EJB-Design und Implementierung in der Praxis

Tobias Frech

7. Oktober 2001

Seminar: Objektorientierter Systementwurf mit Java

Betreuer: Klaus Beschorner

Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Lehrstuhl Technische Informatik
Prof. Dr. W. Rosenstiel
Sand 13
72076 Tübingen

Inhaltsverzeichnis

1 Einleitung **4**

2 Enterprise JavaBeans – Grundlagen **4**

2.1 Die Spezifikation und deren Implementierung 4

2.2 Typen von Enterprise JavaBeans 5

2.3 Schnittstellen 6

2.4 Deskriptoren 8

3 Typenauswahl der Beans beim Design **8**

3.1 Zweckmäßigkeit der verschiedenen Typen 8

3.2 Empfehlung einer Vorgehensweise 9

4 Persistenz **11**

4.1 CMP – Persistenzumsetzung durch den Container 11

4.2 BMP – Persistenzumsetzung durch eigene Programmierung 11

4.3 Vergleich zwischen CMP und BMP 12

5 Vererbung bei Enterprise JavaBeans **14**

5.1 Aussagen der Spezifikation 14

5.2 Realisierungsmöglichkeit 15

5.2.1 Bean-Implementierung 15

5.2.2 Remote-Interface 16

5.2.3 Home-Interface 16

5.2.4 Beispiele 17

5.3 Schwierigkeiten 18

6 Ein praktisches Beispiel **19**

6.1 Anforderungen 19

6.2 Design und Implementierung 20

6.3 Klienten 30

6.4 Testlauf 35

7 Zusammenfassung **37**

A Weitere Listings **38**

Listings

1 vov/PersonBean.java 20
2 vov/StudentBean.java 23
3 vov/VeranstaltungBean.java 24
4 vov/SeminarBean.java 26
5 vov/HauptseminarBean.java 27
6 vov/AnmeldenSessionBean.java 28
7 clients/CreateStudent.java 30
8 clients/CreateHauptseminar.java 32
9 clients/CallAnmeldenSession.java 33
10 vov/Person.java 38
11 vov/PersonHome.java 38
12 vov/Student.java 38
13 vov/StudentHome.java 39
14 vov/Veranstaltung.java 39
15 vov/Seminar.java 40
16 vov/Hauptseminar.java 40
17 vov/HauptseminarHome.java 40
18 vov/AnmeldenSession.java 41
19 vov/AnmeldenSessionHome.java 41
20 META-INF/ejb-jar.xml 42

1 Einleitung

“Todgesagte leben länger” — Das Client-Server-Modell wurde im vergangenen Jahrzehnt – vor allem auf betriebliche Arbeitsplätze bezogen – als Auslaufmodell bezeichnet und feiert nun doch gerade mit der Hochkonjunktur des Internets eine glanzvolle Wiederbelebung. Selbst der neuerdings sehr viel beachteten Peer-to-peer Architektur liegt im Kern eine Client-Server-Architektur zu Grunde. Weiterhin steht auf der Serverseite durch die permanente technische Entwicklung inzwischen soviel Leistung zur Verfügung, dass zu Gunsten von Entwicklungsgeschwindigkeit, Wart- und Erweiterbarkeit vermehrt Frameworks und interpretierende Ablaufmodelle auf dem Server eingesetzt werden können. Diese Frameworks sollen häufig wiederkehrende Infrastrukturfunktionalitäten in Serveranwendungen standardisieren und auch den Programmierer von ihrer zeitaufwendigen und fehleranfälligen Implementierung entbinden.

Genau hier hat SUN basierend auf seinem interpretierenden *Java Runtime Environment (JRE)* im Rahmen der *Java 2 Enterprise Edition (J2EE)* eine Spezifikation für serverseitige JavaBeans geschaffen, die *Enterprise JavaBeans (EJB)* Spezifikation.

Wer den etwas erhöhten Aufwand der Einarbeitung in die Spezifikation auf sich nimmt und einen (die Spezifikation umsetzenden) Applikationsserver zur Verfügung hat, wird dafür mit einer erheblichen Entlastung in der oben genannten Art und Weise belohnt. Eine weitere Hilfestellung in diese Richtung soll dieser Text darstellen, der das Gebiet der Designentscheidungen für die verschiedenen EJB-Typen und vor allem der Realisierung von Vererbung mit EJBs näher beleuchten soll.

2 Enterprise JavaBeans – Grundlagen

2.1 Die Spezifikation und deren Implementierung

SUN hat für die EJB-Plattform nicht ein fertiges Produkt, sondern eine Spezifikation herausgegeben, die in einem offenen Prozess weiterentwickelt wird. Obwohl SUN selbst eine Referenzimplementierung der Spezifikation zur Verfügung stellt, ist diese nach SUNs Angaben nicht für den Produktionseinsatz geeignet. Es liegt also in der Hand der Anbieter von Applikationsservern die Spezifikation umzusetzen und ein produktionstaugliches Produkt auf den Markt zu bringen. Durch diese Konstellation wird zum einen ein stärkerer

Marketingeffekt für die EJB-Plattform erreicht und andererseits kristallisieren sich durch konkurrierende Anbieter "best practices" in der Umsetzung der Spezifikation heraus, die wiederum Eingang in die Produkte der Konkurrenzanbieter finden. Insgesamt existieren deshalb eine Vielzahl von Anbietern für EJB-Server am Markt. Der Benutzer der EJB-Plattform ist somit auch nicht wie bei anderen Frameworks auf einen bestimmten Serverhersteller eingeschränkt, sondern kann dank der durch die Spezifikation gegeben sehr hohen Portabilität mit wenig oder keinem Entwicklungsaufwand den zu Grunde liegenden Server austauschen. Auch wird die Weiterentwicklung der Spezifikation mit großem Interesse durch die Vielzahl der Serveranbieter und die noch größere Anzahl der Benutzer stetig vorangetrieben. In diesem Text wird allerdings ausschließlich auf die Version 1.1 der Spezifikation vom 17.12.1999 Bezug genommen. Alle Literaturverweise beziehen sich deshalb ebenso ausschließlich auf diese *Enterprise JavaBeans Specification, v1.1* [Sun99].

Der in der EJB-Spezifikation abgedeckte Umfang an Funktionalitäten erstreckt sich von der Verwaltung der Datenbankspeicherung von Objekten (Persistenz), über die Bereitstellung von Diensten wie Transaktionen und Sicherheitsmanagement, bis hin zu einer vollständigen Lebenszyklusverwaltung der eingesetzten Objekte. Hierbei werden all diese Funktionalitäten vom EJB-Server zur Verfügung gestellt und können vom Entwickler durch das Ansprechen oder Implementieren von vordefinierten Schnittstellen einfach genutzt werden.

2.2 Typen von Enterprise JavaBeans

Die auf dem Server zu implementierende Funktionalität wird nach dem EJB-Ansatz in einzelne Objekte, sogenannte *Enterprise JavaBeans (EJB)*, zerlegt. Ist ein solches Objekt implementiert und auf dem Server installiert worden, so können sowohl Java-Objekte von außen, als auch andere EJBs innerhalb des Servers auf dieses Objekt zugreifen und die durch dieses Objekt zur Verfügung gestellte Funktionalität nutzen. Hierbei wird die Menge der denkbaren Funktionalitäten nach dem EJB-Konzept in zwei Bereiche unterteilt. Diese beiden Bereiche werden innerhalb des Konzepts durch zwei verschiedene Arten von Beans repräsentiert.

Zum einen sind sogenannte **Entity-Beans** vorgesehen, die vornehmlich zur Datenspeicherung benutzt werden. Instanzen von Entity-Beans werden in der Lebenszyklussteuerung mit einer eindeutigen ID (vergleichbar einem primären Schlüssel) erzeugt. Sie können über

diese ID wiedergefunden werden und somit über mehrere Sitzungen hinweg benutzt werden. Die Eigenschaften von Entity-Beans werden üblicherweise in einer Datenbank gespeichert, so dass sie nach einem Applikationsfehler oder Serverneustart immer noch vorhanden und nutzbar sind. Zu Speicherung in eine Datenbank bietet die EJB-Spezifikation die zwei alternativen Methoden *Container Managed Persistence (CMP)* und *Bean Managed Persistence (BMP)* an, die in Abschnitt 4 vorgestellt werden.

Zum anderen können sogenannte **Session-Beans** genutzt werden, um Instanzen von EJBs für nur eine Sitzung zu erzeugen und zu verwenden. Je nach Serverimplementierung überleben diese Session-Beans zwar auch einen Serverneustart, können aber auf der Klientenseite nicht zwischen zwei unabhängigen Sitzungen weitergegeben werden. Somit bietet sich diese Art Beans dazu an, verarbeitende Funktionalitäten zu beherbergen, welche evtl. selbst zur Datenhaltung auf Entity-Beans zurückgreifen.

Bei den eben vorgestellten Session-Beans findet noch eine feinere Unterscheidung danach statt, ob bei einem Aufruf des Klienten an eine einzelne Session-Bean frühere Aufrufe an die selbe Bean das Ergebnis beeinflussen. Es stellt sich also die Frage ob eine Session-Bean einen Zustand hat, der durch einen Klientenaufruf verändert werden kann. Ist dies nicht der Fall, so können **Stateless Session-Beans** verwendet werden. Eine Instanz kann dabei vom Server bei einem Aufruf dem ersten Klienten zugeordnet werden und beim nächsten Aufruf schon wieder einen anderen Klienten bedienen.

Dagegen sind **Stateful Session-Beans** an einen Klienten gebunden. Sie werden von ihm erzeugt, über eine Sitzung hinweg genutzt und dann auch wieder von ihm zerstört. Dadurch kann eine solche Bean verschiedene Zustände einnehmen, die z.B. durch einen vorhergehenden Aufruf an diese Bean gesetzt wurden, und somit als Reaktion auf verschiedene Zustände verschiedene Ergebnisse liefern.

2.3 Schnittstellen

Für jede der oben vorgestellten Beanarten sind jeweils drei Schnittstellen durch die Spezifikation festgelegt: das *Remote Interface*, das *Home Interface* und das *Bean Interface*.

Das **Bean-Interface** muss dabei durch das Objekt, das die Funktionalität auf dem Server anbieten soll, implementiert werden. Dadurch wird das Objekt zur *Enterprise JavaBean*. Durch diese Schnittstelle kann einerseits die Bean auf den sie umgebenden Server zugreifen und der Server andererseits informiert die Bean über Änderungen im Lebens-

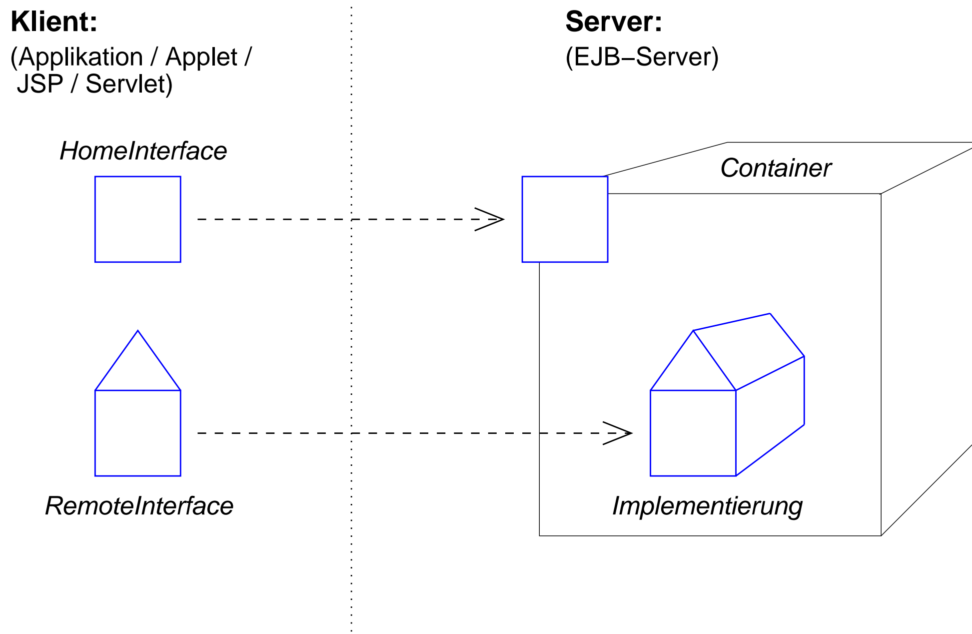

Abbildung 1: Das Home-Interface und Remote-Interface werden auf dem entfernten Klienten verwendet. Über das Home-Interface läßt sich der Lebenzyklus der EJB auf dem Server steuern. Lediglich die Geschäftsdunktionalität ist für das Serverobjekt zu implementieren. Alle anderen Implementierungen übernimmt der Server.

zyklus der Bean, so dass diese darauf reagieren kann. So können Routinen eingebunden werden, die z.B. direkt nach dem Erzeugen einer Instanz oder vor dem Zerstören einer Instanz vom Server aufgerufen werden.

Auf der Seite des Klienten ist zuerst einmal das **Home Interface** interessant. Der Bean-Entwickler legt durch diese Schnittstelle fest, wie neue Instanzen der Bean erzeugt, oder schon bestehende wieder gefunden werden können. Hierbei muss der Bean-Entwickler diese Funktionalität aber nicht selbst implementieren, sondern überlässt dies dem Server. Der Klient kann mit dem Zugriff auf diese Schnittstelle nun den Lebenszyklus der verschiedenen Instanzen der Bean, die zu diesem Home-Interface gehören, steuern (Abbildung 1).

Wird eine neue Instanz erzeugt oder eine bereits bestehende lokalisiert, so erhält der Klient ein Objekt, dass das sogenannte **Remote-Interface** dieser Bean implementiert. Wiederum hat der Bean-Entwickler in dieser Schnittstelle nur die Methoden spezifiziert, die der Klient von der Beaninstanz auf dem Server aufrufen können soll (Abbildung 1). Die konkrete Implementierung, nämlich die Übertragung des Aufrufs an den Server und

die Rückübertragung des Ergebnisses, wird wiederum durch den Server übernommen.

2.4 Deskriptoren

Die fertigen EJB-Objekte mit den Schnittstellendefinitionen und der Implementierung der Funktionalität werden in ein Java-Archiv (jar) zusammengepackt. In dieses Archiv kommt zusätzlich eine Konfigurationsdatei. Diese Datei mit dem Namen `ejb-jar.xml` wird als Deployment-Deskriptor bezeichnet und ist im XML-Format [1] aufgebaut. Die DTD [2] zu dieser Datei wurde von SUN zusammen mit der Spezifikation festgelegt. Ihr Aufbau ist somit für die EJB-Server aller Hersteller identisch. Dieser Deskriptor liefert dem Server Informationen darüber, welche Objekte unter welchem Namen auf dem Server installiert ("deployed") werden sollen. Hier lassen sich auch Transaktionseigenschaften und Persistenz steuern.

Jeder Hersteller eines EJB-Servers bietet aber noch darüber hinausgehende Dienste oder weitere Feineinstellungen für die Leistungsoptimierung an. Diese Einstellungen können auf Grund der vorgegebenen DTD nicht in den `ejb-jar.xml` Deskriptor abgelegt werden, sondern werden üblicherweise durch eine serverspezifische Deskriptordatei festgelegt, die ebenfalls in das Jar-Archiv mit eingebunden wird.

3 Typenauswahl der Beans beim Design

3.1 Zweckmäßigkeit der verschiedenen Typen

In 2.3 wurde bereits darauf eingegangen, dass auf jede der drei Bean-Arten von einem entfernten Klienten zugegriffen werden kann. Damit verfügen alle Bean-Arten über die zentrale Funktionalität für eine Client-Server-Architektur. Grundsätzlich lässt sich auch mit jeder einzelnen Bean-Art die gleich Funktionalität umsetzen wie mit jeder der zwei anderen. Nur der Aufwand zur Umsetzung ist unterschiedlich: Eine *Stateless Session-Bean* kann sich zu jedem Aufruf auch einen Sitzungsschlüssel mitgeben lassen, mit diesem aus

[1] XML steht für *Extensible Markup Language*. Dokumente in diesem Format sind in einem normalen Texteditor lesbar und auf Grund des weit verbreiteten XML-Standards mit vielen XML-Werkzeugen bearbeitbar (siehe auch [W3C2000]).

[2] DTD steht für *Document Type Definition*. Eine DTD legt den Dokumenttyp und damit den Aufbau einer XML-Datei fest. Siehe Abschnitt 2.8 in [W3C2000].

einer zentralen Datenbank den zur Sitzung zugeordneten Zustand herauslesen und somit die Funktionalität einer *Stateful Session Bean* anbieten. In einer *Stateful Session-Bean* wiederum kann das fehlende automatische Datenmanagement ihrer Eigenschaften durch den Server durch zusätzlich programmierte Datenbankzugriffe umgesetzt werden. Sie wäre damit einer *Entity-Bean* gleichwertig.

Theoretisch ist also die Unterscheidung in drei verschiedene Beanarten nicht nötig. Praktisch ist aber eine Auftrennung von einfacheren (Stateless Session) bis hin zu komplexeren Beanarten (Entity-Bean) für die Leistungsoptimierung und die klare Strukturierung der Applikation von Vorteil. Es sollte also bei der praktischen Umsetzung die jeweils passende Beanart gewählt werden, welche die von der Serverseite benötigte Funktionalität unterstützt.

3.2 Empfehlung einer Vorgehensweise

Im Folgenden soll nun als Empfehlung eine Vorgehensweise vorgestellt werden, die anhand einiger Kriterien zur einer Entscheidung führt in welcher Art von Bean eine gewünschte Funktionalität umgesetzt werden soll.

Gemäß dem Paradigma der Objekt-Orientierung werden Funktionen und Daten nicht getrennt, sondern gemeinsam in ein Objekt gekapselt. Sofern also auf Daten aus Entity-Beans zugegriffen wird, sollten die Funktionen wenn möglich direkt in den Entity-Beans implementiert werden.

Dieser Bestrebung stehen aber wiederum Erfordernisse einer verteilten und eventuell parallel läufigen Architektur entgegen. Es ist also eine Abwägung zwischen diesen beiden Polen nötig. Zur Vereinfachung sei hier ein Vorgehensmodell, in dem die einzelnen Punkte nacheinader zu prüfen sind, vorgestellt. Ist die Bedingung für den nächsten Punkt nicht mehr zutreffend, so wird der Vorgang abgebrochen und als Entscheidung der zuletzt gültige Punkt gewählt.

1. Es wird davon ausgegangen, dass für das Erfüllen einer Funktionalität keine oder die nur beim Aufruf übergebenen Daten benötigt werden. Ebenso beeinflusst das Ergebnis des Aufrufs keinen späteren Aufruf der selben Bean. Es kann eine *Stateless Session Bean* verwendet werden, die das Ergebnis direkt berechnet und zurück liefert.

2. Werden weitere Daten als die beim Aufruf übergebenen Daten benötigt oder beein-

flusst dieser Aufruf das Ergebnis eines späteren Aufrufes, so ist eine *Stateful Session-Bean* zu verwenden.

3. Wurden die benötigten Daten evtl. nicht in der gleichen Klientensitzung angelegt oder werden die jetzt aufgerufenen Daten in einer anderen Klientensitzung benötigt, so ist zwingend eine *Entity-Bean* zu verwenden.

4. Innerhalb einer Entity-Bean darf sich immer nur ein Kontrollfaden (*thread of control*) befinden. Finden aus der Entity-Bean Aufrufe an andere Beans statt und es besteht die Möglichkeit, dass wiederum die ursprüngliche Bean aufgerufen werden könnte, so führt dies zu einer Verklemmung (*dead lock*). Ebenso ist bei parallelen Architekturen zu beachten, dass evtl. zwei Entity-Beans versuchen könnten, gegenseitig auf einander zuzugreifen, was ebenfalls zu einer Verklemmungssituation führen kann. Sollen umfangreiche Berechnungen durchgeführt werden, so würden diese bei einer Umsetzung in einer Entity-Bean diese für alle anderen Prozesse für die Dauer der Berechnung sperren. In all diesen Situationen sollte der Wechsel auf eine *Stateful Session-Bean* in Betracht gezogen werden. Diese Session-Bean greift dann für den Datenzugriff jeweils auf die Entity-Beans direkt zu.

5. Werden die für die Funktion benötigen Daten ausschließlich von Entity-Beans bezogen oder beim Aufruf übergeben und beeinflusst die Funktion auch keine Ergebnisse späterer Aufrufe der Session-Bean, so sollte zur Leistungssteigerung wiederum eine *Stateless Session-Bean* gewählt werden.

Die unter 4 dargestellte Situation kann aber auch über eine andere Serverkonfiguration gelöst werden. Jede Bean kann im Deployment-Deskriptor als "reentrant" deklariert werden. Dann ist der gleichzeitige Zugriff auf die Bean nicht nur auf einen Kontrollfaden beschränkt. Die Aufhebung dieser Beschränkung muss aber durch den Applikationsentwickler sorgfältig bedacht werden. Evtl. sind verschiedene kritische Teile der Bean durch entsprechende Deklarationen im Bean-Programmtext zu synchronisieren. Dies bedeutet wiederum zusätzlichen Aufwand für den Bean-Entwickler. Das Ausweichen von Punkt 3 auf den nächsten Punkt 4 kann also durch zusätzlichen Aufwand vermieden werden (vgl. Seite 120, EJB Spec. 1.1).

4 Persistenz

Der Begriff Persistenz findet in vielen Gebieten Anwendung. Hier sei mit Persistenz die Sicherung der konkreten Wertausprägung der Eigenschaften einer Beaninstanz gemeint. Die Sicherung erfolgt hierbei üblicherweise in einer Datenbank. D.h. eine konkrete Beaninstanz kann über mehrere Sitzungen und auch über einen Serverneustart hinweg ihren Zustand vollständig beibehalten. Die EJB-Spezifikation sieht für die Umsetzung der Persistenz bei Entity-Beans zwei unterschiedliche Verfahren vor, die im Folgenden vorgestellt werden.

4.1 CMP – Persistenzumsetzung durch den Container

Bei der Variante der *Container Managed Persistence*, also der durch den Server gesteuerten Persistenz, müssen vom Bean-Entwickler lediglich die Objekteigenschaften angegeben werden, die gesichert werden sollen. Diese Angaben werden wiederum im Deployment-Deskriptor abgelegt. Der Server legt nun selbständig die benötigte Zeile in der entsprechenden Datenbanktabelle an und erzeugt evtl. sogar vorher die für das Objekt benötigte Tabelle automatisch (je nach Serverhersteller). Auch bei Änderungen von Eigenschaftswerten des Objektes werden diese Daten vom Server in der Tabelle wiederum selbsttätig gespeichert oder bei einer Suchanfrage selbsttätig aus der Datenbanktabelle heraus gesucht. Der Bean-Entwickler muss hier also keine einzige Zeile Programmtext für den Datenbankzugriff schreiben.

4.2 BMP – Persistenzumsetzung durch eigene Programmierung

Bei *Bean Managed Persistence* bezieht der Bean-Entwickler zwar immer noch die Verbindung zur Datenbank vom Server, muss aber nun sämtlichen Programmtext einschließlich der SQL-Anweisungen zum Ablegen und zum Einlesen der Objekteigenschaften selbst umsetzen. Der Server stellt Schnittstellen zur Verfügung, durch die der Programmtext für das Sichern und Einlesen der Eigenschaften jeweils festgelegt werden kann. Der Zeitpunkt von Sichern und Einlesen wird also weiterhin durch Aufrufen der vom Entwickler umgesetzten Routinen vom Server gesteuert.

4.3 Vergleich zwischen CMP und BMP

Die beiden zur Verfügung stehenden Verfahren zur Persistenzumsetzung können an Hand der fünf Kriterien Flexibilität, Aufwand, Wartbarkeit, Fehleranfälligkeit und Portierbarkeit verglichen werden.

Flexibilität: Hierbei geht es um die Frage, wieviel Freiheiten der Bean-Entwickler hat, bestimmte Werte auf bestimmte Weise abzulegen und in wie weit er steuern kann, in wieviele und welche Tabellen Daten geschrieben werden bzw. aus welchen gelesen wird.

Aufwand: Der Aufwand bezeichnet qualitativ den Implementierungsaufwand, den der Entwickler zur Umsetzung der Persistenz investieren muss.

Wartbarkeit: Wie gut kann bestehender Programmtext an neue Anforderungen (z.B. eine zusätzliche Objekteigenschaft) angepasst werden?

Fehleranfälligkeit: Hier soll eingeschätzt werden, wie wahrscheinlich es ist, dass der Entwickler bei der Umsetzung der Persistenz Fehler in der Implementierung macht und damit den Aufwand weiter erhöht. Werden die Fehler nicht bemerkt, so korreliert dieses Kriterium auch mit der Wahrscheinlichkeit, dass keine oder falsche Daten in der Datenbank abgelegt werden.

Portierbarkeit: Eine große Stärke der EJB-Spezifikation ist die potentielle Unabhängigkeit von einzelnen Herstellern für Server und Datenbank. Speziell auf den Aspekt der Datenbankportierbarkeit soll hier eingegangen werden.

Kriterium	CMP	BMP
Flexibilität	sehr eingeschränkt, üblicherweise besteht eine 1:1 Relation zwischen einer Bean und einer Tabelle	maximal, mehrere Tabellen pro Bean sind möglich, Reaktion auf Spezialfälle möglich, oft für Integration von Legacy-Systemen benötigt
Aufwand	extrem gering, es müssen nur die zur sichernden Felder im Deskriptor angegeben werden	sehr hoch, der gesamte Datenbank-Programmtext muss erstellt werden, evtl. vereinfacht durch Werkzeugunterstützung
Wartbarkeit	optimal, minimale Konfiguration zieht minimalen Wartungsaufwand nach sich	schwierig, alleine das Hinzufügen eines einzigen Feldes erfordert mehrere konsistente Änderungen an mehreren Stellen im Programmtext, danach muss ein neuer Testzyklus durchlaufen werden
Fehleranfälligkeit	gering, extrem geringer Aufwand verhindert Komplexität und Möglichkeiten Fehler zu machen	hoch, der umgesetzte Datenbank-Programmtext enthält evtl. SQL-Anweisungen, die nicht beim Kompilationslauf überprüft werden, Fehler fallen erst spät auf
Portierbarkeit	hoch, alle vom Server unterstützen Datenbanken können direkt eingesetzt werden	hoch – mittel, bei ausschließlicher Verwendung simpler SQL-Konstrukte sollte eine hohe Portierbarkeit gegeben sein, potentiell können natürlich Eigenheiten einer Datenbank auf Kosten der Portierbarkeit ausgenutzt worden sein

5 Vererbung bei Enterprise JavaBeans

Vererbung ist ein zentrales Prinzip der objekt-orientierten Programmierung. Es trägt zur Übersichtlichkeit und dadurch erheblich zur Wartbarkeit des Programms bei. Dies geschieht durch Unterteilung der Funktionalität in verschiedene Objekte. Die Schnittstellen zwischen diesen Objekten müssen eindeutig als Methodensignaturen[3] definiert werden. Dadurch sind klar definierte Schnittstellen zu kleineren Einheiten an Funktionalität vorhanden, was insgesamt die Übersichtlichkeit eines Programms fördert. Weiterhin findet eine Unterscheidung von allgemeinen und speziellen Fällen dadurch statt, dass Spezialfälle in erbenden Objekten behandelt werden, welche die geerbte Funktionalität erweitern oder modifizieren. Vor allem soll aber durch Vererbung die Wiederverwendbarkeit von Programmtext gefördert werden. Dies wird durch die klar definierten Schnittstellen der Objekte (Methodensignaturen) und die kleinen Einheiten an Funktionalität grundsätzlich gefördert. Die Wahrscheinlichkeit, dass das gleiche Objekt nochmals in einem anderen Projekt wieder Verwendung finden kann, steigt hierdurch. Ich möchte aber auch ganz explizit darauf hinweisen, dass Vererbung (richtig eingesetzt) die Wiederverwendbarkeit innerhalb des *gleichen* Projektes fördert, und somit das Erstellen wieder und wieder gleicher Programmtexte im selben Projekt vermieden werden kann. Genau für diesen Einsatzzweck erweist sich die Kombination von EJBs mit dem Prinzip der Vererbung als äußerst hilfreich.

5.1 Aussagen der Spezifikation

In der EJB-Spezifikation Version 1.1 wird die Verwendung von Vererbung nicht weiter spezifiziert:

> "The current EJB specification does not specify the concept of *component inheritance*." (Seite 290, EJB Spec. 1.1)

Jedoch wird im gleichen Abschnitt darauf verwiesen, dass die Vererbungsmechanismen, die Java von sich aus zur Verfügung stellt, durchaus genutzt werden können, um Schnittstellen oder Implementierungen von Oberklassen zu erben.

Als Konsequenz der prinzipiellen Verwendbarkeit des Vererbungskonzeptes von Java ist Vererbung praktisch einsetzbar. Jedoch mögen sich durch das Fehlen von weiteren Spezi-

[3] Signaturen von Methoden werden hier als der Name einer Methode, die Anzahl der zu übergebenden Parameter und den Parametertypen definiert.

fikationen zu diesem Thema von einem zum anderen Serverhersteller kleine Unterschiede ergeben. Das grundlegende Prinzip bleibt aber gleich, so dass beim Auftreten solcher Unstimmigkeiten diese mit wenig Aufwand beseitigbar sein sollten.

5.2 Realisierungsmöglichkeit

Die Vorgehensweise zur Vererbung mit EJBs sei nun jeweils für die drei zu implementierenden Schnittstellen vorgestellt. Im Anschluss finden sich Beispiele für einen hilfreichen Einsatz von Vererbung.

5.2.1 Bean-Implementierung

Laut Spezifikation hat die Implementierung der Bean nicht nur die eigentliche Funktionalität der Bean, sondern auch die in der Schnittstelle `javax.ejb.EntityBean` (für eine Entity-Bean) aufgeführten Methoden zu implementieren. Diese Methoden können bis auf eine Ausnahme für eine minimale Implementierung leer gelassen werden. Bei einer Entity-Bean muss die ID (Primärschlüssel) der Bean nach dem Aufruf einer `ejbCreate()` Methode in den Eigenschaften der Bean abgelegt worden sein. Da einige Server zur Leistungssteigerung Objektpools anlegen, wird bei der Erzeugung einer "neuen" Entity-Bean Instanz eventuell auch eine alte Instanz wiederverwendet. Dadurch können die Eigenschaften der Instanz nicht die normalerweise gewohnten *null* Werte enthalten, sondern evtl. alte Daten einer anderen Instanz. Es empfiehlt sich also alle Eigenschaften einer Entity-Bean in der `ejbCreate()` Methode mit Standardwerten zu initialisieren.

Soll nun eine Unterklasse einer bereits bestehenden Bean gebildet werden, so kann diese einfach von der Oberklasse per *extends* erben. Es ist nicht nötig, nochmals die `javax.ejb.EntityBean` Schnittstelle zu implementieren, da die Implementierung dieser Schnittstelle ja auch von der Oberklasse geerbt wird. Die Spezifikation erlaubt ausdrücklich eine solche indirekte Implementierung der Schnittstelle (vgl. Seite 121, EJB Spec. 1.1). Soll die Bean nur in ihrer Funktionalität von der Oberklasse abweichen, so genügt es, die entsprechenden Methoden der Oberklasse zu überschreiben. Hierbei kann bei einer Erweiterung der Funktionalität auch über das `super()` Konstrukt auf die bereits vorhandenen Implementierungen trotz des Überschreibens zugegriffen werden. Das Hinzufügen von weiteren Methoden ist ebenfalls möglich. Jedoch ist hierbei zu beachten, das Remote-Interface der Unterklasse ebenfalls um diese Methoden zu erweitern (siehe Abschnitt *Remote-Interface*).

Werden der Unterklasse weitere Eigenschaften hinzugefügt oder soll eine weitere Create Methode eingeführt werden, so ist die entsprechende Create Methode zu implementieren und in dieser die neuen Eigenschaften ebenfalls mit Standardwerten zu initialisieren. Der Server ruft (entgegen der bei Konstruktoren üblichen Semantik) von sich aus *nicht* die Create Methode der Oberklasse auf, was sich aber wiederum durch das `super().ejbCreate()` Konstrukt leicht kompensieren lässt. An dieser Stelle sei ausdrücklich davor gewarnt, anstatt der `ejbCreate()` Methoden zu versuchen, den Klassenkonstruktor einzusetzen. Wie bereits erwähnt fallen auf Grund von Serverimplementierungen und Maßnahmen zur Leistungssteigerung der Aufruf der ejbCreate Methode beim neu Anlegen einer Bean und der Aufruf des Klassenkonstruktors weder zeitlich noch von der Anzahl her zusammen.

5.2.2 Remote-Interface

Das Remote-Interface, das die nach außen für den Klienten sichtbaren Methoden enthält, erweitert laut Spezifikation per `extends` die Schnittstelle `javax.ejb.EJBObject`. Die Verwendung der Vererbung verläuft hier absolut parallel wie bei der Bean-Implementierung. Ein erbendes Remote-Interface erbt per `extends` von einem schon programmierten Remote-Interface ebenfalls die `javax.ejb.EJBObject` Schnittstelle und muss diese nicht getrennt aufführen. Abhängig vom Serverhersteller kann es aber hier notwendig sein, doch überall nicht nur vom Super-Interface sondern ebenfalls explizit von `javax.ejb.EJBObject` zu erben. Dies ist der Fall, wenn der Serverhersteller (aus meiner Sicht unsinniger Weise) überprüft, ob direkt von `javax.ejb.EJBObject` abgeleitet wurde statt nur zu überprüfen, ob das Remote-Interface der Bedingung `instanceof javax.ejb.EJBObject` genügt. Für die praktische Anwendung macht dies einen zusätzlichen Eingabeaufwand von "`,javax.ejb.EJBObject`" für jedes Remote-Interface aus.

5.2.3 Home-Interface

Für das Home-Interface kann keine Vererbung eingesetzt werden. Prinzipiell erlaubt die Spezifikation zwar auch Superklassen für das Home-Interface, jedoch tritt bei einer Vererbungsstruktur analog derer, die bei der Bean-Implementierung und dem Remote Interface verwendet werden, ein Konflikt der Methodensignaturen auf. Eine angenommene erbende Klasse eines Home-Interface erbt auch deren `create()` und `findByPrimaryKey()` Methoden. Die Signatur dieser Methoden wird auf Grund eines oft einheitlichen Objekttyps für

die ID des Objektes auch für die erbende Klasse gleich bleiben. Allerdings muss die erbende Klasse als Rückgabewert den eigenen Objekttyp für diese Methoden spezifizieren. Dies wird in Java als Konflikt zur Wahrung der semantischen Eindeutigkeit nicht erlaubt. Damit kann eine Vererbungsstruktur dieser Art für die Home-Interfaces nicht realisiert werden. Allerdings könnten auf Grund der anderen Rückgabetypen des erbenden Home-Interfaces sowieso keine geerbten Methodenspezifikationen verwendet werden. Möchte man in dieser Situation trotzdem von einem nötigen Zusatzaufwand sprechen, so ist dieser sehr gering, da die definierten Methoden ja vom Server und nicht vom Bean-Entwickler implementiert werden müssen.

5.2.4 Beispiele

Die folgenden drei kurzen Beispiel sollen illustrieren, wie Vererbung die Wiederverwendbarkeit innerhalb des selben Projektes fördern kann.

Müssen einige oder alle Entity-Beans eine gleichbleibende Funktionalität als Methodenzugriff zur Verfügung haben, die aber nicht in eine Session Bean ausgelagert werden soll, so können die entsprechenden Beans von einer Oberklasse erben, die diese Methode zur Verfügung stellt. Als Beispiel ließe sich hier ein Methodenaufruf zur Ausgabe einer Meldung an einen Protokollierdienst nennen.

Soll für eine Methode einer Unterklasse zwar die gleiche Funktionalität wie die der Oberklasse gelten, die übergebenen Objekte oder Werte aber noch zusätzlichen Bedingungen genügen, so kann in der Unterklasse der entsprechende Methodenaufruf die zusätzlichen Bedingungen überprüfen und bei Einhaltung der Restriktionen per `super()` an die Oberklasse weitergeben. So muss die entsprechende Funktionalität nur einmal in der Oberklasse implementiert werden. Als Beispiel sei hier der Vorgang des Anmelden einer Person (Objekt) zu einer Veranstaltungen (Oberklasse) genannt, wobei sich zu speziellen Veranstaltungen (Unterklasse) nur Personen mit bestimmten Eigenschaften anmelden dürfen (Eigenschaften des Objektes werden in der Unterklasse abgefragt, bei Erfolg wir an die eigentliche Anmeldung in der Oberklasse delegiert).

Letztendlich sei auf die sehr hilfreiche Unterstützung der Vererbung zur Implementierung der `ejbCreate()` Methoden einer Entity-Bean hingewiesen. Wie bereits geschildert müssen bei Unterklassen mit zusätzlichen Eigenschaften diese in der `ejbCreate()` Methode der Unterklasse initialisiert werden. Damit nicht der Programmtext zur Initialisierung der geerbten Eigenschaften dupliziert werden muss, kann ein `super().ejbCreate()` Auf-

ruf verwendet werden. Dies ist besonders hilfreich, wenn die Initialisierung über einfache Ein-Zeilen-Zuweisungen hinausgeht, wie z.B. bei der Verwendung einer `HashMap`.

5.3 Schwierigkeiten

Die Anwendung der Vererbungskonzeptes auf EJBs kann in Kombination mit der Persistenzsteuerung auf Schwierigkeiten stoßen. Unterklassen erben alle Methoden ihrer Oberklassen und können somit auch mit dem Klassentyp der Oberklasse verwendet werden ("upcast"). Wird eine solches Objekt, als Oberklasse übergeben (obwohl es eigentlich eine Unterklasse darstellt) und als Referenz in einem anderen Objekt abgelegt, so entsteht bei einem späteren Zugriff auf dieses Objekt ein Problem. Die Referenz auf das Objekt wurde in der Zwischenzeit in einer Datenbank abgelegt und wird nun wieder eingelesen. So ist nun zwar die ID des referenzierten Objektes bekannt, jedoch nicht seine genau Unterklasse. Ein Aufruf der `findByPrimaryKey()` Methode der Oberklasse wird das Objekt nicht lokalisieren können. Die technischen Details sollen an einem Beispiel erläutert werden:

- Von einer Entity-Bean *Person* seien die beiden Unterklassen *Dozent* und *Student* abgeleitet.

- Bei einer Anmeldung einer *Person* bei einer Entity-Bean *Veranstaltung* können nun sowohl *Dozenten* als auch *Studenten* angemeldet werden.

- Die Entity-Bean *Veranstaltung* legt nur die IDs der angemeldeten Personen in der Datenbank ab.

- Bei einem Aufruf zur Ausgabe aller Teilnehmer an die *Veranstaltung* versucht diese, alle *Personen* als EJB zu lokalisieren und wendet sich dazu mit den IDs an das Home-Interface der *Person* Entity-Bean.

- Da aber die Entity-Bean *Dozent* in der Datenbanktabelle "Dozent" gespeichert wurde und ebenso *Student* in der Tabelle "Student", die Entity-Bean *Person* in der Datenbanktabelle "Person" sucht, wird die Anfrage nach IDs von Dozenten oder Studenten an *Person* kein Ergebnis liefern.

- Für ein erfolgreiches Wiederauffinden müsste also die Bean *Veranstaltung* alle möglichen Unterklassen kennen. Zusätzlich müsste bekannt sein, von welcher konkreten

Unterklasse die momentan gesuchte Bean ist, um sich so an das Home-Interface dieser EJB wenden zu können.

Bei der Verwendung von CMP für die Persistenz ist diese Problematik auf Grund der festen Relation zwischen einer CMP-Bean und einer Datenbanktabelle unumgänglich. Für BMP-Entity-Beans ist theoretisch die Ablage der Daten derart denkbar, dass für jede Unterklasse nur die zusätzlichen Eigenschaften in einer neuen Tabelle für die Unterklasse abgelegt werden. Die geerbten Eigenschaften werden weiterhin in der Tabelle der Oberklasse abgelegt (auch rekursiv). Hierdurch wäre es bei einem konkreten Zugriff auf die Unterklasse immer noch möglich, alle benötigten Daten der Datenbank aus den beteiligten Tabellen zu lesen, trotzdem aber bei einer Suchanfrage an die Oberklasse diese in der Tabelle der Oberklasse mit den dann benötigten Informationen zu finden. Das dann instantiierte Objekt der Oberklasse hätte nun aber nicht mehr die eventuell überladenen Methoden der ursprünglichen Unterklasse, sondern eben die Methoden der Oberklasse. Damit wäre hier die Konsistenz der Daten wiederum nicht gewährleistet.

6 Ein praktisches Beispiel

Die vorgestellten Konzepte und Vorgehensweisen zur Kombination von Vererbung und EJBs sollen nun in diesem Abschnitt durch ein lauffähiges Beispiel verdeutlicht werden.

Das Beispiel soll einen sehr kleinen Teil einer *Vorlesungsverwaltung* wiederspiegeln. Der als Beispiel implementierte Teil soll es ermöglichen, sowohl Studenten als auch Veranstaltungen in das System einzutragen und Studenten bei Veranstaltungen anmelden zu können.

6.1 Anforderungen

Die Anforderungen wurden so konstruiert, dass sich einige praktische Einsatzmöglichkeiten der Vererbung bei EJBs demonstrieren lassen:

- *Personen* haben einen Namen.
- *Studenten* sind Personen, haben darüber hinaus aber noch eine Matrikelnummer und *können* ein Vordiplom erworben haben.

- *Veranstaltungen* finden in einem Raum zu einem bestimmten Termin statt. Jede Veranstaltung hat eine Liste von Teilnehmern. Dieser Liste können *Studenten* hinzugefügt werden.

- *Seminare* sind *Veranstaltungen*. Sie haben eine Liste der Vortragenden *Personen* zu der *Personen* hinzugefügt werden können.

- *Hauptseminare* sind *Seminare*. Sie haben jedoch die Einschränkung, dass nur Studenten zur Teilnahme und somit zur Anmeldung zugelassen sind, die bereits das Vordiplom erworben haben.

- Um keinen zusätzlichen Aufwand zur Synchronisierung betreiben zu müssen, soll keine EJB als "reentrant" deklariert werden müssen. Eine Änderung der Objektstruktur wird hier vorgezogen.

6.2 Design und Implementierung

Aus den vorgestellten Anforderungen ergeben sich die folgenden Objekte mit den beschriebenen Eigenschaften als Entity-Beans: Person, Student, Veranstaltung, Seminar, Hauptseminar.

In Abbildung 2 ist die Vererbungshierarchie und in Abbildung 3 sind die Eigenschaften und Methoden der Objektklassen dargestellt.

Im Folgenden findet sich der für die Person-Entity-Bean nötige Programmtext des Serverobjektes:

Listing 1: vov/PersonBean.java

```
package vov;

import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import java.rmi.RemoteException;

public class PersonBean
    implements EntityBean
{

    EntityContext ctx;
```

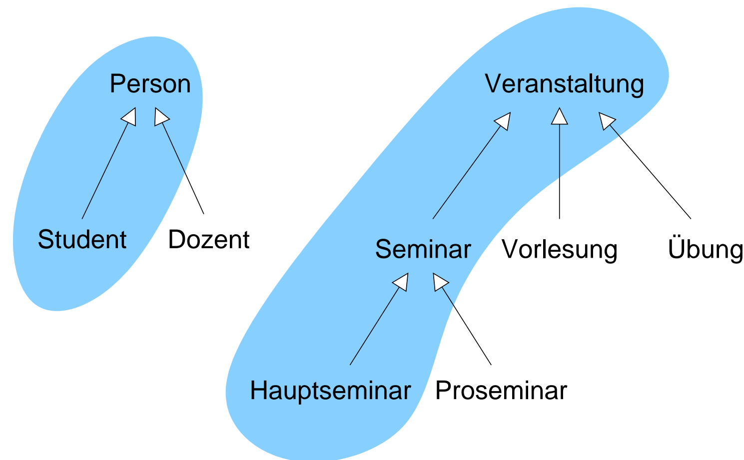

Abbildung 2: Hier sind einige Beispiele an möglichen Klassen und deren Vererbungstruktur dargestellt. Die farblich hinterlegten Klassen sollen in dem hier dargestellten Beispiel implementiert werden.

```
///// Data fields

public Integer Id;
public String Name;

///// implementing EntityBean ////////////////////////////////////

public Integer ejbCreate (Integer newID)
{
    System.out.println("PersonBean.ejbCreate(): mit ID "
                       + newID +" aufgerufen");
    this.Id = newID;
    return null;
}

public void ejbPostCreate(Integer newID) { }

public void setEntityContext(EntityContext ctx) { this.ctx = ctx; }
public void unsetEntityContext() { ctx = null; }

public void ejbActivate() { }
public void ejbPassivate() { }
public void ejbLoad() { }
public void ejbStore() { }
```

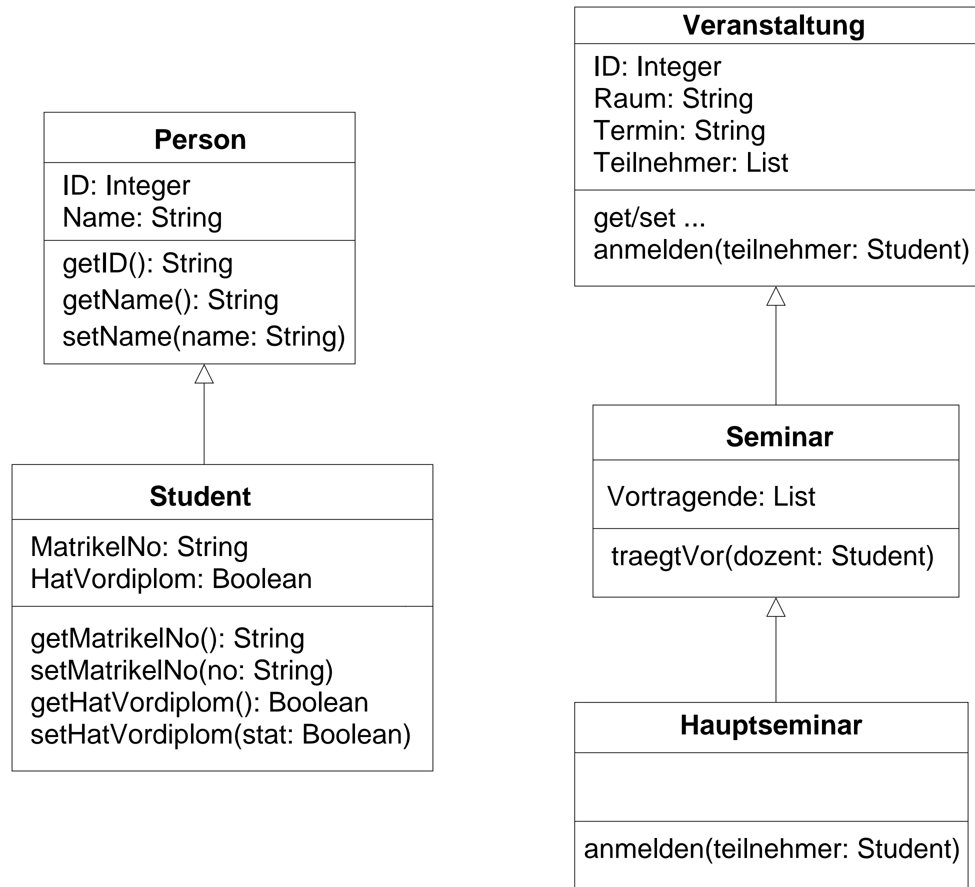

Abbildung 3: Die für das hier dargestellte Beispiel relevanten Klassen mit ihren Eigenschaften und Methoden.

```
public void ejbRemove() { }


///// get-/setters //////////////////////////////////////////////

public Integer getId() {
    return this.Id;
}

public String getName() {
    return this.Name;
}

public void setName(String s) {
    System.out.println("PersonBean.setName(): setze Name auf "+s);
```

```
        this.Name = s;
    }

}
```

Die EJB Person implementiert die vorgeschriebene Schnittstelle `javax.ejb.EntityBean`. Für diese Entity-Bean, wie auch für alle weiteren Programmtexte ist jeweils hier nur der Programmtext der Serverobjekte aufgeführt. Das benötigten Remote-Interface und Home-Interface finden sich im Anhang A.

Die Entity-Bean `Student` erbt von der Entity-Bean Person. Somit muss sie nicht selbst `javax.ejb.EntityBean` implementieren, sondern erbt diese Implementierung mit. In der `ejbCreate()` Methode werden zuerst die neu hinzugekommen Eigenschaften initialisiert. Durch den Aufruf super.ejbCreate(newID) wird die bereits vorhanden Implementierung der Oberklasse zur Initialisierung der ebenfalls geerbten Eigenschaften ausgenutzt. Weiterhin müssen die Datenzugriffsfunktionen für die neuen Eigenschaften ergänzt werden.

Listing 2: vov/StudentBean.java

```
package vov;

public class StudentBean
    extends PersonBean
{

    public String MatrikelNo;
    public boolean HatVordiplom;

    ///// extending implementation of EntityBean //////////////////////

    public Integer ejbCreate (Integer newID)
    {
        System.out.println("StudentBean.ejbCreate(): mit ID "
                           +newID+" aufgerufen");
        this.MatrikelNo = null;
        this.HatVordiplom = false;
        return super.ejbCreate(newID);
    }

    ///// get-/setters ///////////////////////////////////////////////

    public String getMatrikelNo() {
```

```
        return this.MatrikelNo;
    }

    public void setMatrikelNo(String s) {
        System.out.println("StudentBean.setMatrikelNo(): "
                           +"setze Matrikelnummer auf "+s);
        this.MatrikelNo = s;
    }

    public boolean getHatVordiplom() {
        return this.HatVordiplom;
    }

    public void setHatVordiplom(boolean b) {
        System.out.println("StudentBean.setHatVordiplom(): "
                           +"setze Vordiplom auf "+b);
        this.HatVordiplom = b;
    }

}
```

Die Entity-Bean `Veranstaltung` implementiert analog zur Person-EJB die `javax.ejb.EntityBean` Schnittstelle und die benötigten Datenzugriffsfunktionen. Zum Eintragen eines Studenten in die Teilnehmerliste wird die Methode `anmelden()` verwendet.

Listing 3: vov/VeranstaltungBean.java

```
package vov;

import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import java.rmi.RemoteException;
import java.util.List;
import java.util.Vector;

public class VeranstaltungBean
    implements EntityBean
{

    EntityContext ctx;

    ///// Data fields
```

```
public Integer Id;
public String Raum;
public String Termin;
public List Teilnehmer;

///// implementing EntityBean //////////////////////////////////////

public Integer ejbCreate (Integer newID)
{
    System.out.println("VeranstaltungBean.ejbCreate(): mit ID "
                        +newID+" aufgerufen");
    this.Id = newID;
    this.Raum = null;
    this.Termin = null;
    this.Teilnehmer = new Vector();
    return null;
}

public void ejbPostCreate(Integer newID) { }

public void setEntityContext(EntityContext ctx) { this.ctx = ctx; }
public void unsetEntityContext() { ctx = null; }

public void ejbActivate() { }
public void ejbPassivate() { }
public void ejbLoad() { }
public void ejbStore() { }
public void ejbRemove() { }


///// get-/setters ///////////////////////////////////////////////

public Integer getId() {
    return this.Id;
}

public String getRaum() {
    return this.Raum;
}

public void setRaum(String s) {
    this.Raum = s;
```

```
    }

    public String getTermin() {
        return this.Termin;
    }

    public void setTermin(String s) {
        this.Termin = s;
    }

    public void anmelden(Student s)
        throws NichtZugelassenException, RemoteException
    {
        if (s != null) {
            System.out.println("VeranstaltungBean.anmelden(): "
                               +"Trage Teilnehmer "+s.getName()+" ein");
            Teilnehmer.add(s);
        }
    }

}
```

Die Seminar-EJB erbt von der Veranstaltung-EJB. Wiederum wird in der ejbCreate Methode die neu hinzugekommene Eigenschaft initialisiert und danach an die `ejbCreate()` Methode der Oberklasse delegiert. Zusätzlich kann über die `traegtVor()` Methode eine Person in die Liste der Vortragenden eingetragen werden. Zu beachten ist, dass die `anmelden()` Methode der Veranstaltung EJB unverändert geerbt wird.

Listing 4: vov/SeminarBean.java

```
package vov;

import java.util.List;
import java.util.Vector;
import java.rmi.RemoteException;

public class SeminarBean
    extends VeranstaltungBean
{

    public List Vortragende;

```

```
    ///// extending implementation of EntityBean //////////////////////

    public Integer ejbCreate (Integer newID)
    {
        System.out.println("SeminarBean.ejbCreate(): mit ID "
                              +newID+" aufgerufen");
        this.Vortragende = new Vector();
        return super.ejbCreate(newID);
    }

    ///// business methods ///////////////////////////////////////////

    public void traegtVor(Person p) throws RemoteException {
        if (p != null) {
            System.out.println("SerminarBean.traegtVor(): Trage "
                                  +"Vortagenden "+p.getName()+" ein");
            Vortragende.add(p);
        }
    }

}
```

Die Hauptseminar-EJB erbt von der Seminar-EJB und führt keine neuen Eigenschaften ein. Deshalb kann auch die geerbte `ejbCreate()` Methode unverändert belassen werden. Die in den Anforderungen geforderte Einschränkung, dass nur Studenten mit einem Vordiplom sich an Hauptseminaren anmelden dürfen, wird in einer neuen Methode `anmelden()` Me überprüft. Ist diese Bedingung nicht gegeben, so wird der Anmeldevorgang mit einer entsprechenden Ausnahme abgebrochen. Ist die Bedingung jedoch erfüllt, so wird mit `super.anmelden()` an die Oberklasse delegiert und somit deren bereits implementierte Funktionalität ausgenutzt.

Listing 5: vov/HauptseminarBean.java

```
package vov;

import java.rmi.RemoteException;
import java.util.List;
import java.util.Vector;

public class HauptseminarBean
    extends SeminarBean
```

```
{

    ///// business methods ////////////////////////////////////////////

    // ueberlade fruehere Implementierung
    public void anmelden(Student stud)
        throws NichtZugelassenException, RemoteException
    {
        if (stud != null) {
            System.out.println("HauptseminarBean.anmelden() mit Student "
                                   +stud.getName()+" aufgerufen.");

            if (stud.getHatVordiplom() == true ){
                super.anmelden(stud);
            } else {
                throw new NichtZugelassenException("Student hat "
                                                      +"kein Vordiplom!");
            }
        }
    }
}
```

Als Session-Bean *AnmeldenSession*, wird der Anmeldungsvorgang für einen Studenten bei einem Hauptseminar umgesetzt. Diese Funktionalität wurde nicht als Methode in der Entity-Bean *Student* implementiert, weil diese Bean die Entity-Bean *Hauptseminar* aufrufen würde, welche wiederum bei der aufrufenden Bean *Student* die Eigenschaft `HatVordiplom` abfragen würde. Für den letzten Schritt müsste aber die Entity-Bean Student als "reentrant" deklariert werden, da zum Zeitpunkt dieser Abfrage der anfängliche Anmelden-Methodenaufruf in der selben Bean noch nicht beendet ist.

In der Methode `anmeldenBeiHauptseminar()` erhält die Session-Bean die ID des Studenten und die ID des Hauptseminars, bei dem der Studenten angemeldet werden soll. Über die beiden IDs wird mittels der Home Interfaces und der `findByPrimaryKey()` Methoden die jeweilige Entity-Bean lokalisiert und dann die `anmelden()` Methode der Hauptseminar Entity-Bean aufgerufen.

Listing 6: vov/AnmeldenSessionBean.java

```
package vov;

import java.rmi.RemoteException;
```

```
import javax.rmi.PortableRemoteObject;
import javax.naming.InitialContext;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class AnmeldenSessionBean
    implements SessionBean
{
    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext sc) {}

    public void anmeldenBeiHauptseminar(Integer StudentID,
                                        Integer HauptsemID)
        throws RemoteException, NichtZugelassenException
    {
        System.out.println("AnmeldenSessionBean."+
                           "anmeldenBeiHauptseminar():"+
                           " aufgerufen mit StudentID "+StudentID+
                           " und HauptseminarID "+HauptsemID);

        try {
            InitialContext jndiContext = new InitialContext();
            Object ref   = jndiContext.lookup("/Hauptseminar");
            HauptseminarHome hsHome = (HauptseminarHome)
                PortableRemoteObject.narrow (ref,
                                             HauptseminarHome.class);

            ref   = jndiContext.lookup("/Student");
            StudentHome studentHome = (StudentHome)
                PortableRemoteObject.narrow (ref, StudentHome.class);

            Hauptseminar hs = hsHome.findByPrimaryKey(HauptsemID);
            Student student = studentHome.findByPrimaryKey(StudentID);

            hs.anmelden(student);

        } catch (NichtZugelassenException e) {
            throw e;
        } catch (Exception ex) {
            throw new RemoteException(ex.getMessage());
```

```
        }
    }
}
```

Die mit der Vererbung in 5.3 beschriebene Problematik kann durch den Leser an Hand des Beispiels praktisch nachvollzogen werden. Dazu ist zuerst eine Methode zu implementieren, welche die an einer Veranstaltung teilnehmenden Studenten auflistet. Dieses ist ohne Modifikationen machbar. Jedoch tritt beim Versuch eine Liste der Vortragenden eines Seminars zu erstellen die beschriebende Problematik auf, wenn davon ausgegangen werden muss, dass sowohl Studenten als auch Dozenten in der Liste der Vortragenden auftauchen können.

6.3 Klienten

Zur Erzeugung der entsprechenden Instanzen der Entity-Beans und der späteren Anmeldung eines Studenten bei einem Hauptseminar sind im Folgenden die Programmtexte der dazu benötigten Beispielklienten aufgelistet.

Die Klientenapplikation `CreateStudent` verlangt als Eingabeparameter eine noch nicht vorhandene ID, den Namen des Studenten und die Angabe (j/n), ob der Student bereits das Vordiplom erworben hat. Die ID für Entity-Beans wird in produktiven System meist nicht von außen vorgegeben, sondern intern erzeugt. Zur Reduzierung der Komplexität, wird hier die ID von außen vorgegeben. Jede ID darf dabei pro EJB-Klasse nur einmal vorkommen und verwendet werden. Im Programmtext wird zuerst der Lookup-Service über die JNDI API aufgerufen, und von diesem eine Referenz auf das Home-Interface der Student EJB geholt. Über dieses Home-Interface kann nun eine neue Instanz einer Entity-Bean `Student` erzeugt und danach die spezifizierten Werte gesetzt werden.

Listing 7: clients/CreateStudent.java

```
import vov.Student;
import vov.StudentHome;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class CreateStudent
{
    public static void main(String[] args) {
```

```
        if (args.length != 3) {
            System.out.println("Brauche␣Parameter:␣StudentID␣"
                              +"StudentName␣HatVodiplom");
            System.exit(0);
        }

        try {
            System.out.print("InitialContext␣und␣lookup␣...");

            InitialContext  jndiContext = new InitialContext();
            Object ref   = jndiContext.lookup("/Student");

            StudentHome home = (StudentHome)
                PortableRemoteObject.narrow (ref, StudentHome.class);

            System.out.println(" fertig ");

            ////////////////////////////////////////////

            System.out.print("Erzeuge␣Entity␣und␣"
                            +"setze␣Eigenschaften␣...");

            Student student = home.create(Integer.valueOf(args[0]));
            student.setName(args[1]);
            if ("t".equalsIgnoreCase(args[2]) |
                "j".equalsIgnoreCase(args[2])) {
                student.setHatVordiplom(true);
            } else {
                student.setHatVordiplom(false);
            }

            System.out.println(" fertig ");

        } catch (Exception e) {
            System.out.println("Fehler!␣War␣wohl␣nix:␣"+e.getMessage());
            e.printStackTrace();
        }
    }
}
```

Die Klientenapplikation `CreateHauptseminar` arbeitet absolut analog zu `CreateStudent`,

nur dass hier als Parameter die ID und der Raum des Hauptseminars erwartet werden.

Listing 8: clients/CreateHauptseminar.java

```
import vov.Hauptseminar;
import vov.HauptseminarHome;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class CreateHauptseminar
{
    public static void main(String[] args) {

        if (args.length != 2) {
            System.out.println("Brauche Parameter: "+
                              "HauptseminarID Raum");
            System.exit(0);
        }

        try {
            System.out.print("InitialContext und lookup ...");

            InitialContext jndiContext = new InitialContext();
            Object ref   = jndiContext.lookup("/Hauptseminar");

            HauptseminarHome home = (HauptseminarHome)
                PortableRemoteObject.narrow (ref,
                                    HauptseminarHome.class);

            System.out.println("fertig");

            ////////////////////////////////////////////

            System.out.print("Erzeuge Entity "
                            +"und setze Eigenschaften ...");

            Hauptseminar hs = home.create(Integer.valueOf(args[0]));
            hs.setRaum(args[1]);

            System.out.println("fertig");

        } catch (Exception e) {
            System.out.println("Fehler! War wohl nix: "+e.getMessage());
            e.printStackTrace();
```

```
        }
    }
}
```

Die Klientenapplikation `CallAnmeldenSession` führt nun die eigentliche Anmeldung von Studenten bei Hauptseminaren durch. Sie erwartet dazu als Parameter die eine ID des Studenten und die ID des Hauptseminars bei dem er angemeldet werden soll. Es wird nun eine neue AnmeldenSession Bean erzeugt und dieser die beiden IDs zur Durchführung der Anmeldung übergeben (siehe Abbildung 4).

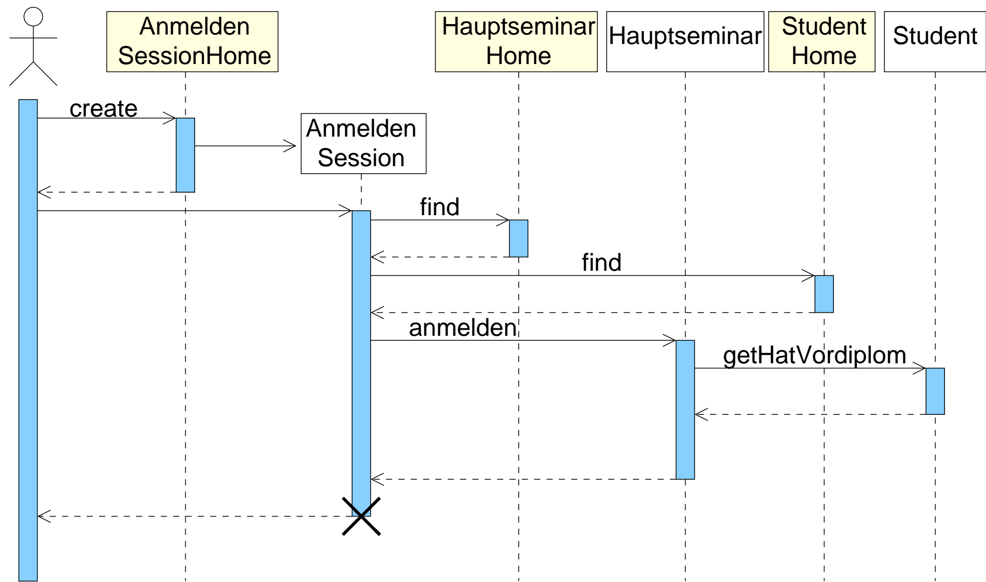

Abbildung 4: Sequenzdiagramm des Anmeldevorgangs

Listing 9: clients/CallAnmeldenSession.java

```
import vov.AnmeldenSession;
import vov.AnmeldenSessionHome;
import vov.NichtZugelassenException;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class CallAnmeldenSession
{
    public static void main(String[] args) {

        if (args.length != 2) {
```

```
            System.out.println("Brauche␣Parameter:"
                    +"␣StudentID␣HauptseminarID");
            System.exit(0);
        }

        try {
            System.out.print("InitialContext␣und␣lookup␣...");

            InitialContext  jndiContext = new InitialContext();
            Object ref   = jndiContext.lookup("/AnmeldenSession");

            AnmeldenSessionHome home = (AnmeldenSessionHome)
                PortableRemoteObject.narrow (ref,
                                    AnmeldenSessionHome.class);

            System.out.println("fertig");

            ////////////////////////////////////////////

            System.out.print("Erzeuge␣Session"+
                    "␣und␣rufe␣anmelden␣auf␣...");

            AnmeldenSession session = home.create();
            session.anmeldenBeiHauptseminar(Integer.valueOf(args[0]),
                                    Integer.valueOf(args[1]));

            System.out.println("fertig");


        } catch (NichtZugelassenException e) {
            System.out.println("Anmeldung␣fehlgeschlagen.");
            System.out.println("Sorry,␣dieser␣Student␣darf␣nicht␣"
                    +"an␣diesem␣Hauptseminar␣teilnehmen!␣");
        } catch (Exception e) {
            System.out.println("Fehler!␣War␣wohl␣nix:␣"+e.getMessage());
            e.printStackTrace();
        }
    }
}
```

6.4 Testlauf

Werden die EJB Klassen kompiliert und mit einer `ejb-jar.xml` Datei (siehe Anhang A) gemeinsam in ein Jar-Archiv gepackt, so kann dieses Archiv auf dem Server installiert ("deployed") werden.

Es werden nun nacheinander die entsprechenden Klienten mit Beispielparametern aufgerufen.

Klient:

```
> java CreateStudent 10 Tobias j
InitialContext und lookup ...fertig
Erzeuge Entity und setze Eigenschaften ...fertig
```

Server:

```
[Student] StudentBean.ejbCreate(): mit ID 10 aufgerufen
[Student] PersonBean.ejbCreate(): mit ID 10 aufgerufen
[Student] PersonBean.setName(): setze Name auf Tobias
[Student] StudentBean.setHatVordiplom(): setze Vordiplom auf true
```

Der Klient ist erfolgreich abgelaufen und wie an der Serverausgabe zu sehen ist, wird auf dem Server unter Einsatz der vererbten Methoden die Entity-Bean entsprechend angelegt.

Klient:

```
> java CreateStudent 10 Pumuckel n
InitialContext und lookup ...fertig
Erzeuge Entity und setze Eigenschaften ...fertig
```

Server:

```
[Student] StudentBean.ejbCreate(): mit ID 11 aufgerufen
[Student] PersonBean.ejbCreate(): mit ID 11 aufgerufen
[Student] PersonBean.setName(): setze Name auf Pumuckel
[Student] StudentBean.setHatVordiplom(): setze Vordiplom auf false
```

Ebenso wird eine Hauptseminar Entity-Bean mit dem entsprechenden Klienten angelegt:

Klient:

```
> java CreateHauptseminar 555 C2P23
InitialContext und lookup ...fertig
Erzeuge Entity und setze Eigenschaften ...fertig
```

Server:

```
[Hauptseminar] SeminarBean.ejbCreate(): mit ID 555 aufgerufen
[Hauptseminar] VeranstaltungBean.ejbCreate(): mit ID 555 aufgerufen
```

Auch hier zeigt die Serverausgabe, dass vererbter Programmcode aufgerufen und ausgeführt wurde.

Letztendlich wird versucht, die beiden Studenten am Hauptseminar anzumelden:

Klient:

```
> java CallAnmeldenSession 10 555
InitialContext und lookup ...fertig
Erzeuge Session und rufe anmelden auf ...fertig
```

Server:

```
[AnmeldenSession] AnmeldenSessionBean.anmeldenBeiHauptseminar(): aufgerufen mit
StudentID 10 und HauptseminarID 555
[Hauptseminar] HauptseminarBean.anmelden() mit Student Tobias aufgerufen.
[Hauptseminar] VeranstaltungBean.anmelden(): Trage Teilnehmer Tobias ein
```

Klient:

```
> java CallAnmeldenSession 11 555
InitialContext und lookup ...fertig
Erzeuge Session und rufe anmelden auf ...Anmeldung fehlgeschlagen.
Sorry, dieser Student darf nicht an diesem Hauptseminar teilnehmen!
```

Server:

```
[AnmeldenSession] AnmeldenSessionBean.anmeldenBeiHauptseminar():
aufgerufen mit StudentID 11 und HauptseminarID 555
[Hauptseminar] HauptseminarBean.anmelden() mit Student Pumuckel aufgerufen.
```

Der Student ohne Vordiplom "Pumuckel" wurde durch die zusätzliche Restriktion, wie in den Anforderungen spezifiziert, nicht zum Hauptseminar zugelassen.

7 Zusammenfassung

Es wurden kurz die hier relevanten Grundlagen von *Enterprise JavaBeans (EJB)* vorgestellt. Eine vorgestellte Vorgehensweise kann die beschriebenen Konzepte ergänzen, nach denen aus den drei EJB-Typen Stateless-Session, Stateful-Session und Entity-Bean der passendste Typ zur Umsetzung eine bestimmten Teilfunktionalität ausgewählt werden kann. Anschließend wurden die Vor- und Nachteile der beiden Möglichkeiten *Container Managed Persistence* und *Bean Managed Persistence* einander gegenüber gestellt. Der Schwerpunkt des Textes bildet die Erörterung in wie weit das Konzept der Vererbung in Kombination mit EJBs eingesetzt werden kann, was die Vorteile und praktischen Einsatzmöglichkeiten dieser Kombination sind. Eine Diskussion der dabei auftretenden Schwierigkeiten zeigt, dass diese Kombination für ein Projekt durchaus erhebliche Verbesserungen bei der Implementierungseffizienz und der Übersichtlichkeit des Programmtextes bringen kann. Jedoch müssen die Konsequenzen der beschrieben Schwierigkeiten möglichst frühzeitig analysiert werden, um nicht in eine Sackgasse hinein zu entwickeln. Die theoretischen Ausführungen wurden abschließend durch das Vorstellen eines lauffähigen Beispiels im letzten Abschnitt verdeutlicht.

A Weitere Listings

Listing 10: vov/Person.java

```
package vov;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Person
    extends EJBObject
{
    ///// get−/setters ///////////////////////////////////////////////

    public Integer getId()        throws RemoteException;
    public String getName()       throws RemoteException;
    public void setName(String s) throws RemoteException;
}
```

Listing 11: vov/PersonHome.java

```
package vov;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import java.util.Collection;

public interface PersonHome
    extends EJBHome
{
    public Person create(Integer id)
        throws RemoteException, CreateException;

    public Person findByPrimaryKey (Integer id)
        throws RemoteException, FinderException;

    public Collection findAll()
        throws RemoteException, FinderException;
}
```

Listing 12: vov/Student.java

```
package vov;

import java.rmi.RemoteException;

public interface Student
    extends Person
{
    public String getMatrikelNo()             throws RemoteException;
    public void setMatrikelNo(String s)       throws RemoteException;
    public boolean getHatVordiplom()          throws RemoteException;
    public void setHatVordiplom(boolean b)    throws RemoteException;
}
```

Listing 13: vov/StudentHome.java

```
package vov;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import java.util.Collection;

public interface StudentHome
    extends EJBHome
{
    public Student create(Integer id)
        throws RemoteException, CreateException;

    public Student findByPrimaryKey (Integer id)
        throws RemoteException, FinderException;

    public Collection findAll()
        throws RemoteException, FinderException;
}
```

Listing 14: vov/Veranstaltung.java

```
package vov;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Veranstaltung
```

```
    extends EJBObject
{
    ///// get-/setters ////////////////////////////////////////////////

    public Integer getId()          throws RemoteException;
    public String getRaum()         throws RemoteException;
    public void setRaum(String s)   throws RemoteException;
    public String getTermin()       throws RemoteException;
    public void setTermin(String s) throws RemoteException;

    public void anmelden(Student s) throws RemoteException,
                                           NichtZugelassenException;
}
```

Listing 15: vov/Seminar.java

```
package vov;

import java.rmi.RemoteException;

public interface Seminar
    extends Veranstaltung
{
    public void traegtVor(Person p) throws RemoteException;
}
```

Listing 16: vov/Hauptseminar.java

```
package vov;

import java.rmi.RemoteException;

public interface Hauptseminar
    extends Seminar
{
    // anmelden() wird in Implementierung ueberladen !
}
```

Listing 17: vov/HauptseminarHome.java

```
package vov;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
```

```
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import java.util.Collection;

public interface HauptseminarHome
    extends EJBHome
{
    public Hauptseminar create(Integer id)
        throws RemoteException, CreateException;

    public Hauptseminar findByPrimaryKey (Integer id)
        throws RemoteException, FinderException;

    public Collection findAll()
        throws RemoteException, FinderException;
}
```

Listing 18: vov/AnmeldenSession.java

```
package vov;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface AnmeldenSession
    extends EJBObject
{
    public void anmeldenBeiHauptseminar(Integer StudentID,
                                        Integer HauptsemID)
        throws RemoteException, NichtZugelassenException;
}
```

Listing 19: vov/AnmeldenSessionHome.java

```
package vov;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

public interface AnmeldenSessionHome
    extends EJBHome
{
    AnmeldenSession create() throws RemoteException, CreateException;
```

```
}
```

Listing 20: META-INF/ejb-jar.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<ejb-jar>
     <enterprise-beans>
       <entity>
         <ejb-name>Student</ejb-name>
         <home>vov.StudentHome</home>
         <remote>vov.Student</remote>
         <ejb-class>vov.StudentBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>Id</field-name></cmp-field>
         <cmp-field><field-name>Name</field-name></cmp-field>
         <cmp-field><field-name>MatrikelNo</field-name></cmp-field>
         <cmp-field><field-name>HatVordiplom</field-name></cmp-field>
         <primkey-field>Id</primkey-field>
       </entity>

       <entity>
         <ejb-name>Hauptseminar</ejb-name>
         <home>vov.HauptseminarHome</home>
         <remote>vov.Hauptseminar</remote>
         <ejb-class>vov.HauptseminarBean</ejb-class>
         <persistence-type>Container</persistence-type>
         <prim-key-class>java.lang.Integer</prim-key-class>
         <reentrant>False</reentrant>
         <cmp-field><field-name>Id</field-name></cmp-field>
         <cmp-field><field-name>Raum</field-name></cmp-field>
         <cmp-field><field-name>Termin</field-name></cmp-field>
         <cmp-field><field-name>Teilnehmer</field-name></cmp-field>
         <cmp-field><field-name>Vortragende</field-name></cmp-field>
         <primkey-field>Id</primkey-field>
       </entity>

       <session>
         <ejb-name>AnmeldenSession</ejb-name>
         <home>vov.AnmeldenSessionHome</home>
         <remote>vov.AnmeldenSession</remote>
```

```
      <ejb-class>vov.AnmeldenSessionBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Bean</transaction-type>
    </session>

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

Literatur

[Sun99] Vlada Matena und Mark Hapner, *Enterprise JavaBeans Specification, v1.1*, SUN Microsystems, 17.12.1999
http://java.sun.com/products/ejb/docs.html

[W3C2000] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen und Eve Maler, *Extensible Markup Language (XML) 1.0*, W3C, 16. Oktober 2000
http://www.w3.org/TR/2000/REC-xml-20001006