikation dar. Mit seiner Hilfe ist die Modellierung und Implementierung von EJB-Komponenten möglich, die asynchron auf Ereignisse, in diesem Fall JMS-basierte Nachrichten, reagieren können. Im Gegensatz zu den bisherigen

Komponententypen (Session- bzw. Entity-Beans) bieten die Message-Driven-Beans keine Komponentenschnittstellen an. Stattdessen können sie mit Hilfe von songenannten On-Message-Methoden dezidiert auf bestimmte Ereignisklassen (Nachrichtentypen) reagieren.

6.1.2 Lokalaufrufbare Schnittstellen

Die sogenannten lokalaufrufbaren Schnittstellen („Local Interfaces“) stellen einen neuen Schnittstellentyp innerhalb der EJB-Spezifikation dar. Sie unterscheiden sich von den nun mehr als entferntaufrufbaren Schnittstellen („Remote-Interfaces“) bezeichneten bisherigen Schnittstellentypen dadurch, dass sie nur innerhalb der selben JVM zur Verfügung stehen. Dadurch findet im Rahmen der EJB-Technologie auch erstmalig das Prinzip der Pass-By-Reference seine Umsetzung. Eine Session- oder Entity-Bean kann seine jeweilige Component- und Home-Schnittstelle gleichzeitig als lokal- und als entferntaufrufbare Schnittstelle anbieten.

Gleichwohl ist die Verwendung nur eines Schnittstellentyps innerhalb der Spezifikation empfohlen; eine Mischung der beiden Schnittstellentypen ist hingegen nicht erlaubt.

6.1.3 Home-Methoden

Die neu eingeführten Home-Methoden („Home Methods“) ermöglichen die Einführung von Geschäftsmethoden auf Seiten der Entity-Beans, deren Ausführung unabhängig von den einzelnen Komponenteninstanzen ist. Sie werden innerhalb der lokal- oder entferntaufrufbaren Home-Schnittstelle einer Entity-Bean definiert und ihr Wirkungsradius umfasst alle Komponenteninstanzen eines bestimmten Entity-Bean-Typs.

6.1.4 Erweiterte Container-Managed-Persistence (CMP)

Die Container-Managed-Persistence wurde im Rahmen der EJB-2.0-Spezifikation signifikant erweitert.

So wurden erstmalig sogenannte Containter-Managed-Relationships (CMRs) eingeführt, die die Modellierung von persistenten Beziehungen zwischen Entity-Beans, ähnlich den datenbankbasierten Foreign-Keys-Beziehungen, ermöglichen. Grob gesprochen sind damit persistente uni- bzw. bidirektionale Assoziationen mit den Multiplizitäten „one-to-one“, „one-two-many“ und „many-to-many“ zwischen Entity-Beans möglich. Darüber hinaus

kann die Lebensdauer einer aggregierten Entity-Bean mittels einer Cascade-Delete-Anweisung beschränkt werden.

Des Weiteren wurde die CMP dahingehend modifiziert, das nun die persistenten Datenfelder innerhalb von Entity-Beans mittels abstrakter Getter-/Setter-Methoden modelliert werden und somit auch eine genaue Zugriffsbeschränkung innerhalb der lokal- oder entferntaufrufbaren Component-Schnittstelle möglich ist.

Sowohl alle definierten CMRs als auch alle vom Container verwalteten persistenten Datenfelder werden innerhalb des sogenannten abstrakten Datenschemas („Abstract Data Schema“) definiert. Dieses befindet sich innerhalb des jeweiligen Deployment-Descriptors und ist damit fester Bestandteil der Komponentenspezifikation.

6.1.5 EJB-QL

Im Rahmen der EJB-2.0-Spezifikation wurde auch die deklarative Abfragesprache EJB-QL, die an SQL angelehnt ist, eingeführt. Mit ihr ist es nun möglich, containerimplementierungsunabhängige Find- und Select-Methoden zu definieren.

6.1.6 Select-Methoden

Select-Methoden, die ebenfalls in der EJB-2.0-Spezifikation neu eingeführt wurden, ermöglichen es, innerhalb der Implementierungsklasse von Entity-Beans, Methoden zu definieren, die mittels der EJB-QL andere Entity-Beans selektieren und als Ergebnis zurückliefern.

6.2 Erweiterung

Die in den folgenden Abschnitten vorgestellte Erweiterung des Profiles basiert im Wesentlichen auf der Strategie des „Top-Down-Ansatzes“ (siehe Abschnitt 4.3.1).

Demnach erfüllen die einzelnen Erweiterungsschritte auch die in Abschnitt 5.2 formulierten Anforderungen.

6.2.1 EJB-Jar

Die Erweiterung des EJB-Jar-bezogenen Stereotypes «EJBJar» beschränkte sich auf die Einführung und Zuordnung der Tag-Definition „EJBSecurityRoles“.

Diese Tag-Definition dient der Definition von einzelnen Sicherheitsrollen, die zusammengenommen die Sicherheitssicht des jeweiligen Deployment-Descriptors bilden.

Die die Tag-Definition ist vom Typ String (Multiplizität: [0..*]).

Da die Definition von Sicherheitsrollen nun im Rahmen der Tag-Definition „EJBSecurityRoles“ stattfindet, wurde das Stereotype «EJBAccess» aus dem erweiterten Profile gestrichen.

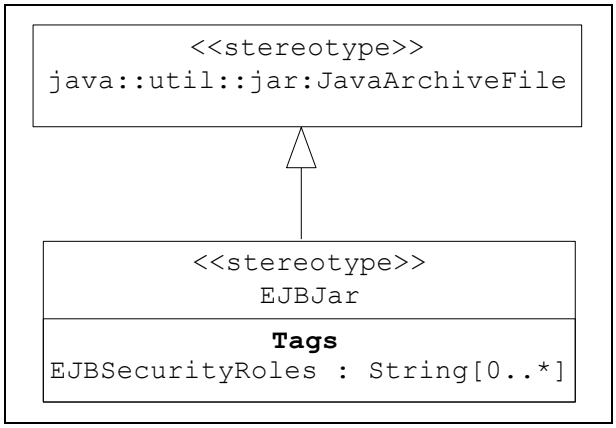


Abbildung 8 - Erweiterte Architektur der EJB-Jar-bezogenen Stereotypes

6.2.2 Komponentenspezifikation

Die Änderungen, die sich im Rahmen der Erweiterung des komponentenspezifikationsbezogenen Architekturabschnitts ergeben haben, sind die insgesamt umfangreichsten.

Zunächst werden diejenigen Änderungen der Profile-Architektur vorgestellt, die Aufgrund der Einführung des neuen EJB-Komponententyps Message-Driven-Bean erfolgten.

Da genauso wie bei den Session-Beans bei den Spezifikation der Message-Driven-Beans eine Aussage über die Art der Transaktionsverwaltung zu treffen ist, wurde die Tag-Definition „EJBTransactionType“ im Rahmen der erweiterten Profile-Architektur nunmehr dem neu eingeführten abstrakten Stereotype «EJBTransactionCapable» zugeordnet. Dessen Aufgabe beschränkt sich gemäß der ersten Anforderung aus Abschnitt 5.2 auf die

Gewährleistung der eindeutigen und einmaligen Zuordnung der Tag-Definition „EJBTransactionType“ innerhalb des Profiles.

Dementsprechend spezialsiert das Stereotype «EJBSessionBean» nicht mehr direkt «EJBEnterpriseBean» sondern das neu eingeführte Stereotype «EJBTransactionCapable».

Zur Kennzeichnung einer Mesage-Driven-Bean wurde das Stereotype «EJBMesageDrivenBean» eingeführt, dass, ananlog zu «EJBSessionBean», ebenfalls von «EJBTransactionCapable» erbt.

Diesem Stereotype wurden die ebenfalls neu eingeführten Tag-Definitions „EJBMessageSelektor“, „EJBAcknowledgeMode“, „EJBDestination“ sowie „EJBOnMsgTransAttr“ zugeordnet.

Die Tag-Definition „EJBOnMsgTransAttr“ dient zur Festlegung des Transaktionsattributs („`Required`“ oder „`NotSupported`“) der vorgegebenen `onMessage(...)`-Methode und ist daher vom Typ String.

Die Bedeutung der anderen drei Tag-Defintions ist [SUN01, Seite 458] zu entnehmen.

Die nächsten beiden Erweiterungen betreffen das Stereotype «EJBEnterpriseBean».

Da die EJB-2.0-Spezifikation nun auch sogenannte „Resource-Environment“-Referenzen vorsieht [SUN01, Seite 427ff.], wurde dem Stereotype «EJBEnterpriseBean» die neu eingeführte Tag-Definition „EJBResourceEnvRefs“ zugeordnet, die vom Typ String [Multiplizität: [0..*]) ist.

Außerdem sieht die EJB-2.0-Spezifikation auch eine sogenannte „run-as“-Identität vor, die es ihrerseits ermöglicht, der EJB eine eigene, vom aufrufenden Client unterschiedliche, Identität zuzuordnen [SUN01, Seite 446f.]. Hierfür wurde dem Sterotype «EJBEnterpriseBean» die ebenfalls neu eingeführte Tag-Definition „EJBSecurityID“ zugeordnet. Diese Tag-Definition ist vom Typ String.

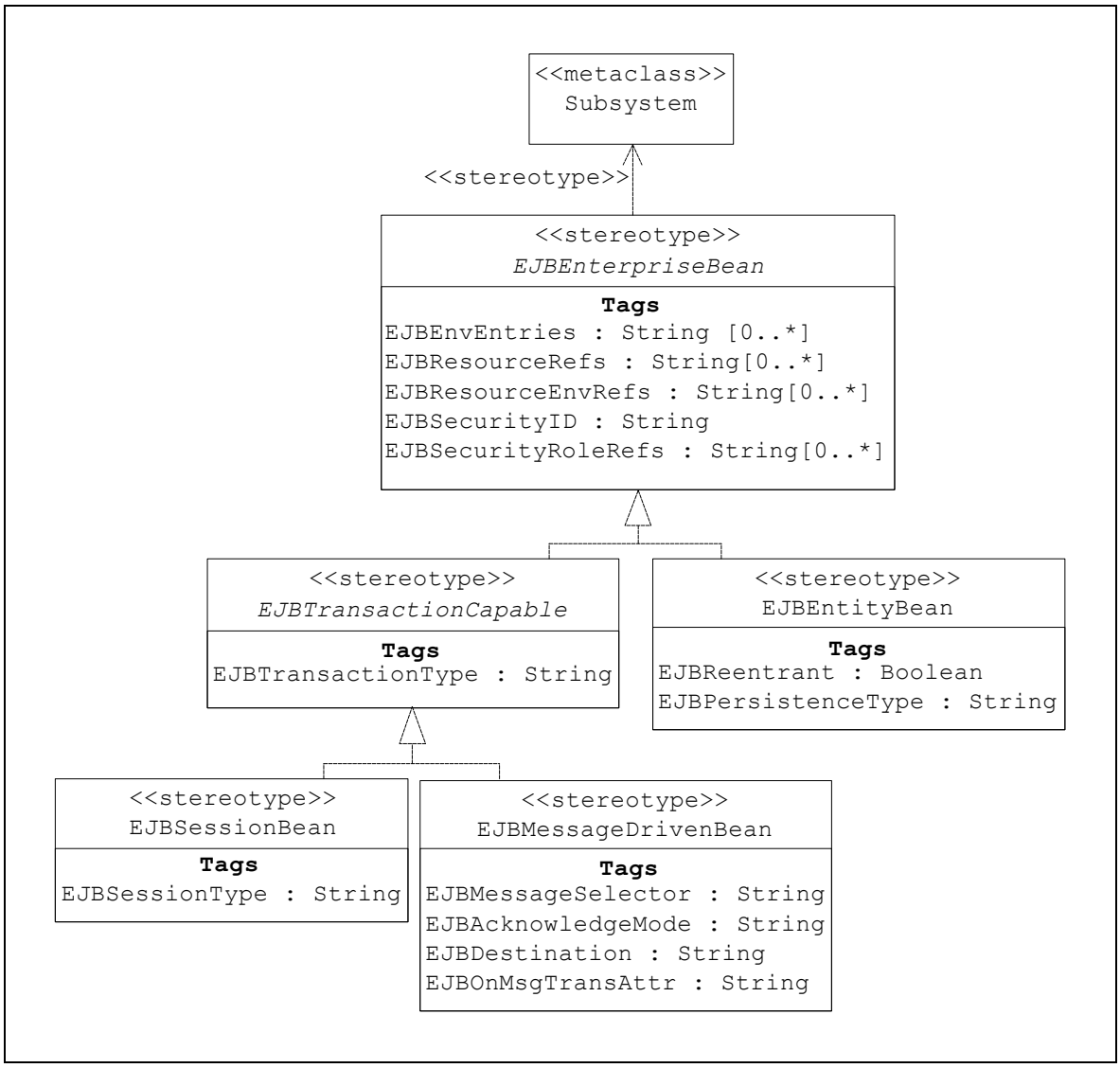


Abbildung 9 - Erweiterte Architektur der komponentenspezifikationsbezogenen Stereotypes (1/2)

Die letzte große Änderung der komponentenspezifikationsbezogenen Profile-Architektur betrifft die in der EJB-2.0-Spezifikation erfolgte Einführung des abstrakten Dataschemas.

So wurde diesbezüglich zunächst das Stereotype «EJBAbstractDataSchema» zur Abbildung der eben angesprochenen und dabei Entity-Bean-bezogenen abstrakten Datenschemata eingeführt.

Dieses Stereotype hat als Base-Class das Metamodellelement ***Class*** und wird im Rahmen des Spezifikationsteils eines Entity-Bean-darstellenden Subsystems modelliert.

Das Stereotype «EJBAbstractDataSchema» wurde hauptsächlich deshalb eingeführt, um eine, dem „Top-Down-Ansatz“ entsprechende, Modellierung von CMP-Felder zu ermöglichen.

Daher kann ein mit dem Stereotype «EJBCMPField» versehenes Attribut nur noch innerhalb der als «EJBAbstractDataSchema» deklarierten Klasse definiert werden und somit gehört das Stereotype im Rahmen des erweiterten Profile nunmehr zur Komponentenspezifikation und nicht mehr zur Komponentenrealisierung.

Die EJB-2.0-basierten Änderungen der CMP-verwalteten Felder sind in Form von zwei neu eingeführten und dabei dem Stereotype «EJBCMPField» zugeordneten Tag-Definitions berücksichtigt worden.

Die Tag-Definition „EJBRemoteAccessModifier" ist vom Typ String und regelt die Veröffentlichung der abstrakten Getter-/Setter-Methoden innerhalb der entferntaufrufbaren Component-Schnittstelle.

Die Tag-Definition „EJBLocalAccessModifier" ist ebenfalls vom Typ String und regelt analog zu „EJBRemoteAccessModifier" die Veröffentlichung der Getter-/Setter-Methoden innerhalb der lokalaufrufbaren Component-Schnittstelle.

Mögliche Werte der jeweils zur Laufzeit bestehenden Tagged-Values sind:

- „Get" für nur lesenden Zugriff,
- „Set" für nur schreibenden Zugriff,
- „Both" für lesenden und schreibenden Zugriff,
- „None" für nicht erlaubten Zugriff.

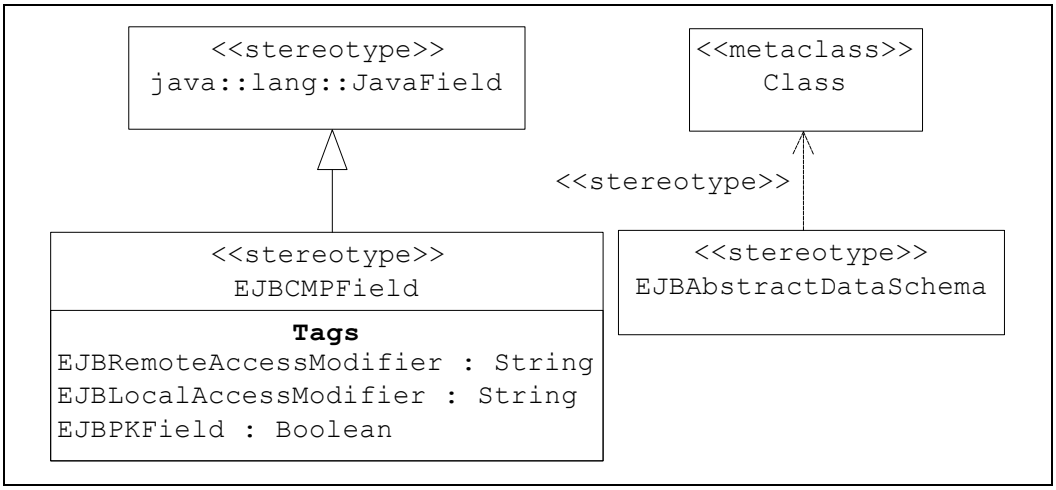


Abbildung 10 - Erweiterte Architektur der komponentenspezifikationsbezogenen Stereotypes (2/2)

6.2.3 Komponentenschnittstellen

Obwohl die Änderungen betreffend der EJB-Komponentenschnittstellen im Rahmen der EJB-2.0-Spezifikation von weitreichender Natur sind, fallen die Erweiterungen der dafür zuständigen Stereotype-Architektur innerhalb des Profiles nur minimal aus.

So wurde der betreffenden Vererbungsstruktur das neu eingeführte abstrakte Stereotype «EJBInterface» hinzugefügt, dass einzig allein der eindeutigen und einmaligen Zuordnung der ebenfalls neu eingeführten Tag-Definition „EJBLocal“ dient. Dabei befindet sich das Stereotype, bezogen auf die Struktur des Vererbungsbaums, zwischen dem Stereotype «JavaInterface» auf der einen Seite und den beiden Stereotypes «EJBComponentInterface» (früher «EJBRemoteInterface») und «EJBHomeInterface» auf der anderen Seite.

Die Tag-Definition „EJBLocal“ dient dabei der Festlegung, ob es sich bei der jeweiligen Schnittstelle um eine lokal- („`True`“) oder entferntaufrufbare („`False`“) handelt; sie ist dementsprechend vom Typ Boolean.

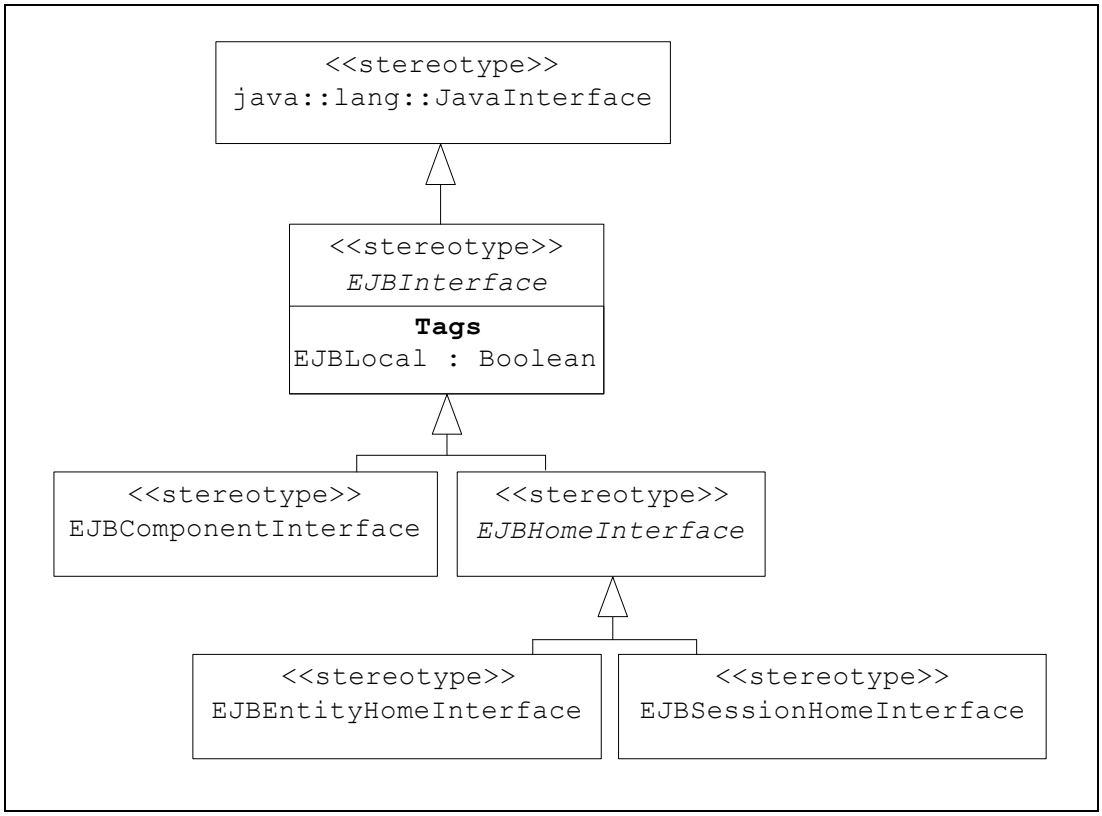


Abbildung 11 - Erweiterte Architektur der komponentenschnittstellenbezogenen Stereotypes

6.2.4 Komponentenoperationen

Weitere einschneidende Veränderungen der Profile-Architektur fanden bei den komponentenoperationsbezogenen Stereotypes statt.

Um die im Rahmen der EJB-2.0-Spezifikation eingeführten Home-Methoden modellieren zu können, wurde das Stereotype «EJBHomeMethod» eingeführt. Dabei spezialisert es, analog zu den beiden Stereotypes «EJBCreateMethod» und «EJBFindMethod», ebenfalls das Stereotype «EJBRemoteMethod».

Die Berücksichtigung der Abfragesprache EJB-QL fand durch die Einführung des abstrakten Stereotypes «EJBQueryMethod» statt. Diesem wurde die ebenfalls neu eingeführte Tag-Definition „EJBQuery“ zugeordnet, die zur Beschreibung einer EJB-QL-Anweisung dient und daher vom Typ String ist.

Da die Find-Methoden im Unterschied zur EJB-1.1-Spezifikation nun mittels einer containerimplementierungsunabhängigen EJB-QL-Anweisung spezifiziert werden, erbt das Stereotype «EJBFindMethod» innerhalb der erweiterten Profile-Architektur vom neu eingeführten Stereotype «EJBQueryMethod».

Gleiches gilt für die auf Seiten der Implementierungsklasse neu eingeführten Select-Methode, die mittels des Stereotypes «EJBSelectMethod» im nunmehr erweiterten Profile berücksichtigt wird, und, analog zur Find-Methode, ebenfalls von «EJBQueryMethod» erbt. Zusätzlich erbt «EJBSelectMethod» noch vom Stereotype «EJBBeanMethod».

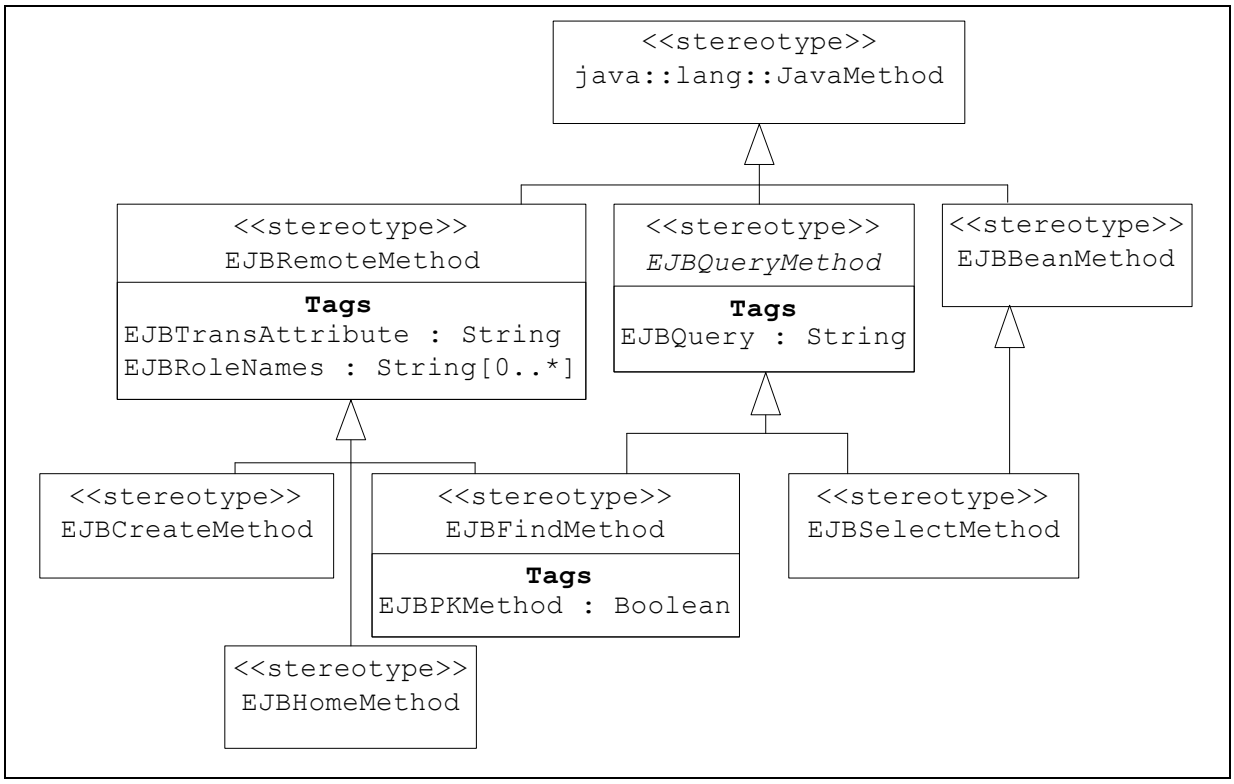


Abbildung 12 - Erweiterte Architektur der komponentenoperationsbezogenen Stereotypes

6.2.5 Komponentenrealisierung

Im Hinblick auf die komponentenrealisierungsbezogene Profile-Architektur erfolgten bis auf die semantische Neubewertung des Stereotypes «EJBCMPField» (im Sinne der Zugehörigkeit zur Komponentenspezifikation, siehe Abschnitt 6.2.2) keine weiteren Änderungen.

6.2.6 Komponentenrelationen

Um den in der EJB-2.0-Spezifikation neu eingeführten CMRs Rechnung zu tragen, musste die komponentenrelationsbezogene Profile-Architektur signifikant erweitert werden.

So wurde zunächst das neue Stereotype «EJBCMRelationship» eingeführt, dessen Base-Class das Metamodellelement ***Association*** ist. Dadurch können zwischen Entity-Beans bestehende CMRs von den EJB-Referenzen-basierten Assoziationen unterschieden werden.

Parallel dazu wurde das Stereotype «EJBCMPAssociationEnd» eingeführt, das die UML-Metaklasse ***AssociationEnd*** erweitert und somit die zum Stereotype «EJBCMRelationship» zugehörigen Assoziationsenden repräsentiert.

Ihm wurden die beiden neu eingeführten Tag-Definitions „EJBAccessModifier“ und „EJBCascadeDelete“ zugeordnet.

Die Tag-Definition „EJBAccessModifier“ entspricht in ihrer Semantik den beiden Tag-Definitions „EJBLocalAccessModifier“ bzw. „EJBRemoteAccessModifier“ (siehe Abschnitt 6.2.5), bezieht sich jedoch im Unterschied dazu auf die lokalaufrufbare Component-Schnittstelle der jeweils assoziierte Entity-Bean.

Mittels der Tag-Definition „EJBCascadeDelete“ kann festgelegt werden, ob die Lebensdauer der assoziierten Entity-Bean gemäß der UML-Kompositionsemantik begrenzt werden soll; sie ist daher vom Typ Boolean.

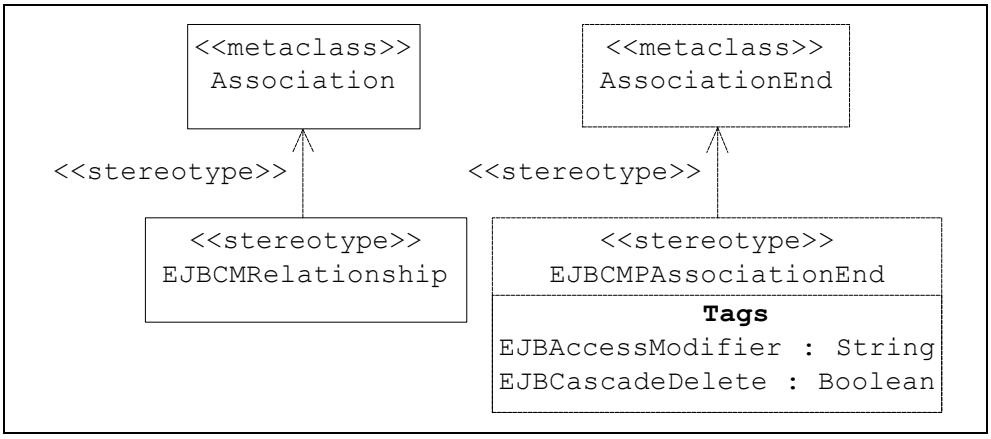


Abbildung 13 - Erweiterte Architektur der komponentenrelationsbezogenen Stereotypes

6.3 Bewertung

Es gilt nun, analog zur Optimierung, das nach der Erweiterung erzielte Ergebnis zu bewerten.

Die Grundlage hierfür bildet eine Statistik, die in ihrer Struktur der aus Abschnitt 5.3.1 entspricht.

6.3.1 Statistik

Tabelle 6 – 2. Statistik über die jeweilige Anzahl der EJB-bezogenen Erweiterungselemente

Anzahl der EJB-Erweiterungselemente des Typs...		
…Stereotype…	...Tag-Definition...	...bezogen auf...
18	11	...das optimierte Profile.
24	24	...das erweiterte Profile.
+06	+15	...die absoluten Veränderungen.

6.3.2 Abschließendes Resümee

Als abschließendes Resümee kann festgehalten werden, dass im Rahmen der Erweiterung eine deutliche Zunahme der im Profile definierten Stereotypes und Tag-Definitions erfolgt ist.

Diese Komplexitätszunahme ist aber aufgrund der signifikaten Erweiterung der EJB-2.0-Spezifikation im Vergleich zu ihrer Vorgängerversion nicht weiter überraschend.

Die Frage, ob die auf dem „Top-Down-Ansatz“ gründende Erweiterung des Profiles das dafür best mögliche Ergebnis erzielt hat, kann jedoch ohne einen dazu alternativen Erweiterungsansatz nur hypothetisch mit „ja“ beantwortet werden.

In Bezug auf die Vorgabe, dass alle in Abschnitt 4.5.2 postulierten Anforderungen zu erfüllen sind, können demnach folgende Ergebnisse festgehalten werden:

- Sowohl die vollständige Berücksichtigung der im zweiten Kapitel formulierten komponentenstrukturabhängigen Anforderungen als auch die vollständige Berücksichtigung der dort ebenfalls formulierten komponentenbeziehungsabhängigen Anforderungen, ist durch die uneingeschränkte Umsetzung der EJB-2.0-Spezifikation gewährleistet.
- Die geforderte Vervollständigung der Profile-Spezifikation erfolgte im Rahmen der Erweiterung und ist als Ergebnis Anhang C zu entnehmen.

Kapitel 7
Abbildungsvorschriften

Dieses Kapitel hat die Erarbeitung von Abbildungsvorschriften zum Inhalt, die, der Semantik des erweiterten „UML Profile for EJB“ folgend, aus auf diesem Profile basierenden UML-Entwurfsmodellen EJB-Codefragemente generieren. Diese Codefragmente dienen somit als Grundlage eines EJB-Jars und müssen nur noch durch manuelle Ergänzungen z.B. der Methodenrümpfe an die jeweiligen Anwendungsfälle angepaßt werden. Da die Java-bezogenen Erweiterungselemente nicht zum eigentlichen Kern dieser Arbeit gehören, werden sie auch nicht im Rahmen der Abbildungsvorschriften berücksichtigt.

Die Erläuterung der einzelnen Abbildungsvorschriften orientiert sich dabei vordergründig an der zuvor schon in den beiden vorhergehenden Kapiteln eingeführten Gliederung (EJB-Jar, ..., Komponentenrelationen) und hintergründig an den zur Verfügung stehenden Erweiterungselementen (Stereotypes und Tag-Definitions).

Um den Umfang dieses Kapitels möglichst klein zu halten und um die Lesbarkeit insgesamt zu erhöhen, folgt der inhaltliche Aufbau der Erläuterungen dem folgenden Konzept:

1. Die Abbildungsvorschriften unterteilen sich in solche, die die Generierung der Java-Codefragmente beschreiben (Überschrift: **Java-Codefragmente**), und solche, die die Generierung der jeweiligen XML-Konstrukte des Deployment-Descriptors beschreiben (Überschrift: **Deployment-Descriptor**). Im Rahmen einzelner Abbildungsvorschriften kann es vorkommen, dass eine der beiden Teilabbildungen nicht definiert wird. Dies bedeutet, dass für die jeweiligen Zielsprache (Java oder XML) keine Abbildung vorgesehen ist.
2. Im Rahmen der Abbildungsvorschriften werden sogenannte Templates (Syntax: **<TEMPLATE>**) verwendet, die dem kompakten Aufbau der Vorschriften dienlich sind und somit auch zur besseren Lesbarkeit beitragen.
3. Die Abbildungsvorschriften folgen der Vererbungsstruktur der im erweiterten Profile definierten Stereotypes. Das bedeutet, dass z.B. Abbildungen von einzelnen Tagged-Values dort beschrieben werden, wo sie im Rahmen von Tag-Definitions

definiert werden. Dadurch werden auch Abbildungsvorschriften für abstrakte und damit nicht unmittelbar anwendbare Stereotypes eingeführt, die aber im Rahmen der jeweils gültigen Vererbungsrelationen der Stereotypes letztendlich dorch zur Anwendung kommen.

4. Stereotypes oder Tagged-Values, die nicht im Rahmen der Abbildungsvorschriften behandelt werden, besitzen auch keine semantischen Auswirkungen auf die Genrierung der EJB-Codefragmente.

Es folgen nun die einzelnen Abbildungsvorschriften.

7.1 EJB-Jar

7.1.1 «EJBJar»

Deployment-Descriptor:

Ein Package, das mit dem Stereotype «EJBJar» gekennzeichnet ist, wird auf das folgende XML-Konstrukt abgebildet:

```
<ejb-jar>
    <enterprise-beans>
    </enterprise-beans/>
    <assembly-descriptor>
    </assembly-descriptor>
</ejb-jar>
```

Der Name (**<NAME>**) des Packages wird dabei auf folgendes XML-Konstrukt abgebildet:

```
<ejb-jar>
    <display-name><NAME></display-name>
</ejb-jar>
```

Die einzelnen Werte (**<VALUE_X>**) des Tagged-Values „EJBSecurityRoles“ werden schließlich jeweils auf ein Exemplar des folgenden XML-Konstrukts abgebildet:

```
<security-role>
    <role-name><VALUE_X></role-name>
</security-role>
```

7.2 Komponentenspezifikation

7.2.1 «EJBEnterpriseBean»

Java-Codefragmente:

Hinweis:
Für alle im Rahmen der Java-Codefragemente generierten Instanzattribute gilt grundsätzlich, dass sie mit dem Schlüsselwort `transient` gekennzeichnet werden, um eventuellen Problemen bei der Serialisierung und später folgenden Deserialisierung von EJB-Handles vorzubeugen.

Die Bestandteile **<NAME>** und **<TYPE>** der einzelnen Strings des Tagged-Values „EJBEnvEntries“ werden zunächst jeweils auf ein Instanzattribut der zugehörigen Implementierungsklasse abgebildet:

```
private transient <TYPE> envEntry<NAME>;
```

Des Weiteren wird eine Methode definiert, die zur Laufzeit die zugewiesenen Werte ausliest:

```
private void getAllEnvEntries() {

    try {
        Context initCtx = new InitialContext();
        Context myEnv = (Context)initCtx.lookup("java:comp/env");

        this.envEntry<NAME_1> = (<TYPE_1>)myEnv.lookup("<NAME_1>");
        ...
        this.envEntry<NAME_N> = (<TYPE_N>)myEnv.lookup("<NAME_N>");
    } catch(java.lang.Exception e) {
    }
}
```

Die Bestandteile **<NAME>** und **<TYPE>** der einzelnen Strings des Tagged-Values „EJBResourceRefs“ werden zunächst jeweils ein Instanzattribut der zugehörigen Implementierungsklasse abgebildet:

```
private transient <TYPE> resRef<NAME>;
```

Des Weiteren wird eine Methode definiert, die zur Laufzeit die zugewiesenen Werte ausliest:

```
private void getAllResourceRefs() {

    try {
        Context initCtx = new InitialContext();
        Context myEnv = (Context)initCtx.lookup("java:comp/env");
```

```
        this.resRef<NAME_1> = (<TYPE_1>)myEnv.lookup("<NAME_1>");
        ...
        this.resRef<NAME_N> = (<TYPE_N>)myEnv.lookup("<NAME_N>");
    } catch(java.lang.Exception e) {
    }
}
```

Die Bestandteile **<NAME>** und **<TYPE>** der einzelnen Strings des Tagged-Values „EJBResourceEnvRefs“ werden zunächst jeweils auf ein Instanzattribut der zugehörigen Implementierungsklasse abgebildet:

```
private transient <TYPE> resEnvRef<NAME>;
```

Des Weiteren wird eine Methode definiert, die zur Laufzeit die zugewiesenen Werte ausliest:

```
private void getAllResourceEnvRefs() {

    try {
        Context initCtx = new InitialContext();
        Context myEnv = (Context)initCtx.lookup("java:comp/env");

        this.resEnvRef<NAME_1> = (<TYPE_1>)myEnv.lookup("<NAME_1>");
        ...
        this.resEnvRef<NAME_N> = (<TYPE_N>)myEnv.lookup("<NAME_N>");
    } catch(java.lang.Exception e) {
    }
}
```

Deployment-Descriptor:

Der Name (**<NAME>**) des Subsystems wird auf das folgende XML-Konstrukt abgebildet:

```
<ejb-name><NAME></ejb-name>
```

Die Bestandteile **<NAME>**, **<TYPE>** und **<VALUE>** der einzelnen Strings des Tagged-Values „EJBEnvEntries“ werden jeweils auf ein Exemplar des folgenden XML-Konstrukts abgebildet:

```
<env-entry>
    <env-entry-name><NAME></env-entry-name>
    <env-entry-type><TYPE></env-entry-type>
    <env-entry-value><VALUE></env-entry-value>
</env-entry>
```

Die Bestandteile **<NAME>**, **<TYPE>**, **<AUTH>** und **<SCOPE>** der einzelnen Strings des Tagged-Values „EJBResourceRefs“ werden jeweils auf ein Exemplar des folgenden XML-Konstrukts abgebildet:

```
<resource-ref>
    <res-ref-name><NAME></res-ref-name>
```

```
    <res-type><TYPE></res-type>
    <res-auth><AUTH></res-auth>
    <res-sharing-scope><SCOPE></res-sharing-scope>
</resource-ref>
```

Die Bestandteile **<NAME>** und **<TYPE>** der einzelnen Strings des Tagged-Values „EJBResourceEnvRefs“ werden jeweils auf ein Exemplar des folgenden XML-Konstrukts abgebildet:

```
<resource-env-ref>
    <resource-env–ref-name><NAME></resource-env–ref-name >
    <resource-env-ref-type><TYPE></resource-env-ref-type >
</resource-env-ref>
```

Der Inhalt (**<VALUE>**) des Tag-Values „EJBSecurityID“ wird, falls vorhanden, auf das folgende XML-Konstrukt abgebildet:

```
<security-identity>
    <run-as>
        <role-name><value></role-name>
    <run-as/>
</security-identity>
```

Falls das Tagged-Value „EJBSecurityID“ nicht definiert ist, wird folgendes XML-Konstrukt generiert:

```
<security-identity>
    <user-caller-identity/>
</security-identity>
```

Die Bestandteile **<NAME>** und **<LINK>** der einzelnen Strings des Tagged-Values „EJBSecurityRoleRefs“ werden schließlich jeweils auf ein Exemplar des folgenden XML-Konstrukts abgebildet:

```
<security-role-ref>
    <role-name><NAME></role-name>
    <role-link><LINK></role-link>
</security-role-ref>
```

7.2.2 «EJBTransactionCapable»

Deployment-Descriptor:

Der Inhalt (**<VALUE>**) des Tagged-Values „EJBTransactionType“ wird auf das folgende XML-Konstrukt abgebildet:

```
<transaction-type><VALUE><transaction-type>
```

7.2.3 «EJBSessionBean»

Deployment-Descriptor:

Ein Subsystem, das mit dem Stereotype «EJBSessionBean» gekennzeichnet ist, steht für die Komponentenspezifikation einer Session-Bean und wird auf das folgende XML-Konstrukt abgebildet:

```
<session>
</session>
```

Der Inhalt (**<VALUE>**) des Tagged-Values „EJBSessionType“ wird schließlich auf das folgende XML-Konstrukt abgebildet:

```
<session-type><VALUE></session-type>
```

7.2.4 «EJBMessageDrivenBean»

Deployment-Descriptor:

Ein Subsystem, das mit dem Stereotype «EJBMessageDrivenBean» gekennzeichnet ist, steht für die Komponentenspezifikation einer Message-Driven-Bean und wird auf das folgende XML-Konstrukt abgebildet:

```
<message-driven>
</message-driven>
```

Die jeweiligen Inhalte **<SELEC>** bzw. **<ACKN>** der Tagged-Values „EJBMessageSelector“ bzw. „EJBAcknowledgeMode“ werden auf folgendes XML-Konstrukt abgebildet:

```
<message-selector><SELEC></message-selector>
<acknowledge-mode><ACKN></acknowledge-mode>
```

Die Bestandteile **<TYPE>** und **<DURABILITY>** des Tagged-Values „EJBDestination“ werden auf das folgende XML-Konstrukt abgebildet:

```
<message-driven-destination>
    <destination-type><TYPE></destination-type>
    <subscription-durability><DURABILITY></subscription-durability>
</message-driven-destination>
```

Die Abbildung des Inhalts (**<VALUE>**) des Tagged-Values „EJBOnMsgTransAttr“ wird schließlich ananlog zum Inhalt des Tagged-Values „EJBTransAttr“ («EJBRemoteMethod») abgebildet (siehe Abschnitt 7.4.1).

7.2.5 «EJBEntityBean»

Deployment-Descriptor:

Ein Subsystem, das mit dem Stereotype «EJBEntityBean» gekennzeichnet ist, steht für die Komponentenspezifikation einer Entity-Bean und wird auf das folgende XML-Konstrukt abgebildet:

```
<entity>
    <cmp-version>2.x</cmp-version>
</entity>
```

Die Inhalte **<REEN>** bzw. **<PERS>** der beiden Tagged-Values „EJBReentrant“ bzw. „EJBPersistenceType“ werden schließlich auf folgendes XML-Konstrukt abgebildet:

```
<reentrant><REEN></reentrant>
<persistence-type><PERS></persistence-type>
```

7.2.6 «EJBPrimaryKeyClass»

Deployment-Descriptor:

Der vollständig qualifizierte Name (**<NAME>**) einer mit dem Stereotype «EJBPrimaryKeyClass» gekennzeichneten Klasse wird auf das folgende XML-Konstrukt abgebildet:

```
<prim-key-class><NAME></prim-key-class>
```

7.2.7 «EJBAbstractDataSchema»

Deployment-Descriptor:

Der Name (**<NAME>**) einer mit dem Stereotypes «EJBAbstractDataSchema» gekennzeichneten Klasse, wird auf das folgende XML-Konstrukt abgebildet:

```
<abstract-schema-name><NAME></abstract-schema-name>
```

7.2.8 «EJBCMPField»

Java-Codefragmente:

Ein mit dem Stereotype «EJBCMPField» gekennzeichnetes Attribut (**<NAME>**, **<TYPE>**) wird zunächst grundsätzlich auf abstrakte Getter-/Setter-Methoden der jeweiligen Implementierungsklassse abgebildet:

```
public abstract <TYPE> get<NAME>();
public abstract void set<NAME>(<TYPE> <NAME>);
```

Abhängig von den Werten der Tagged-Values „EJBRemoteAccessModifier“ und „EJBLocalAccessModifier“ werden die abstrakten Getter-/Setter-Methoden auch in der jeweiligen Component-Schnittstelle veröffentlicht:

```
<TYPE> get<NAME>();
void set<NAME>(<TYPE> <NAME>);
```

Handelt es sich bei der Component-Schnittstelle um eine entferntaufrufbare, so ist der obigen Methodendefinition noch ein eine Throw-Anweisung hinzuzufügen, die die Java-Exception `java.rmi.RemoteException` beinhaltet.

Ist das Tagged-Value „EJBPKField” auf „True” gesetzt, so kann keine Setter-Methode in der jeweiligen Component-Schnittstelle veröffentlich werden.

Deployment-Descriptor:

Der Name des mit Stereotype «EJBCMPField» versehenen Attributs (**<NAME>**) wird auf das folgende XML-Konstrukt abgebildet:

```
<cmp-field>
    <field-name><NAME></field-name>
</cmp-field>
```

Falls der Wert des Tagged-Values „EJBPKField“ auf „True“ gesetzt ist, wird **<NAME>** außerdem noch auf folgendes XML-Konstrukt abgebildet:

```
<primkey-field><NAME></primkey-field>
```

7.3 Komponentenschnittstellen

7.3.1 «EJBInterface»

Bei allen Klassen, die mit einem direkt oder indirekt von «EJBInterface» erbenden Stereotype gekennzeichnet sind, wird zunächst folgende Import-Anweisung an den Anfang der Schnittstellendefinition gestellt:

```
import javax.ejb.*;
```

Außerdem wird der Package-Pfad **<PACKAGE>** des betroffenen Klassennames auf eine Package-Anweisung vor der eigentlichen Schnittstellendefinition abgebildet:

```
package <PACKAGE>;
```

7.3.2 «EJBComponentInterface»

Java-Codefragmente:

Abhängig vom Wert des Tagged-Values „EJBLocal“ erfolgt die Abbildung einer mit dem Stereotype «EJBComponentInterface» gekennzeichneten Klasse (**<NAME>**) entweder („EJBLocal“ = „False“) auf das Codefragment:

```
import java.rmi.*;

...

interface <NAME> extends EJBObject {
}
```

oder („EJBLocal“ = „True“) auf das Codefragment:

```
...

interface <NAME> extends EJBLocalObject {
}
```

Deployment-Descriptor:

Abhängig vom Wert des Tagged-Values „EJBLocal“ erfolgt die Abbildung des voll qualifizierten Names (**<PACKAGE>.<NAME>**) einer mit dem Stereotype «EJBComponentInterface» gekennzeichneten Klasse entweder („EJBLocal“ = „False“) auf das XML-Konstrukt:

```
<remote><PACKAGE>.<NAME></remote>
```

oder („EJBLocal“ = „True“) auf das XML-Konstrukt:

```
<local><PACKAGE>.<NAME></local>
```

7.3.3 «EJBHomeInterface»

Java-Codefragmente:

Abhängig vom Wert des Tagged-Values „EJBLocal“ erfolgt die Abbildung einer Klasse (**<NAME>**), die mit einem Stereotype gekennzeichnet ist, das direkt oder indirekt vom Stereotype «EJBHomeInterface» erbt, entweder („EJBLocal“ = „False“) auf das Codefragment:

```
import java.rmi.*;

...

interface <NAME> extends EJBHome {
```

```
}
```

oder („EJBLocal“ = „`True`“) auf das Codefragment:

```
...

interface <NAME> extends EJBLocalHome {
}
```

Deployment-Descriptor:

Abhängig vom Wert des Tagged-Values „EJBLocal“ erfolgt die Abbildung des voll qualifizierten Names einer Klasse (**<PACKAGE>.<NAME>**), die mit einem Stereotype gekennzeichnet ist, das direkt oder indirekt vom Stereotype «EJBHomeInterface» erbt, entweder („EJBLocal“ = „`False`“) auf das XML-Konstrukt:

```
<home><PACKAGE>.<NAME></home>
```

oder („EJBLocal“ = „`True`“) auf das XML-Konstrukt:

```
<local-home><PACKAGE>.<NAME></local-home>
```

7.4 Komponentenoperationen

Grundsätzlich können Komponentenoperationen wie folgt aufgeschlüsselt werden:

- Bezeichner der Methode (**<NAME>**),
- Rückgabetyp der Methode (**<RET>**),
- Typ des 1. Parameters, falls vorhanden (**<PARAM_1>**),
- Name des 1.Parameters, falls vorhanden (**<PNAME_1>**),
- Typ des N-ten Parameters, falls vorhanden (**<PARAM_N>**),
- Name des N-ten Parameters, falls vorhanden (**<PNAME_N>**).

7.4.1 «EJBRemoteMethod»

Java-Codefragmente:

Schnittstellenmethoden, die entweder direkt mit dem Stereotype «EJBRemoteMethod» gekennzeichnet sind oder mit einem Stereotype, das von diesem erbt, werden generell wie folgt innerhalb der betreffenden Schnittstelle abgebildet:

```
<RET> <NAME>(<PARAM_1> <PNAME_1>, ..., <PARAM_N> <PNAME_N>);
```

Falls es sich bei der zugehörigen Schnittstelle um eine entferntaufrufbare handelt, so muss jede dieser Methoden die Exception `java.rmi.RemoteException` in ihrer Throw-Klausel beinhalten.

Es erfolgt außerdem die parallele Abbildung auf eine Methodendefinition innerhalb der dazugehörigen Implementierungsklasse. Die hierfür geltende Abbildungsregel ist mit der des Stereotypes «EJBBeanMethod» identisch (siehe Abschnitt 7.4.5).

Deployment-Descriptor:

Schnittstellenmethoden, die entweder direkt mit dem Stereotype «EJBRemoteMethod» gekennzeichnet sind oder mit einem Stereotype, das von diesem erbt, können bei Bedarf auf das folgende XML-Konstrukt abgebildet werden (**<EJB>** bzw. **<TYPE>** stehten für die EJB bzw. die jeweilige Schnittstelle, in deren Kontext sich das XML-Konstrukt befindet):

```
<method>
    <ejb-name><EJB></ejb-name>
    <ejb-intf><TYPE></ejb-intf>
    <method-name><NAME></method-name>
    <method-params>
        <method-param><PARAM_1></method-param>
        ...
        <method-param><PARAM_N></method-param>
    </method-params>
</method>
```

Der angesprochene Bedarf ist immer dann gegeben, wenn bei der betroffenen Methode mindestens einem der beiden Tagged-Values „EJBTransAttribute“ oder „EJBRoleNames“ konkrete Werte zugeordnet sind und somit der Container eine Möglichkeit zur Identifikation dieser Methoden benötigt.

Teilt man alle Schnittstellenmethoden (**<METHOD_1>**, ..., **<METHOD_N>**), bezogen auf den Wert des Tagged-Values „EJBTransAttribute“ (**<VALUE>**), in Äquivalenzklassen ein, so wird jede einzelne Äquivalenzklasse jeweils auf ein Exemplar des folgenden XML-Konstrukts abgebildet:

```
<container-transaction>
    <METHOD_1>
    ...
    <METHOD_N>
    <trans-attribute>
        <VALUE>
    </trans-attribute>
</container-transaction>
```

Gleiches gilt für das Tagged-Value „EJBRoleNames“ (abzubildende Methoden = **<METHOD_1>**, ..., **<METHOD_N>**; einzelne Werte des Tagged-Values = **<VALUE_1>**, ..., **<VALUE_N>**). Die einzelnen Äquivalenzklassen dieses Tagged-Values werden dabei jeweils auf ein Exemplar des folgenden XML-Konstrukts abgebildet:

```
<method-permission>
    <role-name>
        <VALUE_1>
        ...
        <VALUE_N>
    </role-name>
    <METHOD_1>
    ...
    <METHOD_N>
</method-permission>
```

Dabei gilt es zu beachten, dass alle diejenigen Methoden (**<METHOD_1>**, ..., **<METHOD_N>**), die dieses Tagged-Value nicht definiert haben, schließlich auf das folgende XML-Konstrukt abgebildet werden:

```
<method-permission>
    <unchecked/>
    <METHOD_1>
    ...
    <METHOD_N>
</method-permission>
```

7.4.2 «EJBCreateMethod»

Java-Codefragmente:

Die Throw-Klausel einer mit dem Stereotype «EJBCreateMethod» gekennzeichneten Methode muss zusätzlich zu den eventuell schon vorhandenen fachlichen Exceptions die Exception `javax.ejb.CreateException` beinhalten.

Die in Abschnitt 7.4.1 angesprochene paralle Abbildung auf eine Implementierungsklassenmethode erfolgt bei einer mit dem Stereotype «EJBCreateMethode» gekennzeichneten Methode jedoch nach speziellen Abbildungsregeln.

Für Session-Beans wird eine ejbCreate-Methode des folgenden Typs generiert (der Parameterkopf entspricht dem der Schnittstellenmethode):

```
public void ejb<NAME>(...) {
}
```

Für Entity-Beans wird hingegen eine ejbCreate-Methode des folgenden Typs generiert (der Parameterkopf entspricht wiederum dem der Schnittstellenmethode; unter **<PRIMKEY>** ist die jeweils verwendete Primärschlüsselklasse gemeint):

```
public <PRIMKEY> ejb<NAME>(...) {
}
```

Außerdem wird in beiden Fällen zusätzlich noch die dazugehörige ejbPostCreate-Methode generiert:

```
public void ejbPost<NAME>(...) {
}
```

7.4.3 «EJBFindMethod»

Java-Codefragmente:

Die Throw-Klausel einer mit dem Stereotype «EJBFindMethod» gekennzeichneten Methode muss zusätzlich zu den eventuell schon vorhandenen fachlichen Exceptions die Exception `javax.ejb.FinderException` beinhalten.

Die in Abschnitt 7.4.1 angesprochene paralle Abbildung auf eine Implementierungsklassenmethode erfolgt, bei einer mit dem Stereotype «EJBFindMethode» gekennzeichneten Methode, nach ebenfalls speziellen Abbildungsregeln.

Die Abbildung auf eine ejbFind-Methode erfolgt zunächst grundsätzlich nur dann, wenn die Persistenzverwaltung der Entity-Bean obliegt („EJBPersistenceType“ = „`Bean`“).

Für den Fall, dass es sich dann bei der betreffenden Find-Methode um einen Single-Object-Finder handelt, erfolgt die Abbildung gemäß der folgenden Vorlage:

```
public <PRIMKEY> ejb<NAME>(...) {
}
```

Handelt es sich hingegen um einen Multi-Object-Finder, so erfolgt die Abbildung wie folgt:

```
public java.util.Collection ejb<NAME>(...) {
}
```

7.4.4 «EJBQueryMethod»

Deployment-Descriptor:

Der Wert (**<VALUE>**) des Tagged-Values „EJBQuery” wird auf das folgende XML-Konstrukt abgebildet:

```
<query>
    <query-method>
        <method-name><NAME></method-name>
        <method-params>
            <method-param><PARAM_1></method-param>
```

```
            ...
            <method-param><PARAM_N></method-param>
        </method-params>
    </query-method>
    <ejb-ql><VALUE></ejb-ql>
</query>
```

7.4.5 «EJBBeanMethod»

Java-Codefragmente:

Implementierungsklassenmethoden, die entweder direkt mit dem Stereotype «EJBBeanMethod» gekennzeichnet sind oder mit einem Stereotype, das von diesem erbt, werden generell wie folgt abgebildet:

```
public <RET> <NAME>(<PARAM_1> <PNAME_1>, ..., <PARAM_N> <PNAME_N>) {
}
```

7.4.6 «EJBSelectMethod»

Java-Codefragmente:

Die Throw-Klausel einer mit dem Stereotype «EJBSelectMethod» gekennzeichneten Methode muss zusätzlich zu den eventuell schon vorhandenen fachlichen Exceptions die Exception `javax.ejb.FinderException` beinhalten.

Deployment-Descriptor:

Zusätzlich zu der in Abschnitt 7.4.4 erläuterten Abbildung der jeweils spezifizierten EJB-QL-Anweisung, erfolgt außerdem eine Abbildung des als Rückgabewert vorgesehenen Schnittstellentyps (**<TYPE>**) und zwar auf das folgende XML-Konstrukt:

```
<query>
    <result-type-mapping><TYPE></result-type-mapping>
</query>
```

7.5 Komponentenrealisierung

7.5.1 «EJBImplementation»

Java-Codefragmente:

Eine Klasse, die mit dem Stereotype «EJBImplementation» versehen ist, beinhaltet grundsätzlich zunächst folgende Import-Anweisungen:

```
import javax.ejb.*;
import javax.jms.*;
import javax.naming.*;
```

Des Weiteren wird der Package-Pfad **<PACKAGE>** des betroffenen Klassennames auf eine Package-Anweisung vor der eigentlichen Klassendefinition abgebildet:

```
package <PACKAGE>;
```

Schließlich beinhaltet die jeweilige Implementierungsklasse auch grundsätzlich die folgende Instanzmethode:

```
public void ejbRemove() throws EJBException {
}
```

Deployment-Descriptor:

Grundsätzlich wird der vollständig qualifizierte Name (**<PACKAGE>.<NAME>**) einer Implementierungsklasse auf das folgende XML-Konstrukt abgebildet:

```
<ejb-class><PACKAGE>.<NAME></ejb-class>
```

7.5.1.1 «EJBSessionBean»

Java-Codefragmente:

Handelt es sich bei der betreffenden EJB um eine Session-Bean, so erfolgt sie Abbildung der mit dem Stereotype «EJBImplementation» gekennzeichnete Klasse (**<NAME>**) speziell wie folgt:

```
...

public class <NAME> implements SessionBean {

    private transient SessionContext ctx;

    public <NAME>() {
    }

    public void ejbActivate() throws EJBException {
    }

    public void ejbPassivate() throws EJBException {
    }

    public void setSessionContext(
            SessionContext ctx) throws EJBException {

        this.ctx = ctx;
    }

   ...
}
```

7.5.1.2 «EJBEntityBean»

Java-Codefragmente:

Handelt es sich bei der betreffenden EJB um eine Entity-Bean, so erfolgt sie Abbildung der mit dem Stereotype «EJBImplementation» gekennzeichnete Klasse (**<NAME>**), abhängig vom Wert des Tagged-Values „EJBPersistenceType“, entweder zunächst auf die folgende Kopfzeile („EJBPersistenceType“ = „Container“):

```
...

public abstract class <NAME> implements EntityBean {
```

Oder die Abbildung erfolgt auf die dazu alternative Kopfzeile („EJBPersistenceType“ = „Bean“):

```
...

public class <NAME> implements EntityBean {
```

Unabhängig vom Wert des Tagged-Values „EJBPersistenceType” sieht der Klassendefinitionsrumpf der Entity-Bean-bezogenen Implementierungsklassen wie folgt aus:

```
    private transient EntityContext ctx;

    public <NAME>() {
    }

    public void ejbActivate() throws EJBException {
    }

    public void ejbLoad() throws EJBException {
    }

    public void ejbPassivate() throws EJBException {
    }

    public void ejbStore() throws EJBException {
    }

    public void setEntityContext(
            EntityContext ctx) throws EJBException {

        this.ctx = ctx;
    }

    public void unsetEntityContext() throws EJBException {
    }

   ...
}
```

7.5.1.3 «EJBMessageDrivenBean»

Java-Codefragmente:

Handelt es sich bei der betreffenden EJB um eine Message-Driven-Bean, so erfolgt sie Abbildung der mit dem Stereotype «EJBImplementation» gekennzeichnete Klasse (**<NAME>**) speziell wie folgt:

```
...
public class <NAME> implements MessageDrivenBean, MessageListener {

    private transient MessageDrivenContext ctx;

    public <NAME>() {
    }

    public void setMessageDrivenContext(MessageDrivenContext ctx)
            throws EJBException {

        this.ctx = ctx;
    }

    public void onMessage(Message message) {
    }

   ...
}
```

7.6 Komponentenrelationen

Bei der Abbildung von Relationen zwischen EJBs wird zwischen zwei unterschiedlichen Assoziationstypen unterschieden:

1. Assoziationen zwischen EJBs, die mittels EJB-Referenzen umgesetzt werden.
2. Persistente Assoziationen zwischen Entity-Beans, die mittels der CMP umgesetzt werden und die zur Unterscheidung von EJB-Referenzen-basierten Assoziationen mit dem Stereotype «EJBCMRelationship» gekennzeichnet sind.

Grundsätzlich lassen sich beide Assoziationstypen in folgende, an die UML-Semantik angelegte, Eigenschaften aufschlüsseln:

- Name der Assoziation (**<ASS>**),
- Name des 1. Assoziationsendes (**<ROL1>**),
- Partizipant des 1. Assoziationsendes (**<PAR1>**),

- Name des 2. Assoziationsendes (**<ROL2>**),
- Partizipant des 2. Assoziationsendes (**<PAR2>**).

Da im Rahmen eines Entwurfsmodells den beteiligten Assoziationsenden in der Regel Multiplizitäten zugeordnet werden, müssen diese auch bei der partiellen Codeerzeugung berücksichtigt werden. Grundsätzlich gilt es daher die folgenden Multiplizitäten zu berücksichtigen:

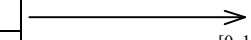
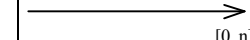
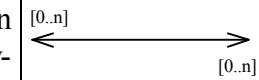
- Multiplizität des 1. Assoziationsendes (**<MUL1>**)
- Multiplizität des 2. Assoziationsendes (**<MUL2>**)

Dabei werden die Multiplizitäten je nach Art der Codeerzeugung (**Java-Codefragmente** oder **Deployment-Descriptor**) jeweils spezifisch abgebildet.

7.6.1 EJB-Referenzen-basierte Assoziationen

Bei Assoziationen, die mittels EJB-Referenzen implementiert werden, muss, basierend auf den Ergebnissen des zweiten Kapitels (siehe Abschnitt 2.5), zwischen vier unterschiedlichen Assoziationstypen unterschieden werden:

Tabelle 7 – EJB-Referenzen-basierte Assoziationstypen (n , beliebig, fest Þ "n: 1 < n <= *)

Typ	Beschreibung	UML-Notation
1a	Unidirektionale Assoziation mit einer maximalen Multiplizität von „1“ die als Supplier eine Session- oder Entity-Bean besitzt.	
1b	Unidirektionale Assoziation mit einer maximalen Multiplizität von „1“ die als Supplier eine Message-Driven-Bean besitzt.	
2	Unidirektionale Assoziation mit beliebiger Multiplizität die als Supplier eine Entity-Bean besitzt.	
3	Bidirektionale Assoziation mit beliebigen Multiplizitäten, deren Clients zustandsbehaftete Session-Beans sind und deren Supplier sowohl zustandsbehaftete Session- als auch Entity-Beans sein können.	

Der vollständig qualifizierte Name der Component-Schnittstelle der jeweils assoziierten EJB soll unter **<COMPONENT>** angesprochen werden, der vollständig qualifizierte Name der betreffenden Home-Schnittstelle soll unter **<HOME>** angesprochen werden und unter **<TYPE>** ist der jeweilige EJB-Typ gemeint („`Session`“ oder „`Entity`“).

Das Template **<NAME>** bezeichnet den Namen der jeweiligen Component-Schnittstelle ohne den Package-Pfad und das Template **<EJBNAME>** steht für den Bezeichner der assoziierten EJB innerhalb des betreffenden EJB-Jars.

7.6.1.1 Assoziationstyp 1a

Java-Codefragmente:

Bei der Codegenerierung werden im Rahmen der betroffenen Implementierungsklasse jeweils zwei Instanzattribute angelegt, mit deren Hilfe auf das zur Laufzeit bestehende Home- bzw. Component-Objekt zugegriffen werden kann:

```
private transient <HOME> ejbHomeRef<NAME>;
private transient <COMPONENT> ejbRef<NAME>;
```

Des Weiteren wird eine Instanzmethode definiert, die zur Laufzeit das zuvor definierte Instanzattribut bzgl. des Home-Objekts mit einer gültigen Referenz initialisiert:

```
private void get<NAME>Home() {

    try {
        Context initCtx = new InitialContext();
        Object result = initCtx.lookup("java:comp/env/ejb/<NAME>");
```

Die vorletzte Anweisung dieser Methode ist dabei abhängig vom Typ der Home-Schnittstelle. Bei lokalaufrufbaren Home-Schnittstellen reicht ein normaler Typ-Cast:

```
        this.ejbHomeRef<NAME> = (<HOME>)result;
```

Bei einer entferntaufrufbaren Home-Schnittstellen ist folgende Anweisung vorgesehen:

```
        this.ejbHomeRef<NAME> = (<HOME>)
                PortableRemoteObject.narrow(result, <HOME>.class);
```

Die letzte Abschnitt der Methode ist für beide Fälle wieder identisch:

```
    } catch(java.lang.Exception e) {
    }
}
```

Abschließend wird für jede auf Seiten der assoziierten Session- oder Entity-Bean vorkommenden Create-Methode und, falls vorhanden, für jede Find-Methode, die ein Single-Object-Finder ist, ein entsprechendes Gegenstück in der betroffenen Implementierungsklasse generiert.

Hinweis:

Grundsätzlich werden bei EJB-Referenzen-basierte Assoziationen, die über die Multiplizität ‚1' auf Seiten des Suppliers verfügen, nur GetByFind-Methoden für sogenannte Single-Object-Finder generiert. Dies hängt mit der relativen Schwierigkeit zusammen, für Multi-Object-Finder ein Auswahlkriterium im Rahmen des Generierungsprozesses zu definieren, dass eine eindeutige Rückgabe im Rahmen der jeweiligen GetByFind-Methode ermöglicht.

So lautet die Kopfzeile der zu generierenden GetByCreate-Methode wie folgt:

```
private void get<NAME>ByCreate<SUFFIX>(<PARAMETERS>) {
```

Handelt es sich bei der assoziierten Schnittstelle um eine entferntaufrufbare, so lautet der Methodenrumpf:

```
    this.ejbRef<NAME> = (<COMPONENT>)PortableRemoteObject.narrow(
            this.ejbHomeRef<NAME>.create<SUFFIX>(<PARAMETERS>),
            <COMPONENT>.class);
}
```

Handelt es sich bei der assoziierten Schnittstelle jedoch um eine lokalaufrufbare, so lautet der Methodenrumpf dementsprechend:

```
    this.ejbRef<NAME>
            = this.ejbHomeRef<NAME>.create<SUFFIX>(<PARAMETERS>);
}
```

Die gleiche Unterscheidung für die GetByFind-Methoden – zunächst für beide Fälle gültige Methodenkopf:

```
private void get<NAME>ByFind<SUFFIX>(<PARAMETERS>) {
```

Der Methodenrumpf bei einer entferntaufrufbaren Schnittstelle:

```
    this.ejbRef<NAME> = (<COMPONENT>)PortableRemoteObject.narrow(
            this.ejbHomeRef<NAME>.find<SUFFIX>(<PARAMETERS>),
            <COMPONENT>.class);
}
```

Der Methodenrumpf bei einer lokalaufrufbaren Schnittstelle:

```
    this.ejbRef<NAME>
            = this.ejbHomeRef<NAME>.find<SUFFIX>(<PARAMETERS>);
}
```

Deployment-Descriptor:

Die Abbildung auf das dafür vorgesehene XML-Konstrukt erfolgt abhängig davon, ob die Component-Schnittstellentyp der assoziierten EJB lokal- oder entferntaufrufbar ist.

Handelt es sich bei der referenzierten Schnittstelle um eine lokalaufrufbare Component-Schnittstelle, so erfolgt die Abbildung auf das folgende XML-Konstrukt:

```
<ejb-local-ref>
    <ejb-ref-name>ejb/<NAME></ejb-ref-name>
    <ejb-ref-type><TYPE></ejb-ref-type>
    <local-home><HOME></local-home>
    <local><COMPONENT></local>
    <ejb-link><EJBNAME></ejb-link>
</ejb-local-ref>
```

Handelt es sich jedoch bei der referenzierten Schnittstelle um eine entferntaufrufbare Component-Schnittstelle, so erfolgt die Abbildung dementsprechen auf das folgende XML-Konstrukt:

```
<ejb-ref>
    <ejb-ref-name>ejb/<NAME></ejb-ref-name>
    <ejb-ref-type><TYPE></ejb-ref-type>
    <home><HOME></home>
    <remote><COMPONENT></remote>
    <ejb-link><EJBNAME></ejb-link>
</ejb-ref>
```

7.6.1.2 Assoziationstyp 1b

Java-Codefragmente:

Für diesen Abschnitt gilt zusätzlich die Namenskonvention, dass sich das Template **<TYPE>** auf den Wert des gleichnamigen Bestandteils des Tagged-Values „EJBDestination“ der jeweils assoziierten Message-Driven-Bean bezieht.

Zunächst werden in der EJB, von der die Assoziation ausgeht, zwei Instanzattribute generiert, mit deren Hilfe später JMS-Nachrichten an die ‚assoziierte’ Message-Driven-Bean geschickt werden können:

```
private transient <TYPE> destOf<NAME>;
private transient <TYPE>Session sessionOf<NAME>;
```

Schließlich wird im selben Zusammenhang eine Instanzmethode generiert, die den oben eingeführten Instanzattributen jeweils eine zur Laufzeit gültige Objektreferenz zuweist:

```
private void create<NAME>Session() {

    try {
```

```
        Context initCtx = new InitialContext();
        Context myEnv = (Context)initCtx.lookup("java:comp/env/");
        this.destOf<NAME> = (<TYPE>)
                myEnv.lookup("jms/destOf<NAME>");
        <TYPE>ConnectionFactory factory = (<TYPE>ConnectionFactory)
                initCtx.lookup(jms/factoryOf<NAME>);
        <TYPE>Connection connection
                = factory.create<TYPE>Connection();
        this.sessionOf<NAME> = connection.create<TYPE>Session(
                false, Session.AUTO_ACKNOWLEDGE);
    } catch(java.lang.Exception e) {
    }
}
```

Deployment-Descriptor:

Schließlich gilt es zu beachten, dass im Rahmen des Deployment-Descriptors die jeweils zu verwendende Conncetion-Factory und Destination gemäß den dafür vorgesehenen Abbildungsregelen (siehe Abschnitt 7.2.1) definiert wird:

```
<resource-ref>
    <res-ref-name>jms/factoryOf<NAME></res-ref-name>
    <res-type>javax.jms.<TYPE>ConnectionFactory</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
```

```
<resource-env-ref>
    <resource-env-ref-name>jms/destOf<NAME></resource-env-ref-name>
    <resoruce-env-ref-type>javax.jms.<TYPE></resoruce-env-ref-type>
</resource-env-ref>
```

7.6.1.3 Assoziationstyp 2

Java-Codefragmente:

Im Rahmen dieses Assoziationstyps können nun beliebig viele Examplare einer Entity-Bean unidirektional assoziiert werden (Multiplizität [0..n]).

Daher wird im Gegensatz zum Assoziationstyp 1a ein Instanzattribut generiert, dass Kollektionen von Component-Schnittstellen referenzieren kann:

```
private transient java.util.Collection ejbRefs<NAME>;
```

Eine Weitere Änderung betrifft die Generierung der GetByCreate- bzw. GetByFind-Methoden – sie liefern nun ein Component-Objekt der jeweils assoziierten Entity-Bean zurück (im Falle der Create-Methoden sowie Single-Object-Finder) oder. direkt Kollektionen von Component-Objekten (im Falle von Multi-Object-Finder):

```
private <COMPONENT> get<NAME>ByCreate<SUFFIX>(<PARAMETERS>) {

    return this.ejbHomeRef<NAME>.create<SUFFIX>(<PARAMETERS>);
```

```
}
```

```
private <COMPONENT> get<NAME>ByFind<SUFFIX>(<PARAMETERS>) {

    return this.ejbHomeRef<NAME>.find<SUFFIX>(<PARAMETERS>);
}
```

```
private java.util.Collection get<NAME>ByFind<SUFFIX>(<PARAMETERS>) {

    return this.ejbHomeRef<NAME>.find<SUFFIX>(<PARAMETERS>);
}
```

Deployment-Descriptor:

Es gelten die Abbildungsvorschriften aus Abschnitt 7.6.1.1.

7.6.1.4 Assoziationstyp 3

Java-Codefragmente:

Dieser Assoziationstyp umfasst bidirektionale Assoziationen bei denen auf Seiten des Suppliers sowohl zustandsbehaftete Session- als auch Entity-Beans vorkommen können, deren Client aber in jedem Fall eine zustandsbehaftete Session-Bean ist.

Grundsätzlich gilt für die Abbildung dieses Assoziationstyps, dass sie symmetrisch, also auf beiden Seiten identisch, erfolgt. Dies vereinfach zunächst die Abbildungsvorschrift.

Später ist dann im Rahmen der manuellen Codeergänzung zu entscheiden, welcher der beiden EJBs diejenige ist, die die bidirektionale Assoziation zur Laufzeit aufbaut.

Demnach wird zunächst auf beiden Seiten ein Instanzattribut gemäß Abschnitt 7.6.1.1 generiert, dass zur Referenzierung der jeweils assoziierten Home-Schnittstelle dient. Des Weiteren wird ein Instanzattribut gemäß Abschnitt 7.6.1.3 generiert, dass Kollektionen von Referenzen auf die entsprechenden Component-Objekte speichern kann.

Um die Bidirektionalität der Assoziation zur Laufzeit gewährleisten zu können, werden modifizierte GetByCreate- bzw. GetByFind-Methoden generiert, die das jeweils eigene Component-Objekt an die referenzierende EJB übergeben.

Der erste Teil dieser modifizierten Methode gleicht noch dem ursprünglichen Vorlage (hier am Beispiel einer GetByCreate-Methode):

```
private <COMPONENT> get<NAME>ByCreate<SUFFIX>(<PARAMETERS>) {

    <COMPONENT> obj = this.ejbHomeRef<NAME>.create<SUFFIX>(
            <PARAMETERS>);
```

Unter der Voraussetzung das die eigene Component-Schnittstelle lokalaufrufbar ist, lautet die vorletzte Anweisung dieser Methode jedoch wie folgt:

```
    obj.add<NAME>EjbRef(this.ctx.getEJBLocalObject());
```

Ist die eigene Component-Schnittstelle dagegen entferntaufrufbar, so lautet die vorletzte Anweisung stattdessen:

```
    obj.add<NAME>EjbRef(this.ctx.getEJBObject());
```

Die letzte Abschnitt dieser Methode ist hingegen bei beiden Varianten wieder identisch:

```
    return obj;
}
```

Damit ein solcher ‚Rückruf' stattfinden kann, bedarf es noch einer entsprechenden Instanzmethode in beiden Implementierungsklassen.

Diese registriert die übergebene Referenz auf das jeweilige Component-Objekt bei dem dafür vorgesehenen Instanzattribut.

Handelt es sich bei der übergebenen Referenz um eine lokalaufrufbare Component-Schnittstelle, so lautet die Kopfzeile dieser Methode:

```
public void add<NAME>EjbRef(EJBLocalObject obj) (
```

Handelt es sich jedoch um eine entferntaufrufbare Component-Schnittstelle, so lautet die Kopfzeile stattdessen:

```
public void add<NAME>EjbRef(EJBObject obj) (
```

Der Methodenrumpf ist hingegen bei beiden Varianten identisch:

```
    if (this.ejbRefs_<NAME> == null) {
        this.ejbRefs_<NAME> = new java.util.Collection();
    }
    this.ejbRefs_<NAME>.add(obj);
)
```

Deployment-Descriptor:

Es gelten die Abbildungsvorschriften aus Abschnitt 7.6.1.1.

7.6.2 «EJBCMRelationship» und «EJBCMPAssociationEnd»

Je nach Art der Navigierbarkeit der Assoziation, also uni- oder bidirektional navigierbar, sind in den jeweils an der Assoziation beiteiligten Entity-Beans sogenannte CMR-Felder

vorgesehen. Die Namensgebung dieser Felder ist im Rahmen der CMP jedoch fest vorgegeben. Der CMR-Feldbezeichner ist mit dem Namen der Component-Schnittstelle der jeweils assoziierten Entity-Bean identisch, wobei die Konvention gilt, dass der erste Buchstabe des CMR-Feldbezeichners grundsätzlich klein geschrieben wird.

Folgende Templates werden für die CMR-Feldbezeichner eingeführt:

- Bei einer unidirektionalen Assoziation ist **<EJB>** als Template eindeutig.
- Bei einer bidirektionalen Assoziation gelten jeweils **<EJB1>** und **<EJB2>** als eindeutige Templates.

Die beiden in Abschnitt 7.6 eingeführten Templates für die verschiedenen Multiplizitäten (**<MUL1>**, **<MUL2>**) werden im Rahmen des Deployment-Descriptors wie folgt abgebildet:

Tabelle 8 - Abbildung von UML-Multiplizitäten auf die Deployment-Descriptor-Notation

UML-Notation	Deployment-Descriptor-Notation	
	<multiplicity>	<cmr-field-type>
[0..1] oder [1]	„One“	-
[0..n], wobei 1 < n <= *	„Many“	„java.util.Collection“

Schließlich steht das Template **<TYPE>**, abhängig von der Multiplizität der Assoziation, entweder für

- den vollständig qualifizierten Namen der jeweils assoziierten Component-Schnittstelle (Multiplizität [0..1] oder [1])

oder für

- die Java-Klasse „java.util.Collection“ (Multiplizität [0..n]).

Java-Codefragmente:

Grundsätzlich werden Assoziationen, die mit dem Stereotype «EJBCMRelationship» gekennzeichnet sind, abhängig von der Navigierbarkeit des jeweiligen Assoziationsendes auf abstrakte Getter-/Setter-Methoden der Implementierungsklasse der jeweils beteiligten Entity-Beans abgebildet:

```
public abstract <TYPE> get<EJB>();
public abstract void set<EJB>(<TYPE>);
```

Abhängig vom Wert des Tagged-Values „EJBAccessModifier“ werden die zuvor in der Implementierungsklasse eingeführten Getter-/Setter-Methoden auch in den lokalaufrufbaren Component-Schnittstellen der betroffenen Entity-Beans veröffentlicht:

```
<TYPE> get<EJB>();
void set<EJB>(<TYPE>);
```

Deployment-Descriptor:

Generell werden Assoziationen, die mit dem Stereotype «EJBCMRelationship» versehen sind und die ihrerseits zwei mit dem Stereotype «EJBCMPAssociationEnd» gekennzeichnete Assoziationsenden beinhalten, auf das folgende XML-Konstrukt abgebildet, wobei das XML-Element `<relationships/>` pro Deployment-Descriptor insgesamt nur einmal definiert wird:

```
<relationships>
    <ejb-relation>
        <ejb-relation-name><ASS></ejb-relation-name>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                <ROL1>
            </ejb-relationship-role-name>
            <multiplicity><MUL1></multiplicity>
            <relationship-role-source>
                <PAR1>
            </relationship-role-source>
        </ejb-relationship-role>
        <ejb-relationship-role>
            <ejb-relationship-role-name>
                <ROL2>
            </ejb-relationship-role-name>
            <multiplicity><MUL2></multiplicity>
            <relationship-role-source>
                <PAR2>
            </relationship-role-source>
        </ejb-relationship-role>
    <ejb-relation/>
</relationships>
```

Abhängig von der Navigierbarkeit der jeweiligen Assoziation, müssen noch die dafür verwendeten CMR-Felder im Rahmen des Deployment-Descriptors definiert werden:

Bei unidirektional navigierbaren Assoziationen wird das folgende XML-Konstrukt für das Assoziationsende verwendet, dass durch das CMR-Feld dargestellt wird (**<MUL_X>** steht für die jeweils ausschlaggebende Multiplizität, also entweder **<MUL1>** oder **<MUL2>**):

```
<ejb-relationship-role>
    ...
    <cmr-field>
        <cmr-field-name><EJB></cmr-field-name>
        <cmr-field-type><MUL_X></cmr-field-type>
```

```
    </cmr-field>
</ejb-relationship-role>
```

Bei bidirektional navigierbaren Assoziationen wird das folgende XML-Konstrukt für die Relation verwendet:

```
<ejb-relation>
    <ejb-relationship-role>
        <cmr-field>
            <cmr-field-name><EJB1></cmr-field-name>
            <cmr-field-type><MUL1></cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
    <ejb-relationship-role>
        <cmr-field>
            <cmr-field-name><EJB2></cmr-field-name>
            <cmr-field-type><MUL2></cmr-field-type>
        </cmr-field>
    </ejb-relationship-role>
<ejb-relation>
```

Falls in einem der beiteiligten Assoziationsenden das Tagged-Value „EJBCascadeDelete“ definiert ist, so muss es jeweils noch durch das folgende XML-Konstrukt berücksichtigt werden:

```
<ejb-relationship-role>
    <cascade-delete/>
</ejb-relationship-role>
```

Kapitel 8
Generatorkonzept

Nachdem im vorangegangenen Kapitel die für die Generierung der EJB-Codeartefakte notwendigen Abbildungsregeln definiert wurden, wird in diesem Kapitel eine mögliche Umsetzung dieser mittels eines prototypischen Generatorkonzepts vorgestellt.

Dazu gehört auch der prinzipielle Funktionsnachweis anhand zweier einfacher Beispiele.

Das Kapitel beginnt zunächst mit der Formulierung der für alle Generatorkonzepte gleichermaßen geltenden Anforderungen.

Im Anschluß daran werden zwei grundsätzlich unterschiedliche Generatorkonzepte erläutert, wobei eines davon im Rahmen dieses Kapitels weitere Anwendung findet.

Des Weiteren werden zwei beispielhafte, auf dem erweiterten Profile basierende EJB-Modelle beschrieben, die auch die Grundlage des erwähnten Funktionsnachweises darstellen.

Abschließend erfolgt eine Bewertung des gewählten Generatorkonzepts.

8.1 Anforderungen

Basierend auf der dieser Arbeit zugrundeliegenden Aufgabenstellung müssen folgende Anforderungen im Rahmen eines Generatorkonzepts erfüllt sein:

1. Als Eingabedaten sind UML-Klassendiagramme vorgesehen, die auf der erweiterten Profile-Fassung (Kapitel 6) basieren und die als XMI-1.1-konforme XML-Dateien vorliegen müssen.
2. Die eigentliche durch das Generatorkonzept zu implementierende Transformation der Eingabedaten muss den in Kapitel 7 definierten Abbildungsregeln entsprechen.
3. Die zu generierenden Ausgabedaten müssen den EJB-Codeartefakten aus Kapitel 7 entsprechen, d.h. es muss der jeweilige Deployment-Descriptor samt den zugehörigen Java-Klassendefinitionen generiert werden.

8.2 Generatorkonzepte

Um den in Abschnitt 8.1 formulierten Anforderungen zu entsprechen, können grundsätzlich zwei verschiedene Generatorkonzepte angewandt werden:

1. Anwendung eines objektorientierten Generatorkonzepts.
2. Anwendung eines deklarativen Generatorkonzepts.

8.2.1 Objektorientiertes Generatorkonzept

Unter der Anwendung eines objektorientierten Generatorkonzepts ist die Nutzung einer objektorientierten Programmiersprache zur Umsetzung der in Abschnitt 8.1 formulierten Anforderungen zu verstehen.

Dies könnte z.B. unter Verwendung der Programmiersprache Java geschehen, indem dort auf die schon vorhandenen Konzepte der Miteinbeziehung von XMI-basierten Metamodellen zurückgegriffen wird [SUN02b, IBM00].

Die wesentlichen Vorteile dieses Ansatzes liegen in der enormen Flexibilität solcher Generatorkonzepte und in der durch die Objektorientierung gewährleisteten Erweiterbarkeit und Anpaßbarkeit der entwickelten Architektur.

Als nachteilig erweist sich jedoch der relative Aufwand, der mit der eigenen Entwicklung von den, zu diesem Generatorkonzept dazugehörigen Komponenten (z.B. der Parser oder die für die Syntaxbaumtransformation zuständige Komponente) verbunden ist.

Da gemäß der Aufgabenstellung lediglich ein prototypisches Generatorkonzept entworfen werden soll, wird dieser Ansatz nicht weiter verfolgt.

8.2.2 Deklaratives Generatorkonzept

Unter der Anwendung eines deklarativen Generatorkonzepts ist die Nutzung einer deklarativen Sprache zur Umsetzung der in Abschnitt 8.1 formulierten Anforderungen zu verstehen.

Da in diesem Fall die Eingabedaten als XMI-1.1-konforme Datei vorliegen, bietet sich die Nutzung der deklarativen Sprache XSLT [WWW10] an. Diese Sprache ermöglicht die

Formulierung von Regeln, auf deren Basis XML-Dateien in beliebige Zielformate mittels XSLT-fähiger Stylesheet-Prozessoren tranformiert werden können [SC01].

Der wesentliche Vorteil von deklarativen Generatorkonzepten liegt in der relativ einfachen Problembeschreibung. Verglichen mit objektorientierten Programmiersprachen wie z.B. Java und C++, mit denen normalerweise eher maschinennahe Konzepte implementiert werden, arbeiten deklarative Sprachen weniger auf der Implementierungsebene, sondern konzentrieren sich auf die Beschreibung eines Problems. So kann bei diesen Sprachen in der Regel eine Notation verwendet werden, die sehr nahe an der konzeptionellen Beschreibung des zu bearbeitenden Wissensgebietes liegt.

In diesem Fall erübrigt sich die sonst notwendige Implementierung von eigenen Parsern oder anderen notwendigen Generatorkomponenten, da auf vorhandene XSLT-fähige Stylesheet-Prozessoren zurückgegriffen werden kann.

Der wesentliche Nachteil dieses Konzepts ist die relative Komplexität und die damit einhergehende Unübersichtlichkeit dieses Ansatzes im Vergleich zum objektorientierten Vorgehen.

Da in der Sprache XSLT nur sehr wenige und dabei auch nur sehr einfach strukturierte programmiersprachenähnliche Schleifenkonstrukte existieren, muss dieses Fehlen durch die komplexe und unter Umständen auch stark verschachtelte Anwendung der vorhandenen Sprachkonstrukte kompensiert werden.

Dies führt in der Regel auch zu sehr komplexen Transformationsregeln, die sich später, im Hinblick auf eine eventuell anstehende Veränderung der Zielsprache, nur sehr schwer anpassen bzw. erweitern lassen.

Dennoch soll für den weiteren Verlauf dieses Kapitels das deklarative Generatorkonzept als Ausgangspunkt gewählt werden, da es eine sehr pragmatische Umsetzung der in Kapitel 7 definierten Abbildungsregeln erlaubt.

8.3 Verwendetes Generatorkonzept

Das in diesem Kapitel verwendete Generatorkonzept basiert im Grundsatz auf zwei XSLT-Skripten, die unter Anwendung eines XSLT-fähigen Stylesheet-Prozessors die Transformation von Profile-basierten EJB-Modellen zu EJB-Codeartefakte ermöglichen.

Dabei beinhaltet das erste XSLT-Skript die Beschreibung derjenigen Regeln, die für die Generierung der Java-basierten Klassendefinitionen (Java-Codefragmente) notwendig sind. Das zweite XSLT-Skript ist für die Generierung des Deployment-Descriptors zuständig.

Eine Trennung in zwei unterschiedlich zuständige XSLT-Skripte ist nicht nur aufgrund der in Kapitel 7 verwendeten Gliederung naheliegend, sondern ergibt sich auch aus der unterschiedlichen Zielsyntax der beiden zu generierenden Ausgangsformate:

- Die Java-Codefragmente basieren, wie der verwendete Name schon impliziert, auf der in [GJSB00] spezifizierten Programmiersprache Java 2.
- Der Deployment-Descriptor basiert demgegenüber auf einer in [SUN01, Seite 455ff.] definierten Teilmenge der Beschreibungssprache XML.

8.3.1 Generierung der Java-Codefragmente

In diesem und im nächsten Abschnitt wird jeweils nur der prinzipielle Aufbau des betreffenden XSLT-Skripts erläutert; ein ausführlicher Einblick ist nur unter Zuhilfenahme des beiligenden elektronischen Datenträges möglich, auf dem beide Skripte gespeichert sind.

Die Beschreibung der für die Generierung der Java-Codefragmente zuständigen Transformationsregeln orientiert sich am prinzipiellen Aufbau einer EJB.

Für jede im Modell definierte EJB wird demnach, gemäß den in Kapitel 7 definierten Abbildungsregeln, jeweils eine Implementierungsklasse sowie, falls vorhanden, die von der Komponente angebotene Schnittstellenlogik generiert.

Da die zugrundeliegende Profile-Architektur über ein relativ hohes Abstraktionsniveau verfügt (siehe Abschnitt 4.3.1) existiert zwischen den einzelnen Transformationsregeln eine hohe Anzahl von Abhängigkeiten. Die einzelnen Abhängigkeiten werden im Rahmen des XSLT-Skriptes mittels eines Call-Mechanismus, der vergleichbar mit der Methodenaufrufsemantik der Programmiersprache Java ist, umgesetzt.

Daher ist es auch möglich, wiederkehrende Aufgaben, wie z.B. das Auslesen eines bestimmten Tagged-Values, mittels methodenähnlicher Templates zusammenfassend zu beschreiben. Dieses Konzept ermöglicht auch XSLT-basierte Frameworks, die an dieser Stelle aber nicht weiter vertieft werden (siehe dazu [Ober00]).

Das Ergebnis einer Java-Codefragmente-bezogenen Transformation ist eine einzelne Ausgabedatei im Plain-Text-Format. Sie beinhaltet alle durch das jeweilige UML-Modell beschriebenen EJB-Klassen- und Schnittstellendefinitionen; diese sind innerhalb der Datei durch Kommentare voneinander getrennt.

8.3.2 Generierung des Deployment-Descriptors

Die Beschreibung der für die Generierung des Deployment-Descriptors zuständigen Transformationsregeln orientiert sich in erster Hinsicht an den in Kapitel 7 dafür vorgesehenen Abbildungsregeln und in zweiter Hinsicht an der die Zielsyntax vorgebenden EJB-2.0-Document-Type-Definition (DTD); letztere wird im Rahmen der EJB-2.0-Spezifikation beschrieben.

Das Ergebnis dieser Transformation ist eine auf dem jeweiligen UML-Modell basierende Deployment-Descriptor-Datei im dafür vorgesehenen XML-Format.

8.4 Beispielmodelle

Um die prinzipielle Funktionsweise des zuvor vorgestellten Generatorkonzepts nachweisen zu können, wird in den folgenden zwei Abschnitten jeweils ein einfaches, auf dem erweiterten Profile basierendes, EJB-Klassenmodell beschrieben.

Beide Modelle dienen dem, in Abschnitt 8.5 zu erbringenden Funktionsnachweis als Grundlage.

8.4.1 Einfache Entity-Bean

Das erste Beispiel beinhaltet ein EJB-Klassenmodell, dass nur aus einer einzigen Entity-Bean besteht.

Diese Entity-Bean, die den Namen „OrderEJB“ trägt, verfügt über eine Komponentenspezifikation, über zwei Komponentenschnittstellen samt dort definierter (Komponenten-)Operationen sowie über eine der Komponentenrealisierung dienenden Implementierungsklasse.

Im Rahmen der Home-Schnittstelle bietet die Entity-Bean lediglich eine parameterlose Create- sowie die vorgegebene `findByPrimaryKey(...)`-Methode an.

Als einziges CMP-Feld verfügt sie über einen Integer-Wert, der gleichzeitig auch den Primärschlüssel der Entity-Bean darstellt.

Die zugehörige Primärschlüsselklasse ist dementsprechend vom Typ Integer und wurde aus dem Package „java::lang“ in den Spezifikationsteil des Subsystems importiert.

Die Entity-Bean befindet sich im Package „da::example1“, das sich selber im Package „EntityBeanJAR“ befindet. Letzteres stellt das, die Entity-Bean umgebende, EJB-Jar-Konstrukt dar.

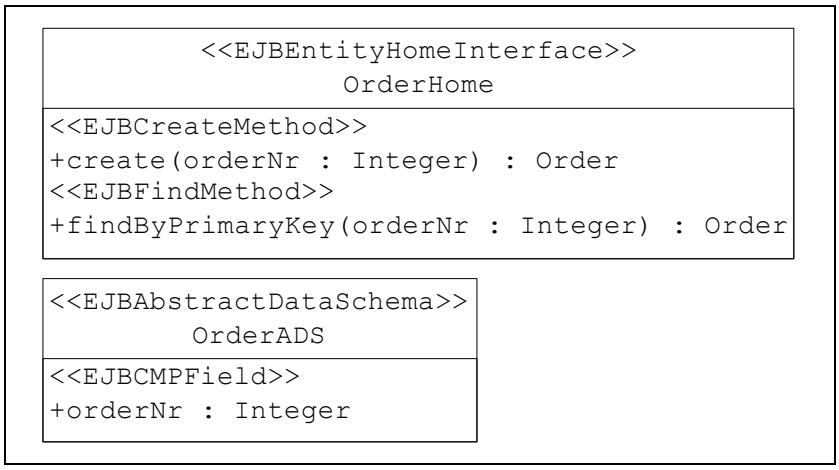


Abbildung 14 - Home-Schittstelle und abstraktes Datenschema der Entity-Bean

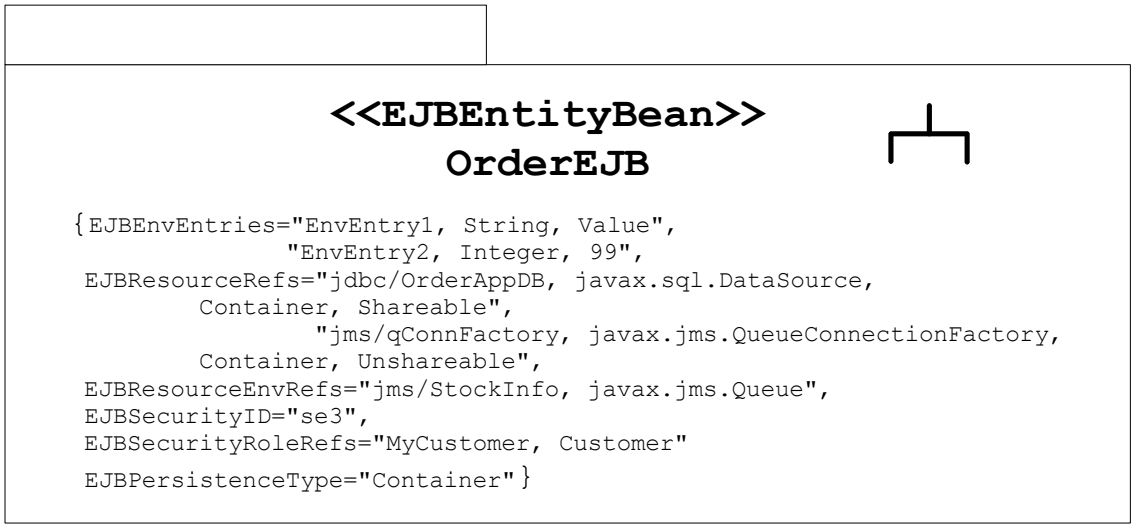


Abbildung 15 - In der Komponentenspezifikation definierte Eigenschaften

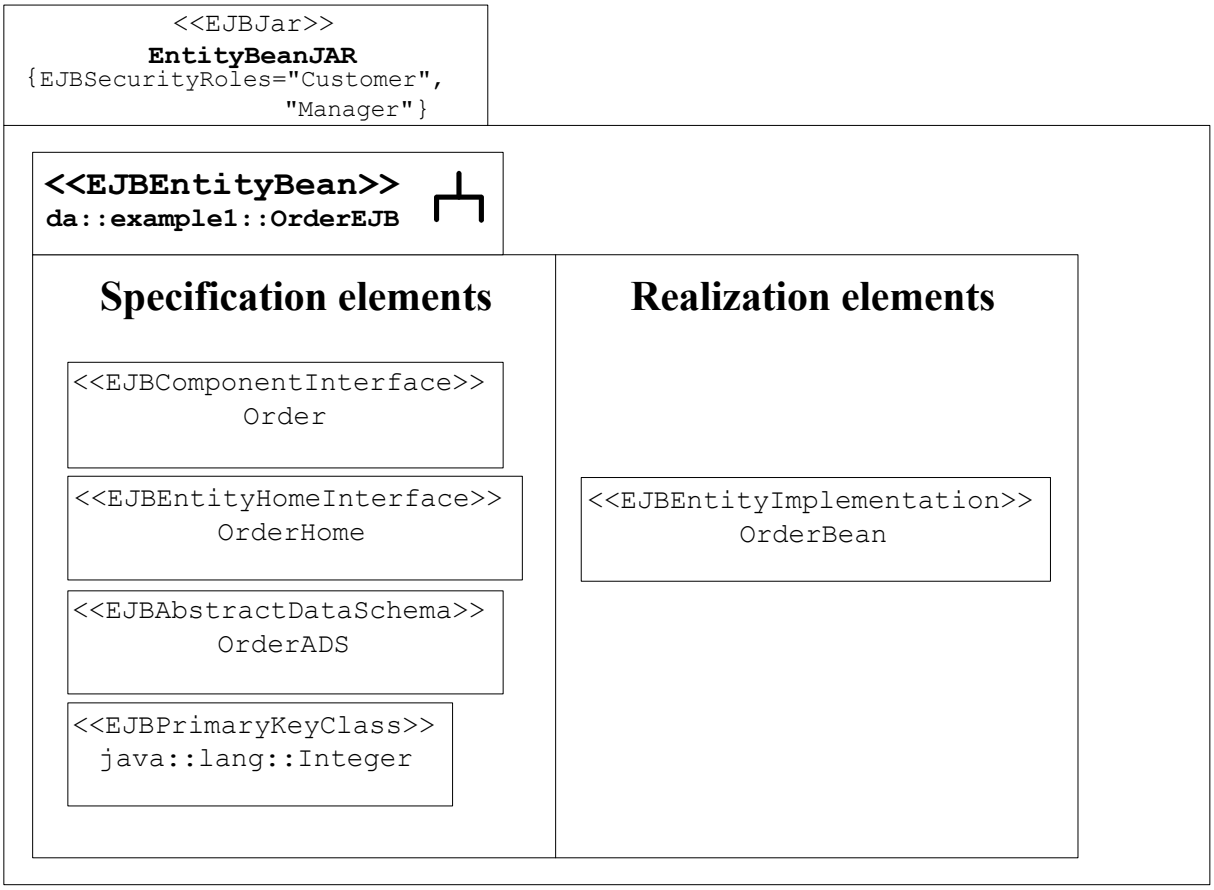


Abbildung 16 - EJB-Jar samt der sich dort befindlichen Entity-Bean

8.4.2 Persistente Assoziation zwischen zwei Entity-Beans

Der Schwerpunkt des zweiten Beispiels liegt auf der Modellierung einer, zwischen zwei Entity-Beans bestehenden, persistenten Assoziation.

An der Assoziation ist sowohl die im letzten Abschnitt vorgestellte Entity-Bean „OrderEJB“ beteiligt, als auch die Entity-Bean mit dem Namen „ArticleEJB“, die über ähnliche Eigenschaften wie „OrderEJB“ verfügt. Im Gegensatz zu „OrderEJB“ bietet „ArticleEJB“ jedoch nur lokalaufrufbare Schnittstellen an.

Die zwischen den beiden Entity-Beans modellierte Assoziation trägt den Namen „OrderToArticle“ und ist nur in Richtung „ArticleEJB“ navigierbar.

Das Assoziationsende auf Seiten der Entity-Bean „OrderEJB“ trägt den Rollennamen „myOrder“, besitzt die Multiplizität [0..*] und weist dem Tagged-Value „EJBAccessModifier“ den Wert „`None`“ zu. Außerdem verfügt dieses Assoziationsende über das Tagged-Value „EJBCascadeDelete“.

Das Assoziationsende auf Seiten der Entity-Bean „ArticleEJB“ trägt den Rollennamen „myArticle“ und besitzt die Multiplizität [0..1]. Da bei diesem Assoziationsende das Tagged-Value „EJBAccessModifier“ den Wert „Get“ besitzt, kann über die Entity-Bean „OrderEJB“ mittels der dort bereitgestellten `getArticle()`-Methode auf eine entsprechende Komponenteninstanz von „ArticleEJB“ zugegriffen werden.

Im übertragenen Sinne zusammengefasst, besagt dieses Beispiel, dass sich eine Bestellung („OrderEJB“) jeweils genau auf einen Artikel („ArticleEJB“) bezieht. Umgekehrt kann ein bestimmter Artikel beliebig vielen verschiedenen Bestellungen zugeordnet sein, ohne jedoch selbst von dieser Zuordnung zu wissen.

Wird ein bestimmter Artikel gelöscht, so verlieren auch alle ihn referenzierenden Bestellungen ihre Gültigkeit („EJBCascadeDelete“ = „True“).

Die beiden Entity-Beans befindet sich im Package „da::example2, das sich wiederum im Package „EntityBeanJAR“ befindet.

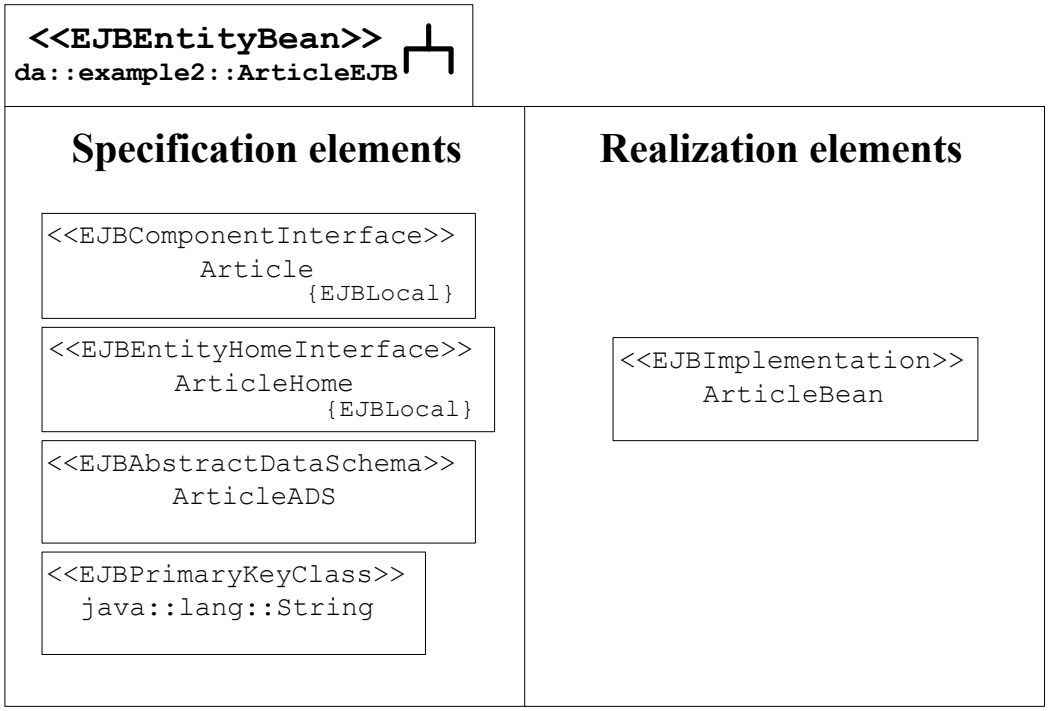


Abbildung 17 - Die Komponentenspezifikation der Entity-Bean "ArticleEJB"

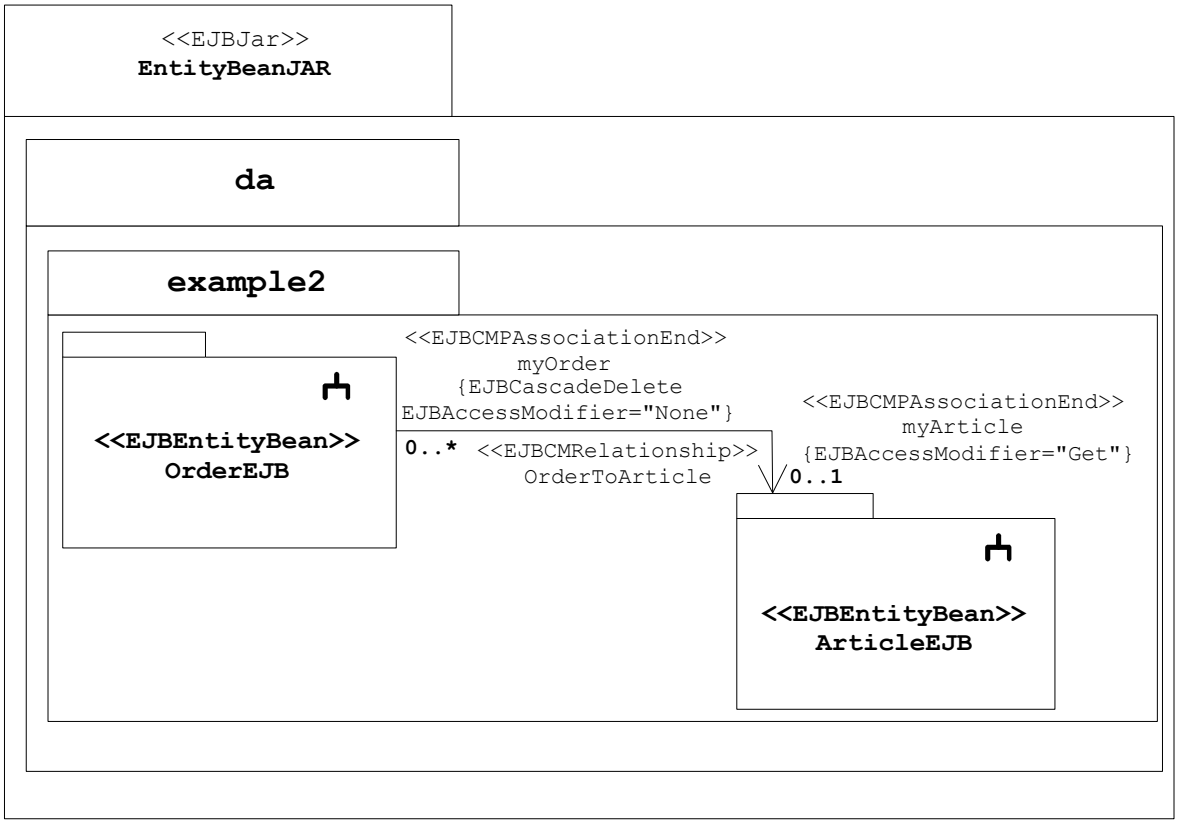


Abbildung 18 - Die persistente Assoziation zwischen "OrderEJB" und "ArticleEJB"

8.5 Funktionsnachweis

Der zu erbringende Funktionsnachweis basiert im Wesentlichen auf der Transformation der beiden zuvor vorgestellten EJB-Modelle mittels des in Abschnitt 8.3 gewählte Generatorkonzepts.

Der Funktionsnachweis gilt generell dann als erbracht, wenn folgende Anforderungen diesbezüglich erfüllt worden sind:

1. Die erzeugten Java-Codefragmente müssen in ihrer Syntax der momentan gültigen Java-Spezifikation [GJSB00] entsprechen.
2. Der erzeugte Deployment-Descriptor muss in seiner Syntax der, durch die EJB-2.0-DTD vorgegebenen Syntax entsprechen.
3. Sowohl die erzeugten Java-Codefragmente als auch der zugehörige Deployment-Descriptor müssen in ihrer Semantik den vorgegebenen UML-Modellen entsprechen.
4. Schließlich muss es möglich sein, die erzeugten EJB-Codefragmente innerhalb der, durch Sun frei vertriebenen J2EE-1.3.1-Referenzarchitektur zu ‚deployen', d.h. dort

im Rahmen eines EJB-Jars zu installieren. Die dafür eventuell nötigen Änderungen an den EJB-Codeartefakten dürfen nur ‚ergänzender' Art sein.

Die konkrete Umsetzung der Transformation findet auf Basis eines Java-basierten Komandozeilenwerkzeugs statt, dass aus einer vorgegebenen XMI-Datei, die das betreffende UML-Modell darstellt, mittels der beiden zuvor definierten XSLT-Skripte die gefordertern EJB-Codeartefakte generiert.

Der im Werkzeug genutzte Stylesheet-Prozessor mit der Produktbezeichnung „XALAN" ist im Internet frei verfügbar und kann unter folgender URL samt zugehöriger Dokumentation bezogen werden [WWW11].

Der erbrachte Funktionsnachweis inklusive der erzielten Teilergebnisse, sowie alle daran beteiligten Komponenten (Komandozeilenwerkzeug, „XALAN", XMI-Beispieldateien, XSLT-Skripte) befinden sich allesamt auf dem beiligenden elektronischen Datenträger.

Dadurch ist gewährleistet, dass der Funktionsnachweis unabhängig vom hier schriftlich erbrachten Nachweis jederzeit auf einem dafür geeigneten System reproduziert und damit nachvollzogen werden kann.

8.6 Bewertung

Die abschließende Bewertung des gewählten Generatorkonzepts kann aufgrund einer nicht vorhandenen alternativen Implementierung nur bezogen, auf die in diesem Kapitel spezifizierten allgemeingültigen Anforderungen erfolgen.

Da die in Abschnitt 8.1 formulierten, für alle Generatorkonzepte gleichermaßen geltenden Anforderungen erfolgreich umgesetzt worden sind und auch der geforderte Funktionsnachweis erbracht worden ist, kann also zunächst von einem gelungenen Ansatz gesprochen werden.

Werden jedoch die in Abschnitt 8.2.2 erwähnten Nachteile des deklarativen Ansatzes in die Bewertung miteinbezogen, so muss die oben festgestellte positive Bewertung nachträglich eingeschränkt werden.

Der gewählte Ansatz ist nur im Rahmen eines von vorneherein eingeschränkten und dadurch auch relativ übersichtlichen Projekts empfehlenswert. Die schnell ansteigende Komplexität der angesprochenen XSLT-Skripte sowie die insgesamt eher fehlende Erweiterbarkeit dieses Ansatzes läßt das objektorientierte Generatorkonzept für größere Projekte als unumgänglich erscheinen.

Diese Feststellung gilt zumindest solange, bis die für die aktuelle XSLT-Version (1.1) gültige Mängelliste im Rahmen einer neuen Version behoben ist.

Kapitel 9
Zusammenfassung und Ausblick

Das letzte Kapitel dient der Zusammenfassung aller erzielten Ergebnisse sowie der abschließenden Beurteilung, ob die in der zugrundeliegenden Aufgabenstellung formulierte Zielstellung erreicht wurde.

Den Abschluß dieser Arbeit bildet ein Ausblick.

Um die Zusammenfassung nachvollziehbar zu gestalten, wird zunächst der ursprüngliche Ausgangspunkt dieser Arbeit wiederholt. Im Anschluß daran erfolgt eine Aufzählung der erzielten Ergebnisse sowie die Formulierung, ob letztere zur Erfüllung der Aufgabenstellung beigetragen haben.

Bevor der angesprochene Ausblick erfolgt, werden die im Rahmen dieser Arbeit enstandenen „offenen“ Punkte diskutiert.

9.1 Ausgangspunkt

Den Ausgangspunkt dieser Arbeit bildet eine Aufgabenstellung, die zunächst allgemein auf das Fehlen von Anforderungen hinweist, deren Erfüllung „die Beschreibung der Kombination von serverseitiger Komponenten“ ermöglichen würde.

Des Weiteren wird in der Aufgabenstellung auf die Untersuchung der sich momentan in der Entstehung befindlichen Standards bezüglich der Komponentenmodellierung gedrängt und zwar in besonderem Maße hinsichtlich der eventuell möglichen Nutzung des „UML Profile for EJB“.

Dieses ist, dort wo es den noch zu erarbeitenden Anforderungen nicht genügt, anzupassen bzw. zu erweitern.

Schließlich wird auch die Erarbeitung von Abbildungsvorschriften gefordert, die eine spätere Codeabbildung von Profile-basierten EJB-Modellen ermöglicht.

Diese sind im Rahmen des praktischen Teils mittels eines prototypischen Generators umzusetzen.

9.2 Erzielte Ergebnisse

Zunächst fand im Rahmen des zweiten Kapitels eine Einführung in diejenigen Standardiesierungsbemühungen der OMG statt, die sich entweder direkt oder indirekt mit der Komponentenmodellerierung befassen; dazu gehören die MDA-Architektur und das EDOC-Profile.

Nachdem der Zusammenhang zwischen dem „UML Profile for EJB“ und den zuvor erwähnten Standardisierungsbemühungen hergestellt worden war, wurden im weiteren Verlauf dieses Kapitels Anforderungen an die Beschreibung der Kombination von serverseitigen Komponenten gesammelt und zwar konkret auf die EJB-2.0-Technologie bezogen. Dazu gehören einerseits die komponentenstrukturabhängigen Anforderungen und andererseits die komponentenbeziehungsabhängigen Anforderungen; letztere setzen sich hauptsächlich mit den zwischen EJBs prinzipiell möglichen Assoziationen auseinander. Die Vererbungsbeziehung wurde in Bezug auf das EJB-2.0-Komponentenmodell ausgeschlossen.

Das dritte Kapitel führte in die Architektur der ursprünglichen Profile-Fassung ein und vermied dabei zunächst jede mögliche Form der inhaltlichen Wertung.

Die Hauptaufgabe des vierten Kapitels bestand in der Bewertung der zuvor erläuterten Profile-Architektur. Die Bewertung orientierte sich an vier Kriterien: die Erfüllung der eigens im Profile definierten Zielsetzung, die semantische Nutzung des UML-Metamodells, die gewählte Erstellungsstrategie des Profiles und die Erfüllung der in Kapitel 2 formulierten Anforderungen.

Als Ergebnis wurde eine grundlegende Überarbeitung des Profiles empfohlen. Diese sollte sowohl in Bezug auf die noch nicht optimale Profile-Architektur erfolgen als auch im Hinblick auf die fehlende Unterstützung der EJB-2.0-Spezifikation, die bis dato z.B. die Modellierung von zwischen Entity-Beans bestehenden persistenten Assoziationen verhinderte.

Das fünfte Kapitel hatte die Optimierung der Profile-Architektur gemäß der in Kapitel 4 gestellten Anforderungen zum Inhalt. So wurde einerseits die UML-1.4-Konformität hergestellt und andererseits wurde durch die konsequente Anwendung des „Top-Down-Ansatzes“ eine Komplexitätsverminderung der Profile-Architektur erreicht.

Die Erweiterung des Profiles um die Neuerungen der EJB-2.0.-Spezifikation fand in Kapitel 6 statt. So ist nun sowohl die Modellierung von Message-Driven-Beans als auch die Beschreibung von persistenten Assoziationen zwischen Entity-Beans möglich. Insgesamt fand dadurch jedoch auch eine merkliche Erhöhung der Anzahl der Erweiterungselemente (Stereotypes und Tag-Definitions) statt, womit die Komplexität des Profiles wieder zunahm.

Um die, durch die konsequente Anwendung des „Top-Down-Ansatzes“ enstandene, hohe Abstraktion von Profile-basierten Modellen bei der angestrebten Codeabbildung überbrücken zu können, wurden im Rahmen des siebten Kapitels Abbildungsvorschriften spezifiziert, die die Abbildung dieser Modelle auf EJB-Codeartefakte ermöglichen.

Die EJB-Codeartefakte unterteilen sich in die für die Komponentenrealisierung benötigten Java-Klassendefinitionen und in den Deployment-Descriptor, der hauptsächlich der Komponentenspezifikation dient.

Der wesentliche Schwerpunkt der Abbildungsvorschriften lag in der semantischen Umsetzung von den zwischen EJBs prinzipiell möglichen Assoziationen. Dadurch ist letzten Endes auch die Modellierung von Komponentenbeziehungen, so wie es die Aufgabenstellung vorsieht, ermöglicht worden.

Das achte und vorletzte Kapitel hatte die prototypische Umsetzung der in Kapitel 7 definierten Abbildungsvorschriften mittels eines Generators zur Aufgabe.

Das dort gewählte Generatorkonzept basiert auf Anwendung zweier XSLT-Skripte, die mittels der durch sie beschriebenen Transformationsregeln vorliegende XMI-Dateien, die wiederum Profile-konforme EJB-Modelle repräsentieren, in die oben beschriebenen EJB-Codeartefakte umwandeln.

Die allgemeine Funktionsfähigkeit dieses Konzepts wurde schließlich anhand zweier einfacher Beispiele nachgewiesen.

9.3 Fazit

Das Gesamtergebnis dieser Arbeit kann, gemessen an der in der Aufgabenstellung formulierten Zielstellung, als gelungen angesehen werden.

Sowohl die zunächst geforderte Spezifizierung von Anforderungen, deren Erfüllung eine vollständige Modellierung von komponentenbasierten Anwendungen ermöglicht, als auch die konkrete Umsetzung dieser im Rahmen eines erweiterten „UML Profile for EJB“ ist erfolgt.

Schließlich erfolgte auch die geforderte Spezifikation der zur Codeabbildung notwendigen Abbildungsvorschriften sowie deren prinzipieller Nachweis mittels eines prototypischen Generators.

9.4 Offen gebliebene Punkte

Trotz des im letzten Abschnitt stattgefundenen positiven Resümees, müssen die im Rahmen dieser Arbeit enstandenen „offenen“ Punkte zumindest erwähnt werden.

Dort, wo sich ein möglicher Lösungsansatz zwar abzeichnete, aber aufgrund des beschränkten Zeitrahmens nicht mehr berücksichtigt werden konnte, werden dessen grundlegenden Elemente kurz skizziert.

9.4.1 Vererbungsrelation

Obwohl die Vererbungsrelation in Kapitel 2 für das EJB-Komponentenmodell ausgeschlossen wurde, ist vielleicht doch ein Ansatz vorhanden, der diese im Rahmen einer Codegenerierung zumindest semantisch nachbilden könnte.

Allerdings wäre zuvor eine präzise Untersuchung des noch nicht allgemeingültig defiinierten Komponentenvererbungsbegriffs notwendig – und diese Untersuchung böte wahrscheinlich genug Spielraum für eine eigene wissenschaftliche Arbeit.

9.4.2 Profile-Erstellungsstrategie

Die in Kapitel 4 nur sehr oberflächlich erfolgte Einführung in den sogenannten „Top-Down-Ansatz“ bietet nicht unbedingt die Grundlage für eine optimale Profile-Erstellungsstrategie.

Es wäre in diesem Rahmen durchaus vorstellbar, dass es einen dazugehörigen Musterkatalog gäbe, der eine deterministische, dabei präzis beschriebene und allgemein anwendbare Profile-Erstellungsstrategie ermöglichen würde.

Auch dies kann als mögliche Aufgabenstellung einer separaten Arbeit angesehen werden.

9.4.3 „Abbildungslücken“

Im Rahmen der in Kapitel 7 formulierten Abbildungsvorschriften sind die Java-bezogenen Erweiterungselemente vollständig ignoriert worden.

Da jedoch ein eigenes „UML Profile for Java“ zu erwarten ist, kann die fehlende Unterstützung der dafür bisher vorgesehenen Erweiterungselemente als nicht so gravierend angesehen werden.

Ein anderes Problem der bestehenden Abbildungsvorschriften ist die zu geringe Verknüpfung der duch sie entstehenden Java-Codefragmente untereinander.

So wäre z.B. die Generierung des automatischen Aufrufens bestimmter, zuvor erzeugter Hilfsmethoden in den dafür vorgesehenen ejbCreate()-Methoden der Implementierungsklasse wünschenswert. Allerdings hätte die Berücksichtigung dieser Abhängigkeiten den engen dieser Arbeit zugrundeliegenden Zeitraum gesprengt und sollte somit im Rahmen späterer Erweiterungen vorgesehen werden.

9.4.4 „Round-Trip-Engineering“

Ein weiterer interessanter Aspekt ist die Frage, ob die durch dieses Profile erzeugten EJB-Codeartefakte prinzipiell dem Paradigma des „Round-Trip-Engineerings“ genügen.

Der Aspekt des „Forward-Engineerings“ ist durch die Formulierung dafür gültiger Abbildungsvorschriften komplett bedacht worden; vielleicht es aber auch möglich, mittels noch zu formulierender Ansätze, ein vorliegendes EJB-Jar vollständig in ein Profile-basiertes EJB-Modell rückzutransformieren. Dies entspräche dann dem Paradigma des „Reverse-Engineerings“.

9.4.5 Werkzeugunterstützung

Auch die Profile-Unterstützung durch CASE-Werkzeuge, die dadurch testbare Anwendbarkeit des Profiles sowie die Austauschbarkeit der damit erstellten EJB-Modelle wäre ein für zukünftige Untersuchungen interessanter Bereich.

Zuvor müssten jedoch Vergleichs- bzw. Bewertungskritieren erarbeitet werden, ohne die eine solche Untersuchung wissenschaftlich gesehen überhaupt keinen Sinn ergäbe.

Zu diesen Kriterien könnte z.B. die Vollständigkeit der erbrachten Profile-Unterstützung gehören (UML-1.4-Konformität, XMI-1.1-Unterstützung) aber auch, ob eine durch das jeweilige Werkzeug unterstützte Codeabbildung möglich ist (z.B. mittels eines Template-Mechanismus).

9.4.6 Generatorkonzept

Abschließend bleibt festzustellen, dass die prototypische Umsetzung, die sich an der vergleichsweisen geringen Anzahl angewandter Beispiele manifestiert, der in Kapitel 7 definierten Abbildungsvorschriften mittels eines Generatos als „offener“ Punkt schlechthin gilt.

Aus diesem Grund ist sowohl die Vervollständigung der in den beiden XSLT-Skripten beschriebenen Transformationsregeln erstrebenswert als auch die Implementierung eines dazu alternativen objektorientierten Ansatzes.

Beides zusammengenommen bildet wiederum die Grundlage einer eigenen Arbeit.

9.5 Ausblick

Zum Abschluß dieses Kapitels und damit auch zum Abschluß dieser Arbeit, soll versucht werden, einen Ausblick bezüglich der Zukunft des erweiterten „UML Profile for EJB“ zu geben.

Dabei spielen drei Entwicklungen eine wesentliche Rolle:

1. Der Fortschritt des mit der ursprünglichen Fassung des Profiles verbundenen JCPs.
2. Die abschließende Entwicklung des EDOC-Profiles und damit verbunden auch die Frage, ob sich das diesem Profile zugrundeliegende EJB-Metamodell nochmals ändern wird.
3. Die Entwicklung der UML im Hinblick auf die kommende UML-2.0-Spezifikation.

Jede der drei Entwicklungen kann für sich genommen das Ergebnis dieser Arbeit obsolet machen. Allerdings beeinflussen sich die drei Entwicklungen auch gegenseitig, den stärksten Einfluß hat dabei die Änderung der UML-Architektur, die durch die Einführung der UML-2.0-Spezifikation zustande kommt.

Insofern kann von dieser Stelle aus keine ernsthafte Prognose betreffend der möglichen Auswirkungen auf das dieser Arbeit zugrundeliegende Ergebnis gemacht werden.

Es bleibt daher zu hoffen, dass sich einerseits die umsichtige Wahl der diesem Profile zugrundeliegenden Metamodelle auszahlt und dass die sich andererseits abzeichnenden Entwicklungen ihrerseits Rücksicht nehmen – und zwar im Sinne der Wahrung von Abwährtskompatibilität.

Im schlimmsten Fall bleiben aber immer noch die in Kapitel 2 erarbeiteten und dabei allgemeingültig formulierten Anforderungen an die Komponentenmodellierung als verwertbares Ergebnis dieser Arbeit übrig.

Anhang A
„UML Profile for EJB“

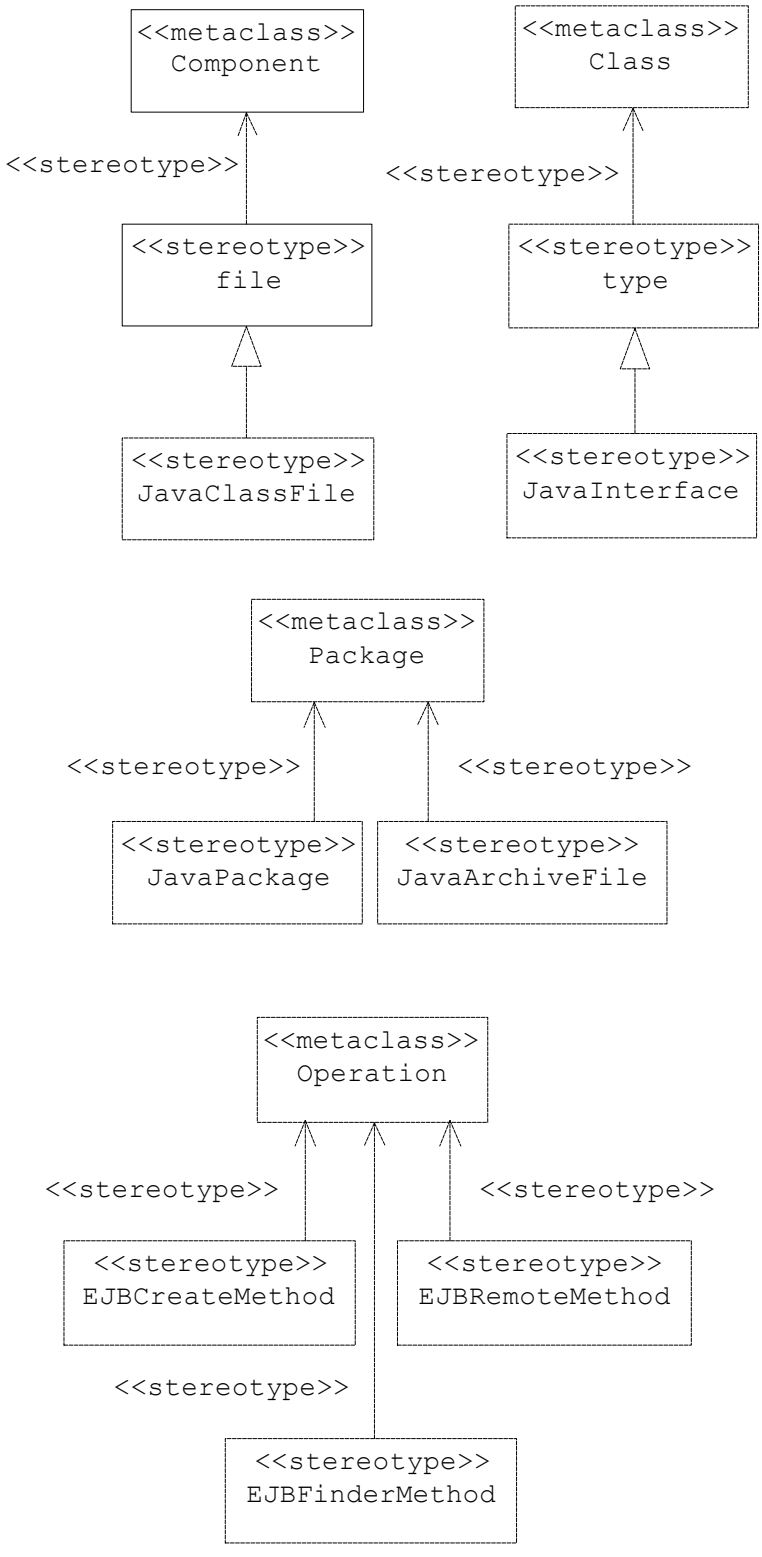


Abbildung 19 - Im "UML Profile for EJB" definierte Stereotypes (1/4)

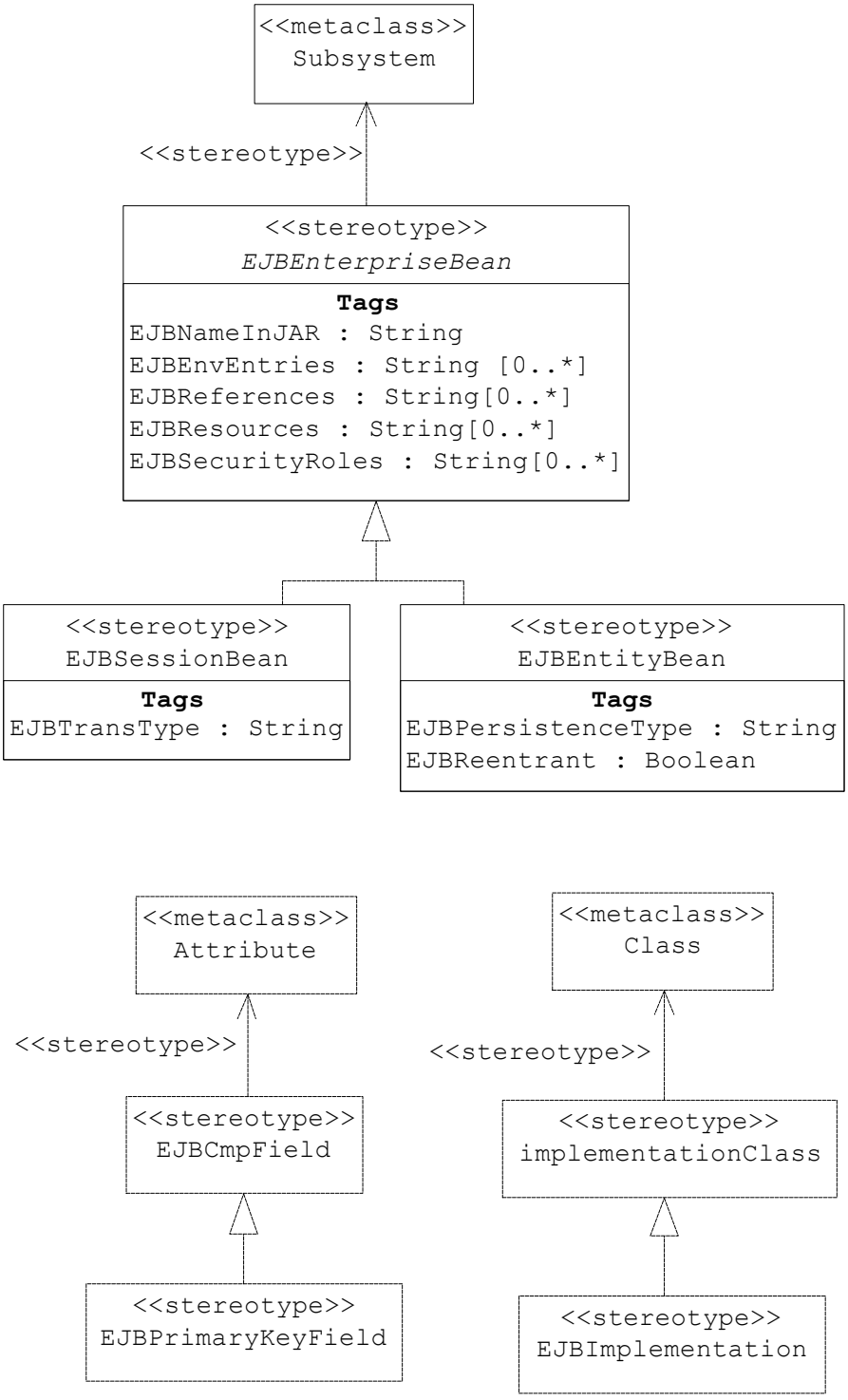
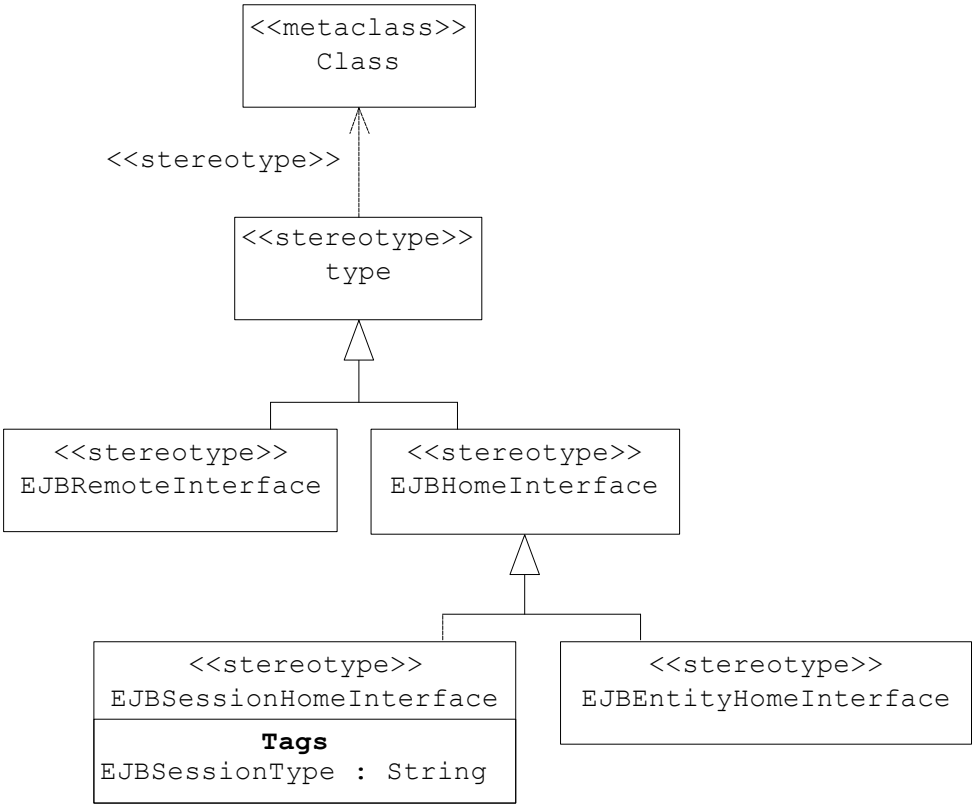


Abbildung 20 - Im "UML Profile for EJB" definierte Stereotypes (2/4)



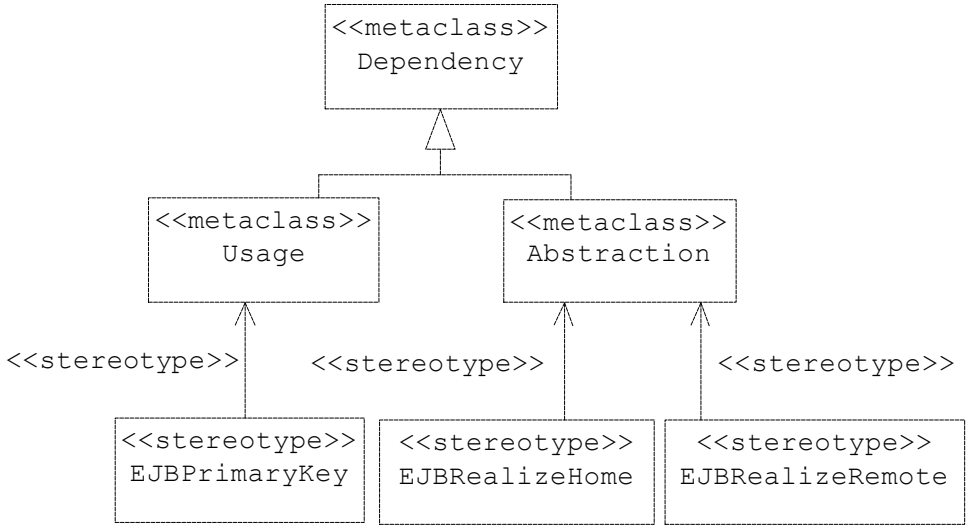


Abbildung 21 - Im "UML Profile for EJB" definierte Stereotypes (3/4)

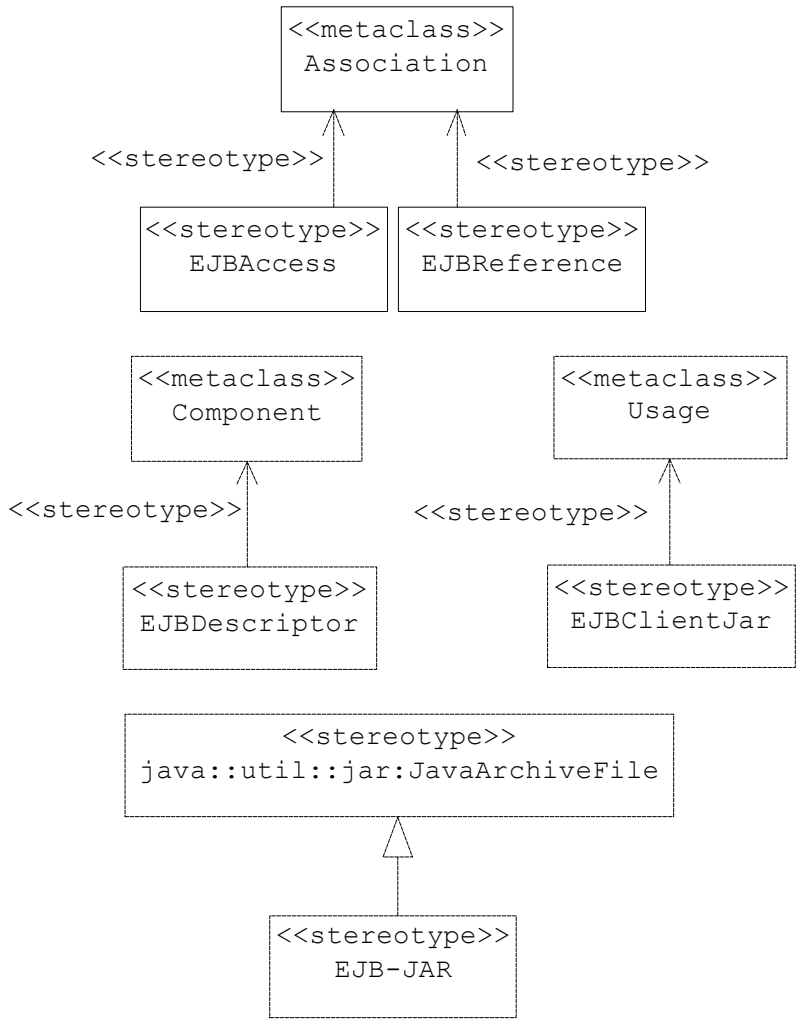


Abbildung 22 - Im "UML Profile for EJB" definierte Stereotypes (4/4)

Tabelle 9 – Im "UML Profile for EJB" definierte Stereotypes

Stereotype	Base Class	Parent	Tags	Constraints	Description
JavaInterface	Class	type	N/A	N/A	Indicates that the Cass represents a Java Interface.
JavaClassFile	Component	file	N/A	N/A	Indicates that the Component represents a Java Class File.
JavaArchiveFile	Package	N/A	N/A	N/A	Indicates that the Package represents a JAR.
EJBCreate-Method	Operation	N/A	N/A	N/A	Indicates that the Operation represents an EJB Create Method.
EJBFinder-Method	Operation	N/A	N/A	N/A	Indicates that the Operation represents an EJB Finder Method.
EJBRemote-Method	Operation	N/A	N/A	N/A	Indicates that the Operation represents an EJB Remote Method.
EJBRemote-Interface	Class	type	N/A	N/A	Indicates that the Class represents an EJB Remote Interface.
EJBHome-Interface	Class	type	N/A	N/A	The abstract Stereotype indicates that the Class represents an EJB Home Interface.
EJBSession-HomeInterface	Class	EJBHomeInterface	EJBSessionType	N/A	Indicates that the Class represents an EJB Session Home.
EJBEntity-HomeInterface	Class	EJBHomeInterface	N/A	N/A	Indicates that the Class represents an EJB Entity Home.
EJBPrimaryKey	Usage	N/A	N/A	N/A	Indicates that the supplier of the Usage represents the EJB Primary Key Class for the EJB Entity Home represented by the client.
EJBCmpField	Attribute	N/A	N/A	N/A	Indicates that the Attribute represents a container-managed field for an EJB Entity Bean with container-managed persistence.
EJBPrimary-KeyField	Attribute	EJBCmpField	N/A	N/A	Indicates that the Attribute is the primary key field for an EJB Entity Bean with container-managed persistence.
EJBRealize-Home	Abstraction	N/A	N/A	N/A	Indicates that the supplier of the Abstraction represents an EJB Home Interface for the EJB Implementation Class represented by the client.
EJB-Implementation	Class	implementation-Class	N/A	N/A	Indicates that the Class represents an EJB Implementation Class.
EJBEnterprise-Bean	Subsystem	N/A	EJBEnvEntries EJBNameInJAR EJBReferences EJBResources EJBSecurity-Roles	N/A	The abstract Stereotype indicates that the Subsystem represents an EJB Enterprise Bean.
EJBSession-Bean	Subsystem	EJBEnterprise-Bean	N/A	N/A	Indicates that the Subsystem represents an EJB Session Bean.
EJBEntityBean	Subsystem	EJBEnterprise-Bean	EJBPersistence-Type EJBReentrant	N/A	Indicates that the Subsystem represents an EJB Entity Bean.
EJBReference	Association	N/A	N/A	N/A	Indicates that the navigable end of the Association represents a referenced EJB Enterprise Bean.

Stereotype	Base Class	Parent	Tags	Constraints	Description
EJBAccess	Association	N/A	N/A	N/A	Indicates that the Association defines a security role name relationship between an Actor and an EJB Enterprise Bean.
EJB-JAR	Package	JavaArchiveFile	N/A	N/A	The abstract Stereotype indicates that the Class represents a Home Interface.
EJBDescriptor	Component	file	N/A	N/A	Indicates that the Component represents an EJB Deployment Descriptor.
EJBClientJAR	Usage	N/A	N/A	N/A	Indicates that the client of the Usage represents an ejb-client-jar for the EJB-JAR represented by the supplier of the Usage.

Tabelle 10 - Im "UML Profile for EJB" definierte Tagged-Values

Tag	Stereotype	Description
JavaStrictfp	Class	A Boolean value indicating whether or not the Java Class is FP-strict.
JavaStatic	Class or Interface	A Boolean value indicating whether or not the Java Class or Interface is static.
JavaVolatile	Attribute or AssociationEnd	A Boolean value indicating whether or not the Java Field is volatile.
JavaDimensions	Attribute or AssociationEnd	An Integer value indicating the number of array dimensions declared by the Java Field.
JavaCollection	Attribute or AssociationEnd	A String containing the name of the Java Collection Type used to implement an Attribute or AssociationEnd with complex multiplicity.
JavaNative	Operation	A Boolean value indicating whether or not the Java Method is native.
JavaThrows	Operation	A comma-delimited list of names of Java Exception Classes.
JavaFinal	Parameter	A Boolean value indicating whether or not the Parameter is final.
JavaDimensions	Parameter	An Integer value indicating the number of array dimensions declared by the Paramter.
EJBSessionType	Class «EJBSessionHome-Interface»	An enumeration with values “Stateful” or “Stateless”. Indicates whether or not the Session Bean maintains state.
EJBRoleNames	Operation	Contains the security roles that may invoke the Operation.
EJBTransAttribute	Operation	An enumeration with values “Not Supported”, “Supports”, “Required”, “RequiresNew”, “Mandatory” or “Never”. Defines the transaction management policy for the Operation.
EJBEnvEntries	Subsystem «EJBEnterpriseBean»	A comma-delimited list of tuples of the form “**<NAME>**, **<TYPE>**, **<VALUE>**”, designating the environment entries used by the EJB.
EJBNameInJAR	Subsystem «EJBEnterpriseBean»	The name used for the EJB in the EJB-JAR. Defaults to the name of the Remote Interface.
EJBReferences	Subsystem «EJBEnterpriseBean»	A comma-delimited list of tuples of the form “**<NAME>**, **<TYPE>**, **<HOME>**, **<REMOTE>**”, designating the other EJBs referneced by the EJB.”
EJBResources	Subsystem «EJBEnterpriseBean»	A comma delimited list of tuples of the form “**<NAME>**, **<TYPE>**, **<AUTH>**”, designating the resource factories used by the EJB.
EJBSecurityRoles	Subsystem «EJBEnterpriseBean»	A comma-delimited list of tuples of the form “**<NAME>**, **<LINK>**”, designating the role names that may invoke all Operations on the EJB.
EJBTransType	Subsystem «EJBSessionBean»	An enumeration with values “Bean” or “Container”. Indicates whether the transactions of the Session Bean are managed by the Bean or by its container, respectively.
EJBPersistenceType	Subsystem «EJBEntityBean»	An enumeration with values “Bean” or “Container”. Indicates whether the persistence of the Entity Bean is managed by the Bean or by its container, respectively.
EJBReentrant	Subsystem «EJBEntityBean»	A Boolean value indicating whether or not the Entity Bean can be called reentrantly.

Anhang B
Optimiertes Profile

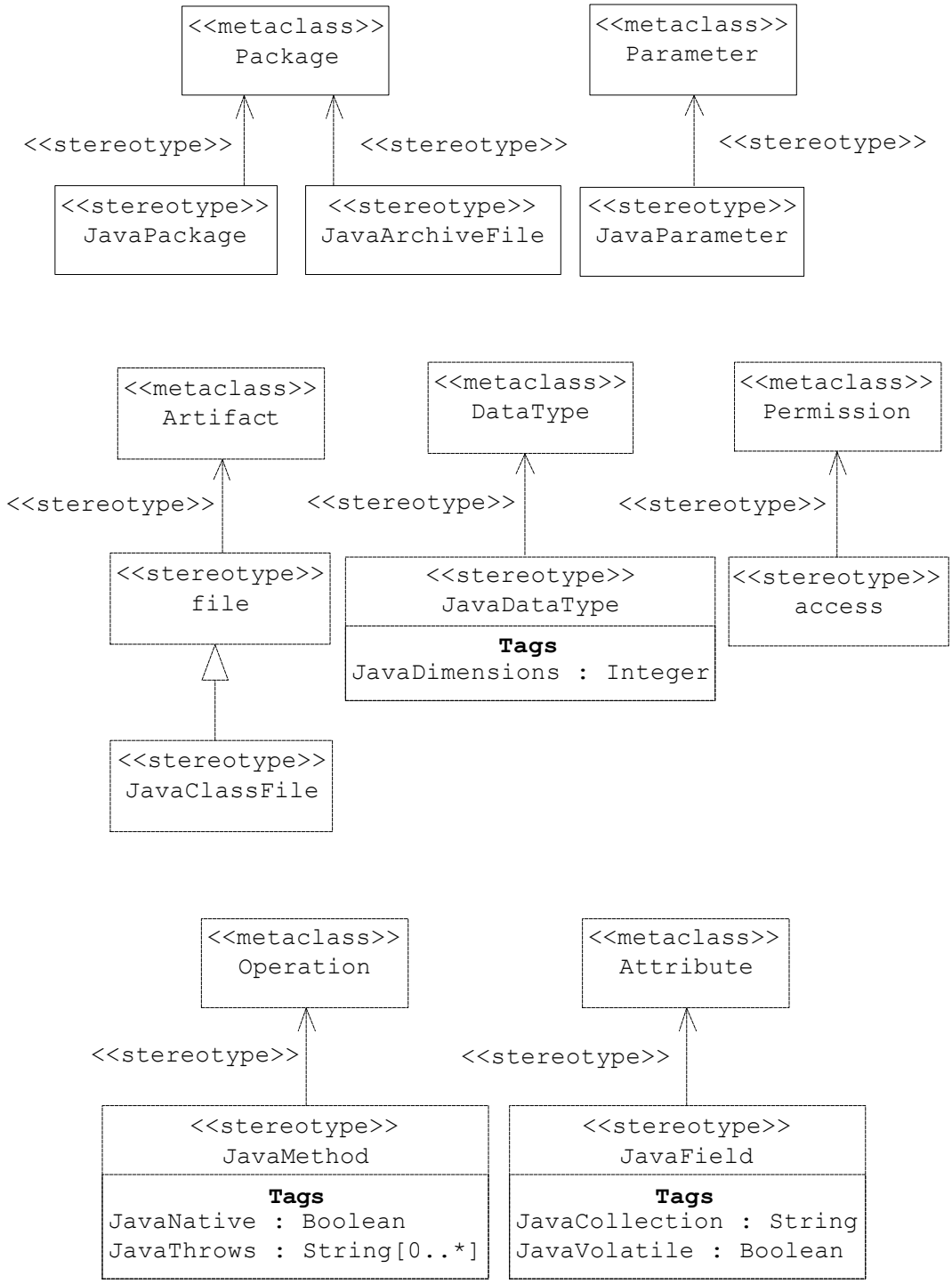


Abbildung 23 – Im optimierten Profile definierte Stereotypes (1/4)

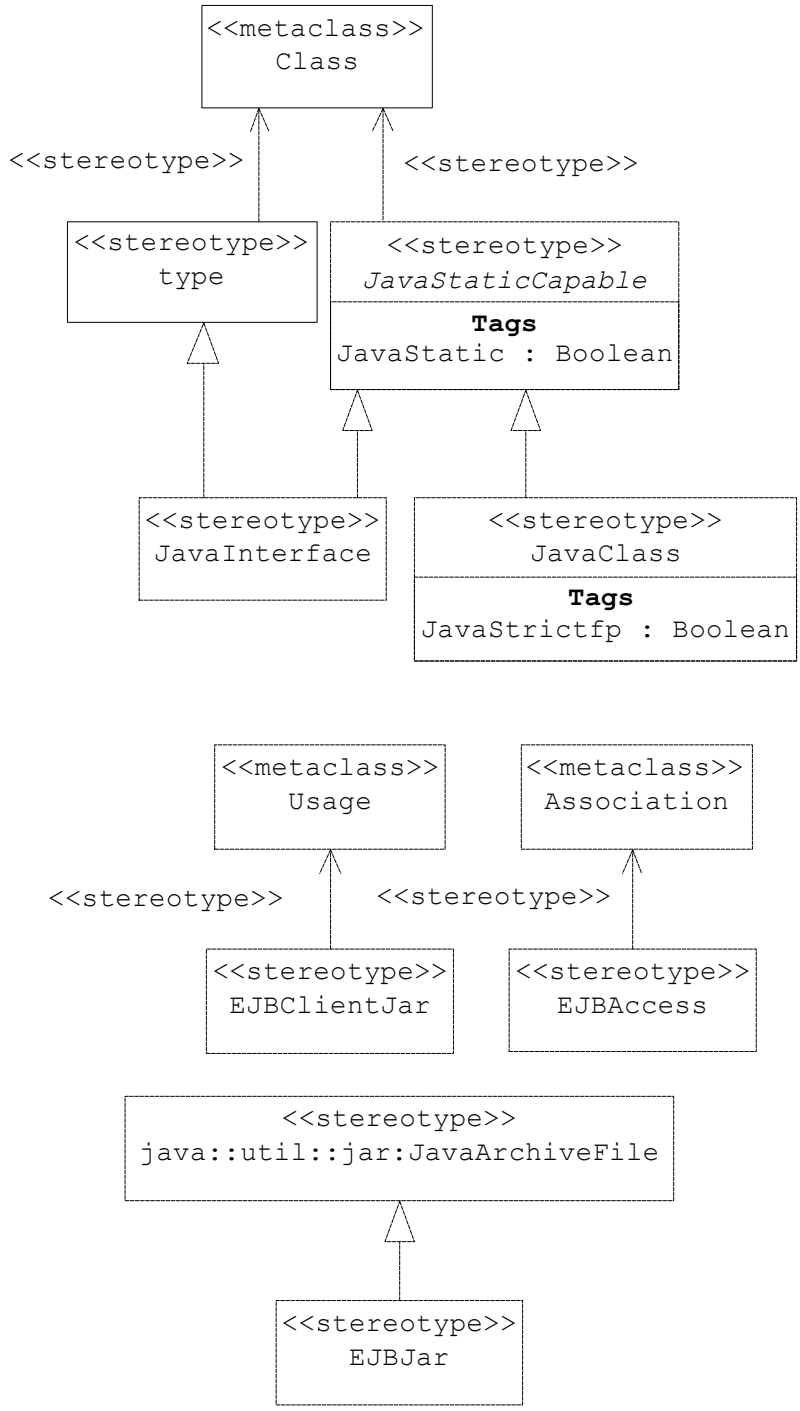


Abbildung 24 - Im optimierten Profile definierte Stereotypes (2/4)

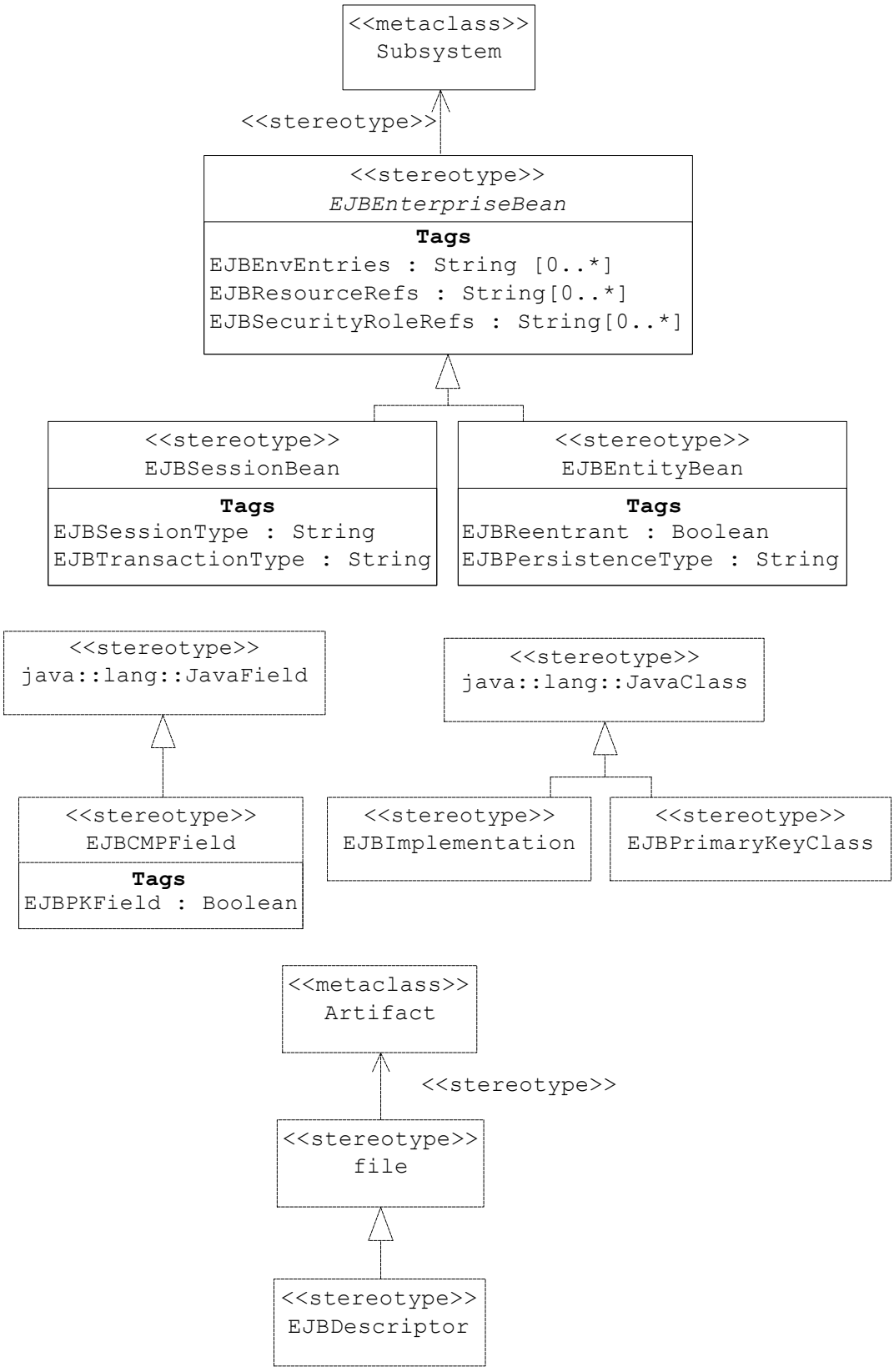


Abbildung 25 - Im optimierten Profile definierte Stereotypes (3/4)

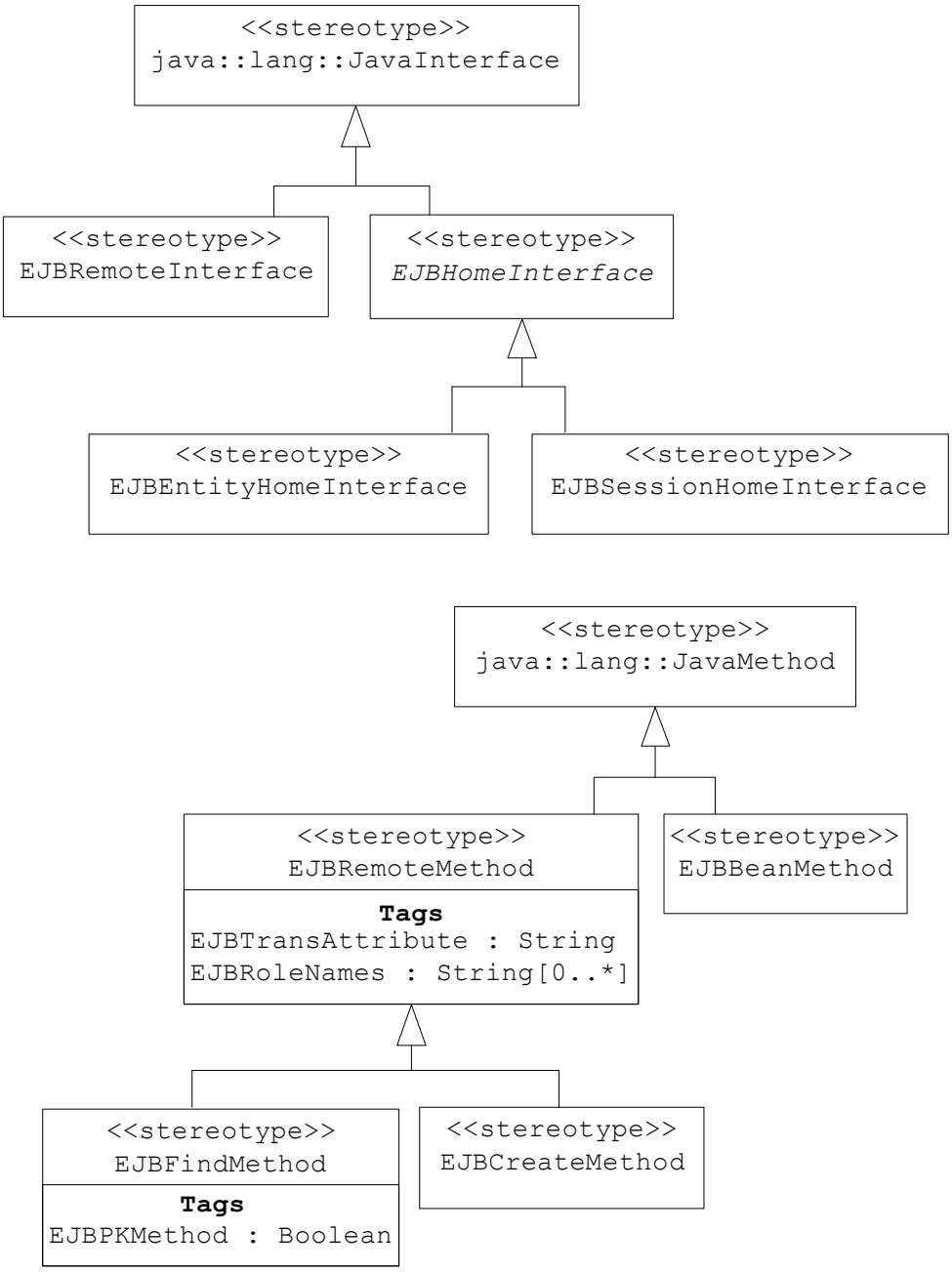


Abbildung 26 - Im optimierten Profile definierte Stereotypes (4/4)

Tabelle 11 - Im optimierten Profile definierte Stereotypes

Stereotype	Base Class	Parent	Tags	Constraints	Description
JavaPackage	Package	N/A	N/A	N/A	Indicates that the Package describes a Java Package.
JavaArchiveFile	Package	N/A	N/A	N/A	Indicates that the Package describes a JAR.
JavaParameter	Parameter	N/A	N/A	N/A	Indicates that the Parameter describes a Java Parameter.
JavaClassFile	Artifact	File	N/A	N/A	Indicates that the Artifact describes a Java Class File.
JavaDataType	DataType	N/A	JavaDimensions	N/A	Indicates that the Data Type describes a Java Data Type.
JavaImport	Permission	Access	N/A	N/A	Indicates that the Permission describes a Java Import Statement.
JavaMethod	Operation	N/A	JavaNative JavaThrows	N/A	Indicates that the Operation describes a Java Method.
JavaInterface	Class	Type AbstractJavaClass	N/A	N/A	Indicates that the Class describes a Java Interface.
JavaStaticCapable	Class	N/A	JavaStatic	N/A	The abstract Stereotype indicates that the Class is capable of being static.
JavaClass	Class	AbstractJavaClass	JavaStrictfp	N/A	Indicates that the Class describes a Java Class.
JavaField	Attribute	N/A	JavaCollection JavaVolatile	N/A	Indicates that the Attribute describes a Java Field.
EJBRemoteMethod	Operation	JavaMethod	EJBRoleNames EJBTrans-Attribute	N/A	Indicates that the Operation represents an Remote Interface Method.
EJBCreateMethod	Operation	EJBRemoteMethod	N/A	N/A	Indicates that the Operation represents a Create-Method.
EJBFindMethod	Operation	EJBRemoteMethod	EJBPKMethod	N/A	Indicates that the Operation represents a Find-Method.
EJBBeanMethod	Operation	JavaMethod	N/A	N/A	Indicates that the operation represents an Implementation Class Method.
EJBRemote-Interface	Class	JavaInterface	N/A	N/A	Indicates that the Class represents a Remote Interface.
EJBHomeInterface	Class	JavaInterface	N/A	N/A	The abstract Stereotype indicates that the Class represents a Home Interface.
EJBSession-HomeInterface	Class	EJBHomeInterface	N/A	N/A	Indicates that the Class represents a Session Bean Home Interface.
EJBEntity-HomeInterface	Class	EJBHomeInterface	N/A	N/A	Indicates that the Class represents an Entity Bean Home Interface.
EJBEnterpriseBean	Subsystem	N/A	EJBEnvEntries EJBResource-Refs EJBSecurity-RoleRefs	N/A	An abstract Stereotype indicating that the Subsystem represents an EJB.
EJBSessionBean	Subsystem	EJBEnterprise-Bean	EJBSessionType EJBTransaction-Type	N/A	Indicates that the Subsystem represents a Session Bean.
EJBEntityBean	Subsystem	EJBEnterprise-Bean	EJBReentrant EJBPersistence-Type	N/A	Indicates that the Subsystem represents an Entity Bean.
EJBImplementation	Class	JavaClass	N/A	N/A	Indicates that the Class represents an Implementation Class.

Stereotype	Base Class	Parent	Tags	Constraints	Description
EJBPrimary-KeyClass	Class	JavaClass	N/A	N/A	Indicates that the Class represents a Primary Key Class.
EJBCmpField	Attribute	JavaField	EJBPKField	N/A	Indicates that the Attribute represents a container-managed field of the Implementation Class.
EJBAccess	Association	N/A	N/A	N/A	Indicates that the Association defines a security role name relationship between an Actor and an EJB Enterprise Bean.
EJBJar	Package	JavaArchiveFile	N/A	N/A	Indicates that the Package represents an EJB-JAR.
EJBDescriptor	Artifact	File	N/A	N/A	Indicates that the Component represents a Deployment Descriptor.
EJBClientJar	Usage	N/A	N/A	N/A	Indicates that the client of the Usage represents an ejb-client-jar for the supplier.

Tabelle 12 – Im optimierten Profile definierte Tag-Definitions

Tag	Stereotype	Type	Multiplicity	Description
JavaDimensions	JavaDataType	Integer	1	An Integer value indicating the number of array dimensions declared by a Java Data Type.
JavaNative	JavaMethod	Boolean	1	A Boolean value indicating whether or not the Java Method is native.
JavaThrows	JavaMethod	String	*	Contains all names of used Java Exception Classes.
JavaStatic	AbstractJavaClass	Boolean	1	A Boolean value indicating whether or not the Abstract Java Class is static.
JavaStrictfp	JavaClass	Boolean	1	A Boolean value indicating whether or not the Java Class is FP-strict.
JavaCollection	JavaField	String	1	A String containing the name of the Java Collection Type used to implement a Java Field with complex multiplicity.
JavaVolatile	JavaField	Boolean	1	A Boolean value indicating whether or not the Java Field is final.
EJBRoleNames	EJBRemoteMethod	String	*	Contains the security roles that may invoke the Operation.
EJBTransAttribute	EJBRemoteMethod	String	1	An enumeration with values “Not Supported”, “Supports”, “Required”, “RequiresNew”, “Mandatory” or “Never”. Defines the transaction management policy for the Operation.
EJBEnvEntries	EJBEnterpriseBean	String	*	Contains tuples of the form “**<NAME>**, **<TYPE>**, **<VALUE>**”, designating the environment entries used by the EJB.
EJBResourceRefs	EJBEnterpriseBean	String	*	Contains tuples of the “**<NAME>**, **<TYPE>**, **<AUTH>**”, designating the resource factories used by the EJB.
EJBSecurityRoleRefs	EJBEnterpriseBean	String	*	Contains tuples of the form “**<NAME>**, **<LINK>**”, designating the role names that may invoke all Operations on the EJB.
EJBSessionType	EJBSessionHome-Interface	String	1	An enumeration with values “Stateful” or “Stateless”. Indicates whether or not the Session Bean maintains state.
EJBTransactionType	EJBSessionBean	String	1	An enumeration with values “Bean” or “Container”. Indicates whether the transactions of the Session Bean are managed by the Bean or by its container, respectively.

Tag	Stereotype	Type	Multiplicity	Description
EJBReentrant	EJBEntityBean	Boolean	1	A Boolean value indicating whether or not the Entity Bean can be called reentrant.
EJBPersistenceType	EJBEntityBean	String	1	An enumeration with values “Bean” or “Container”. Indicates whether the persistence of the Entity Bean is managed by the Bean or by its container, respectively.
EJBPKField	EJBCMPField	Boolean	1	A Boolean value indicating whether or not the Attribute is an Primary Key Field.
EJBPKMethod	EJBFindMethod	Boolean	1	A Boolean value indicating whether or not the Operation represents the findByPrimaryKey-Method..

Anhang C
Erweitertes Profile

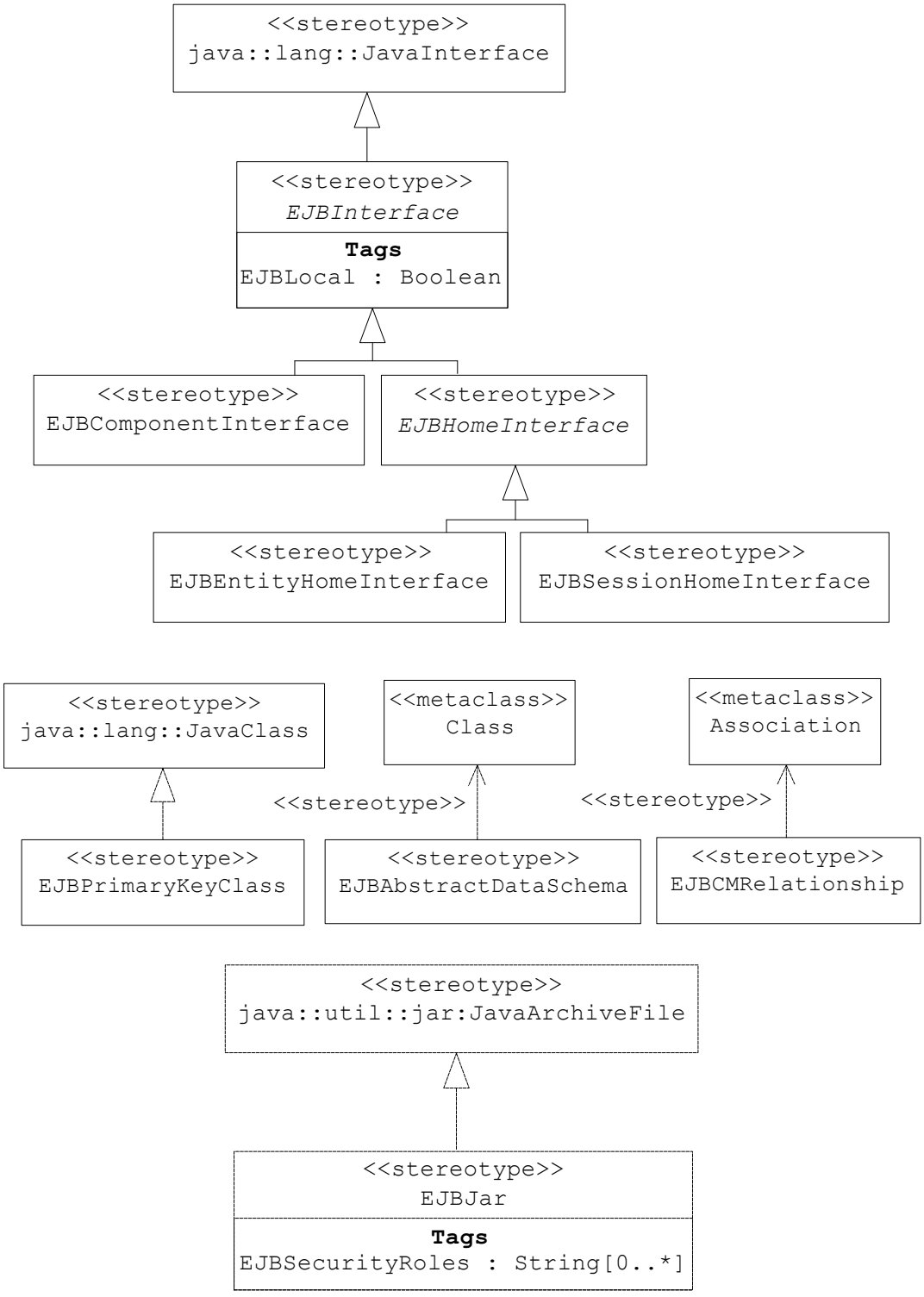


Abbildung 27 – Im erweiterten Profile definierte EJB-Stereotypes (1/3)

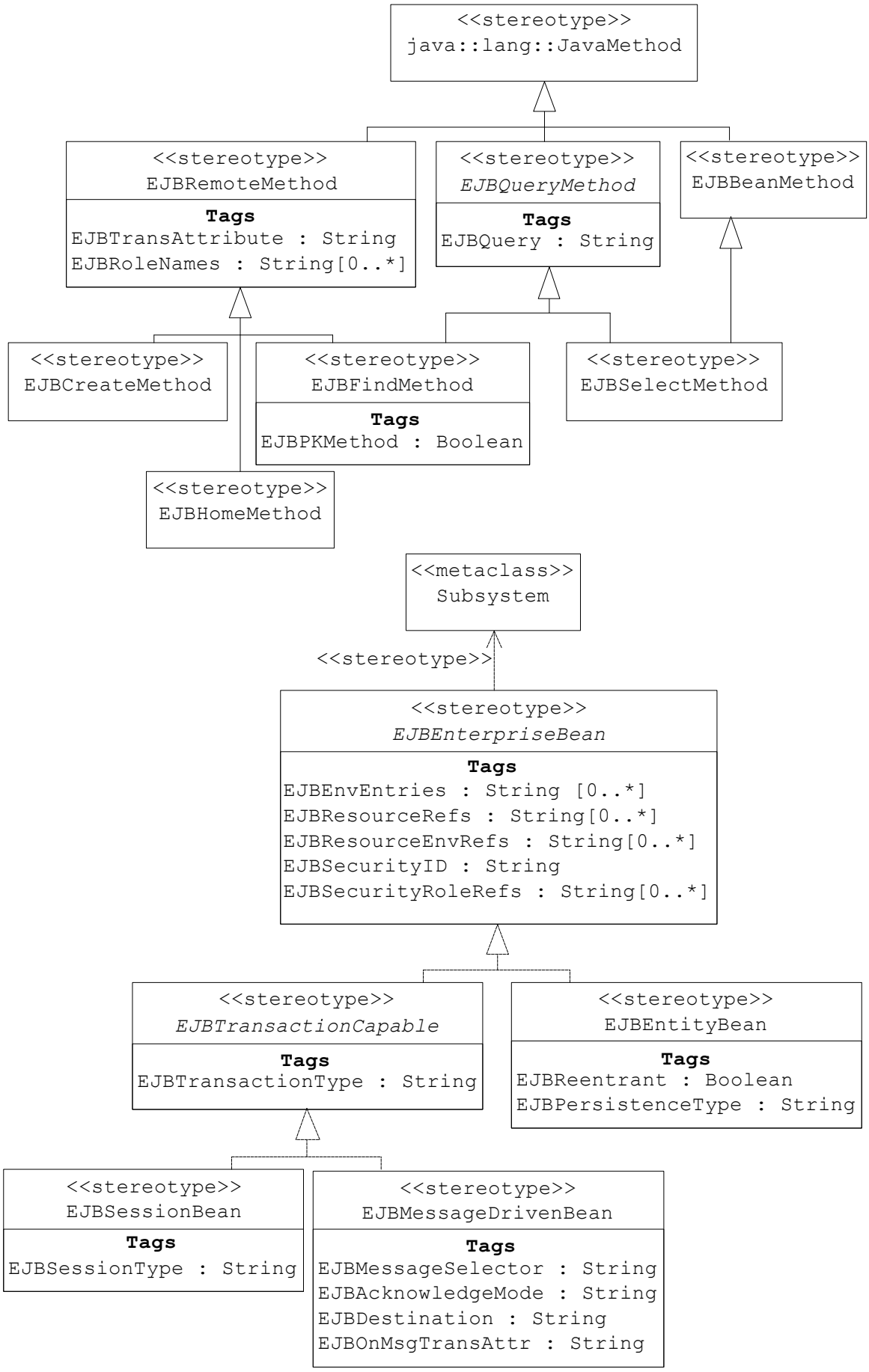


Abbildung 28 - Im erweiterten Profile definierte EJB-Stereotypes (2/3)

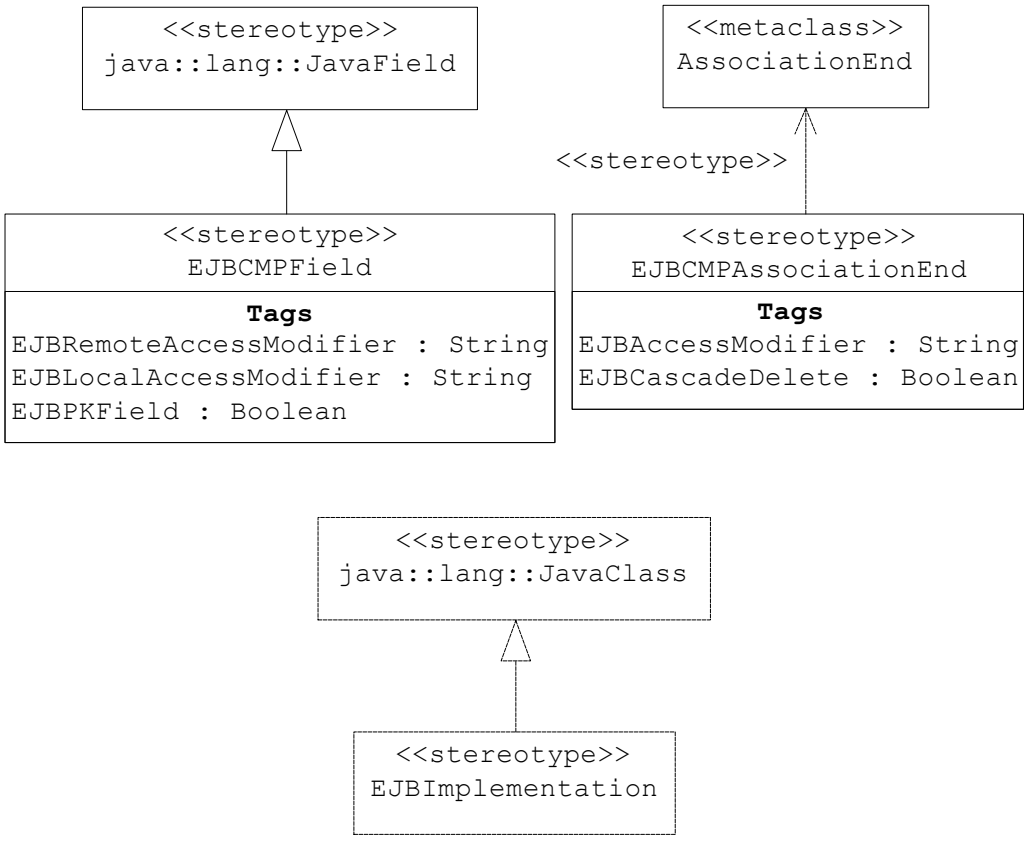


Abbildung 29 - Im erweiterten Profile definierte EJB-Stereotypes (3/3).

Tabelle 13 – Im erweiterten Profile definierte EJB-Stereotypes

Stereotype	Base Class	Parent	Tags	Constraints	Description
EJBInterface	Class	JavaInterface	EJBLocal	N/A	The abstract Stereotype indicates that the Class represents an EJB Interface.
EJBComponent-Interface	Class	EJBInterface	N/A	1	Indicates that the Class represents a Component Interface.
EJBHomeInterface	Class	EJBInterface	N/A	N/A	The abstract Stereotype indicates that the Class represents a Home Interface.
EJBSession-HomeInterface	Class	EJBHomeInterface	N/A	2-4	Indicates that the Class represents a Session Bean Home Interface.
EJBEntity-HomeInterface	Class	EJBHomeInterface	N/A	5-6	Indicates that the Class represents an Entity Bean Home Interface.
EJBCMPField	Attribute	JavaField	EJBRemote-AccessModifier EJBLocal-AccessModifier EJBPKField	N/A	Indicates that the Attribute represents an container-managed field.
EJBCM-Relationship	Association	N/A	N/A	7	Indicates that the Association represents an container-managed relationship.
EJBEnterpriseBean	Subsystem	N/A	EJBEnvEntries EJBResourceRefs EJBResource-EnvRefs EJBSecurityID EJBSecurity-RoleRefs	8	An abstract Stereotype indicating that the Subsystem represents an EJB.
EJBTransaction-Capable	Subsystem	EJBEnterprise-Bean	EJBTransaction-Type	N/A	An abstract Stereotype indicating that the Subsystem represents an transaction capable EJB.
EJBSession-Bean	Subsystem	EJBTransaction-Capable	EJBSessionType	9-13	Indicates that the Subsystem represents a Session Bean.
EJBEntityBean	Subsystem	EJBEnterprise-Bean	EJBReentrant EJBPersistence-Type	14-18	Indicates that the Subsystem represents an Entity Bean.
EJBMessage-DrivenBean	Subsystem	EJBTransaction-Capable	EJBMessage-Selector EJBAcknowledge-Mode EJBDestination EJBOnMsgTrans-Attr	19	Indicates that the Subsystem represents a Message-Driven Bean.
EJBPrimary-KeyClass	Class	JavaClass	N/A	20-21	Indicates that the Class represents a Primary Key Class.
EJBAbstract-DataSchema	Class	JavaClass	N/A	22-23	Indicates that the Class represents an Abstract Data Schema.
EJBJar	Package	JavaArchiveFile	EJBSecurityRoles	24	Indicates that the Package represents an EJB-JAR.
EJBRemoteMethod	Operation	JavaMethod	EJBRoleNames EJBTrans-Attribute	N/A	Indicates that the Operation describes an Interface Method.

Stereotype	Base Class	Parent	Tags	Constraints	Description
EJBCreateMethod	Operation	EJBRemote-Method	N/A	25-26	Indicates that the Operation represents a Create-Method.
EJBFindMethod	Operation	EJBRemote-Method EJBQueryMethod	EJBPKMethod	27-29	Indicates that the Operation represents a Find-Method.
EJBHomeMethod	Operation	EJBRemote-Method	N/A	30	Indicates that the Operation represents a Home-Method.
EJBBeanMethod	Operation	JavaMethod	N/A	31	Indicates that the Operation represents an Implementation Class Method.
EJBQueryMethod	Operation	JavaMethod	EJBQuery	32	Indicates that the Operation is specified by an EJB-QL-statement.
EJBSelectMethod	Operation	EJBBeanMethod EJBQueryMethod	N/A	33	Indicates that the Operation represents a Select-Method.
EJBImplementation	Class	JavaClass	N/A	34	Indicating that the Class represents an Implementation Class.
EJBCMP-AssociationEnd	Association-End	N/A	EJBAccess-Modifier EJBCascade-Delete	35	Indicates that the AssociationEnd represents an end of a CMR.

Tabelle 14 – Im erweiterten Profile definierte Tag-Definitions

Tag	Stereotype	Type	Multiplicity	Description
EJBLocal	EJBInterface	Boolean	1	A Boolean value indicating whether or not the interface is of a local type (otherwise it is a remote interface).
EJBRemote-AccessModifiert	EJBCMPField	String	1	An enumeration with values “Set”, “Get”, “Both” or “none”. Indicates how the CMPField is to be published in the remote Component Interface of the Entity Bean.
EJBLocal-AccessModifier	EJBCMPField	String	1	An enumeration with values “Set”, “Get”, “Both” or “none”. Indicates how the CMPField is to be published in the local Component Interface of the Entity Bean.
EJBPKField	EJBCMPField	Boolean	1	A Boolean value indicating whether or not the Attribute is an Primary Key Field.
EJBPKMethod	EJBFindMethod	Boolean	1	A Boolean value indicating whether or not the Find-Method is a findByPrimaryKey()-Method.
EJBEnvEntries	EJBEnterpriseBean	String	*	Contains tuples of the form “**<NAME>**, **<TYPE>**, **<VALUE>**”, designating the environment entries used by the EJB.
EJBResourceRefs	EJBEnterpriseBean	String	*	Contains tuples of the “**<NAME>**, **<TYPE>**, **<AUTH>**”, designating the resource factories used by the EJB.
EJBResource-EnvRefs	EJBEnterpriseBean	String	*	Contains tuples of the “**<NAME>**, **<TYPE>**”, designating the resource environment references used by the EJB.
EJBSecurityID	EJBEnterpriseBean	String	1	Containts the ‘run-as’ security role name of the EJB.
EJBSecurity-RoleRefs	EJBEnterpriseBean	String	*	Contains tuples of the form “**<NAME>**, **<LINK>**”, designating the role names that may invoke all Operations on the EJB.
EJBTransaction-Type	EJBTransaction-Capable	String	1	An enumeration with values “Bean” or “Container”. Indicates whether the transactions of the EJB are managed by the Bean or by its container, respectively.

Tag	Stereotype	Type	Multiplicity	Description
EJBReentrant	EJBEntityBean	Boolean	1	A Boolean value indicating whether or not the Entity Bean can be called reentrant.
EJBPersistence-Type	EJBEntityBean	String	1	An enumeration with values “Bean” or “Container”. Indicates whether the persistence of the Entity Bean is managed by the Bean or by its container, respectively.
EJBSessionType	EJBSessionBean	String	1	An enumeration with values “Stateful” or “Stateless”. Indicates whether or not the Session Bean maintains state.
EJBMessage-Selector	EJBMessage-DrivenBean	String	1	Containts a String, designating the Message-driven bean’s message selector.
EJBAcknowledge-Mode	EJBMessage-DrivenBean	String	1	Containts a String, designating the Message-driven bean’s acknowledgment mode.
EJBDestination	EJBMessage-DrivenBean	String	1	Contains a String of the form “**<TYPE>**, **<DURABILITY>**”, designating the destination of the message-driven bean.
EJBOnMsgTrans-Attr	EJBMessage-DrivenBean	String	1	An enumeration with values “Not Supported” and “Required”. Defines the transaction management policy for the OnMessage-Method.
EJBSecurityRoles	EJBJar	String	*	Contains Strings of the form “**<NAME>**”, designating the security role names that may used within the EJB-Jar.
EJBTransAttribute	EJBRemoteMethod	String	1	An enumeration with values “Not Supported”, “Supports”, “Required”, “RequiresNew”, “Mandatory” or “Never”. Defines the transaction management policy for the Operation.
EJBRoleNames	EJBRemoteMethod	String	*	Contains the security roles that may invoke the Operation.
EJBQuery	EJBQueryMethod	String	1	Contains the EJB-QL Statement that is used by the Operation.
EJBAccessModifier	EJBCMP-AssociationEnd	String	1	An enumeration with values “Set”, “Get”, “Both” or “none”. Indicates how the CMR in Component Interface of the designated Entity Bean.
EJBCascadeDelete	EJBCMP-AssociationEnd	Boolean	1	A Boolean value indicating whether or not the cascade-delete-semantic has to be applied to the AssociationEnd.

Tabelle 15 – Im erweiterten Profile definierte Constraints („Well-Formedness Rules“)

Constraint	Language	Boolean-Expression
01	English	The operations of the class must all be stereotyped as «EJBRemoteMethod».
02	English	The operations of the class must all be stereotypes as «EJBCreateMethod».
03	English	The class must define at least one operation which is stereotyped as «EJBCreateMethod».
04	English	If the corresponding subsystem defines a state (“EJBSessionTyp”=”Stateless”) the class must define exactly one operation which is stereotyped as «EJBCreateMethod» and which does not contain any parameters.
05	English	The operations of the class must all be stereotyped either as «EJBCreateMethod», «EJBFindMethod» or «EJBHomeMethod».
06	English	The class must define at least one operation which is stereotyped as «EJBFindMethod» and which defines the value “True” for its tag“EJBPKMethod”.
07	English	The association must contain exactly two associationends which are both stereotyped as «EJBCMPAssociationEnd».

Constraint	Language	Boolean-Expression
08	English	The subsystem must at least contain or import a class in its realization part which is stereotyped as «EJBImplementation».
09	English	The subsystem must at least contain or import one class in its specification part which must be stereotyped as «EJBComponentInterface».
10	English	For every class in its specification part which is stereotyped as «EJBComponentInterface» must exist exactly one other class in its specification part which is stereotyped as «EJBSessionHomeInterface» and which has the same value for its tag "EJBLocal".
11	English	The maximum of classes in its specification part which are stereotyped as «EJBComponentInterface» is two and they must have different values for their tag "EJBLocal".
12	English	The maximum of classes in its specification part which are stereotyped as «EJBSessionHomeInterface» is two and they must have different values for their tag "EJBLocal".
13	English	The subsystem may not contain or import any class in its specification part which has one of the following stereotypes: «EJBEntityHomeInterface», «EJBAbstractDataSchema» or «EJBPrimayKeyClass».
14	English	The subsystem must at least contain or import one class in its specification part which must be stereotyped as «EJBComponentInterface».
15	English	For every class in its specification part which is stereotyped as «EJBComponentInterface» must exist exactly one other class in its specification part which is stereotyped as «EJBEntityHomeInterface» and which has the same value for its tag "EJBLocal".
16	English	The maximum of classes in its specification part which are stereotyped as «EJBComponentInterface» is two and they must have different values for their tag "EJBLocal".
17	English	The subsystem may not contain or import any class in its specification which is stereotypes as «EJBSessionHomeInterface».
18	English	The maximum of classes in its specification part which are stereotyped as «EJBEntityHomeInterface» is two and they must have different values for their tag "EJBLocal".
19	English	The subsystem may not contain or import any class in its specification part which has one of the following stereotypes: «EJBComponentInterface», «EJBSessionHomeInterface», «EJBEntityHomeInterface», «EJBAbstractDataSchema» or «EJBPrimayKeyClass».
20	English	The class must be a legal Value Type in RMI-IIOP.
21	English	The class must provide suitable implementation of the hashCode() and equals(Object other) methods.
22	English	The class may only have attributes as features. Operations are not allowed.
23	English	Every contained attribute must be stereotyped as «EJBCMPField».
24	English	The package may only contain other packages, subsystems or associations. The contained packages must not be stereotyped as «EJBJar»; the contained subsystems may only have a stereotype which is derived from «EJBEnterpriseBean»; the contained associations may only be stereotyped as «EJBCMReleationship».
25	English	The name of the operation must start with "create".
26	English	The return type of the operation must be the bean's component interface type.
27	English	The name of the operation must start with "find".
28	English	The return type of the operation must either be the entity bean's component interface type or a valid Java Collection Type (e.g. "java.util.Collection").

Constraint	Language	Boolean-Expression
29	English	If the tag "EJBPKMethod" has the value "True" then the name of the operation must be "findByPrimaryKey" and the one and only allowed parameter of the operation is of the type «EJBPrimaryKeyClass» and the return type of the operation is the entity bean's component interface type.
30	English	The name of the operation must not start with "create", "find" or "remove".
31	English	The name of the operation must not start with "ejbCreate", "ejbPostCreate", "ejbHome", "ejbFind" or "ejbRemove".
32	English	The value of the tag "EJBQuery" must be a valid EJB-QL statement.
33	English	The name of the operation must start with "ejbSelect".
34	English	The class may only define operations which are stereotyped as «EJBBeanMethod».
35	English	The only allowed participant of the associationend is a subsystem which is stereotyped as «EJBEntityBean».

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

Tabelle 16- Literaturquellen

[CD00] John Cheesman, John Daniels. UML Components. A simple process for specifying component-based software.Addison-Wesley, 2000.

[Dejo01] Jennifer Dejong. UML Makes Its Way Into the IDE. In: Software Development Times. August 15, 2001.

[DW99] Desmond Fancis D'Souza, Alan Cameron Wills. Objects, Components, and Frameworks with UML. The Catalysis Approach. Addison-Wesley, 1999.

[GJSB00] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. The Java Language Specification. Second Edition. Addison-Wesley, 2000.

[GT00] Volker Gruhn, Andreas Thiel. Komponentenmodelle. DCOM, Java Beans, Enterprise JavaBeans, CORBA. Addison-Wesley, 2000.

[HK99] Martin Hinz, Gerti Kappel. UML@Work. Von der Analyse zur Realisierung.dpunkt.verlag, 1999.

[IBM00] IBM. XMI Application Framework. April 2000

[Klin02] Ronny Klinger. Beispielkatalog für EJB-Entwurfsmuster. Bakkalaureatsarbeit. Technische Universität Dresden, 2002.

[Kobr99]] Kris Kobryn. UML 2001: A Standardisation Odyssey. In: Communications of the ACM, 42:10, 1999, 29-37.

[Kobr00] Kris Kobryn. Modeling Components and Frameworks with UML. In: Communications of the ACM, 43:10, 2000, 31-38.

[META01] META Group. State of the J2EE / EJB Application Server Market in North America. September 2001.

[Mon99] Richard Monson-Haefel. Enterprise JavaBeans. O'Reilly & Associates, 1999.

[MS01] Vlada Matena, Beth Stearns. Applying Enteprise JavaBeans. Component-Based Development for the J2EE Platform. Addison-Wesley, 2001.

[NT01] N. S. Nagaraj, Srinivas Thonse. MDA – Enabling seamless application development. Infosys Technologies Ltd., September 2001.

[Ober00] Sven Obermaier. Entwicklung eines Frameworks für die Codegenerierung am Beispiel SQL. Diplomarbeit. Technische Universität Dresden. 2000.

[OMG99] Object Management Group. Unified Modeling Language Specification. Version 1.3, June 1999.

[OMG00] Object Management Group. UML 2.0 Superstructure. RFP. September 18, 2000.

[OMG00b] Object Management Group. XML Metadata Interchange (XMI) Specification. Version 1.1, November 2000.

[OMG00c] Object Management Group. Model Driven Architecture. White Paper, Draft 3.2, November 2000.

[OMG01] Object Management Group. Unified Modeling Language Specification. Version 1.4, September 2001.

[OMG01b] Object Management Group. Model Driven Architecture (MDA). Draft, July 2001.

[OMG01c] Object Management Group. Developing in OMG's Model-Driven Architecture. White Paper, Revision 2.6, November 2001.

[OMG02] Object Management Group. UML Profile for Enterprise Distributed Object Computing Specification. FTF Final Adopted Specification, February 2002.

[Pleu01] Andreas Pleuß. Werkzeugunterstützung für UML-Profiles. Großer Beleg. Technische Universität Dresden. 2001.

[Poo01] John D. Poole. Model-Driven Architecture: Vision, Standards And Emerging Technologies. Position Paper Submitted to ECOOP 2001. April 2001.

[RAJ02] Ed Roman, Scott Ambler, Tyler Jewell. Mastering Enterprise JavaBeans. Second Edition. Wiley Computer Publishing, 2002.

[RAT01] Rational Software Corporation. UML Profile for EJB. Public Draft, 2001.

[SC01] Soumen Sarkar, Craig Cleveland. Code Generation Using XML based Document Transformation. November 2001

[SUN99] Sun Microsystems. Enterprise JavaBeans Specification. Version 1.1, Final Release, 1999.

[SUN01] Sun Microsystems. Enterprise JavaBeans Specification. Version 2.0, Final Release, 2001.

[SUN02] Sun Microsystems. Java Message Service API Tutorial. 2002.

[SUN02b] Sun Microsystems. Java Metadata Interface (JMI) Specification. Version 1.0, Public Final Draft, March 2002.

[WBGP01] Torben Weis, Christian Becker, Kurt Geihs, Noël Plouzeau. A UML Meta-model for Contract Aware Components.In: Martin Gogolla, Chris Kobryn (Ed.). «UML» 2001 – The Unified Modeling Language. Fourth Int. Conference. Toronto, Canada, October 2001, Springer, 2001.

Tabelle 17- WWW-Quellen

[WWW01] <http://www.rational.com/>.

[WWW02] <http://www.jcp.org/jsr/detail/26.jsp>.

[WWW03] Justin Hill, Timo Salo. Protecting Technology's Integrity: The Java Community Process. <http://www.adtmag.com/joop/article.asp?id=4432&mon=1&yr=2000>.

[WWW04] Jen-Yen Jason Chen, Shih-Chien Chou. An Object-Oriented Analysis Technique Based on the Unified Modeling Language. <http://www.joopmag.com/html/from_pages/article.asp?id=3666&mon=6&yr=2001>.

[WWW05] <http://www.borland.com/devsupport/appserver/faq/45/ejb/home_interface.html>.

[WWW06] Jean-Christophe Cimetiere. Does the App Server Market Still Exist? Conclusion. <http://serverwatch.internet.com/articles/appsmkt/appsmkt_8.html>.

[WWW07] Floyd Marinescu. The State of The J2EE Application Server Market. History, important trends and predictions. <http://www.theserverside.com/resources/article.jsp?l=State-Of-The-Server-Side.>

[WWW08] Carl Zetie, John Meyer, Randy Heffner. Market Overview: Data Modeling Tools and Trends, 2001 Update. <http://www.quest.com/industry_coverage/data_modeling_tools_and_trends.asp>.

[WWW09] Deb Melewski. Application Development Trends. UML gains ground. <http://www.platinum.com/products/reprint/uml_adt.htm>.

[WWW10] W3C. XSL Transformations (XSLT). Version 1.0, Recommendation, November 1999. <http://www.w3.org/TR/xslt>.

[WWW11] <http://xml.apache.org/xalan-j/index.html>.

Erklärung:

Ich erkläre, dass ich die vorliegende Arbeit selbstständig, unter Angabe aller Zitate und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Dresden, den 30. April 2002