MASTER THESIS

zur Erlangung des akademischen Grades
„Master of Science in Engineering“
im Studiengang Multimedia und Softwareentwicklung

Umsetzung einer auf Java EE 6 basierenden generischen Persistenzumgebung für das quelloffene Framework JVx

Ausgeführt von: Stefan Wurm, Bakk. rer. soc. oec.
Personenkennzeichen: 1110299037

1. Begutachter: DI Dr. Markus Schordan
2. Begutachter: Roland Hörmann

Wien, 15. Mai 2012

FACHHOCHSCHULE
TECHNIKUM WIEN

Eidesstattliche Erklärung

„Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbständig angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Ort, Datum

Unterschrift

Kurzfassung

Das JVx Framework ist ein quelloffenes Framework für Java Plattformen, welches die Entwicklung von professionellen und effizienten Datenbankanwendungen in kurzer Zeit und mit wenig Quellcode ermöglicht. Das Ziel von JVx ist, dem Entwickler mehr Zeit für anwendungsspezifische Anforderungen zu geben, indem Lösungen für häufig wiederkehrende Aufgaben angeboten werden. Das Framework beruht auf einem Drei-Schichten-Modell: Einer Datenhaltungsschicht, einer Anwendungsschicht über einen integrierten Kommunikationsserver und einer technologieunabhängigen Darstellungsschicht. Diese technologieunabhängige Erstellung von Anwendermasken ermöglicht es, eine geschriebene Anwendung ohne Codeänderungen als Java Desktop Anwendung oder als Web Anwendung zu verwenden.

Um JVx für die Java Welt noch attraktiver zu gestalten, kann die Umsetzung einer bereits bekannten und etablierten Spezifikation wesentlich von Vorteil sein. Java Enterprise Edition (Java EE) ist die Spezifikation einer Softwarearchitektur für transaktionsbasierte Anwendungsentwicklung. Durch die Integration einer auf Java EE 6 basierenden generischen Persistenzumgebung in JVx soll die Attraktivität des Frameworks gesteigert werden und Motivation für weitere Entwicklungen liefern. Ziel dieser Masterarbeit ist es, diese Spezifikation bzw. benötigte Teilspezifikation zu analysieren, sowie die Integration in das JVx Framework zu planen und umzusetzen.

Schlagwörter: JVx, Java Enterprise Edition, Java Persistence API

Abstract

JVx is an open source framework for Java platforms which supports the development of lean database applications in a short time. Common periodical functions are generally included in JVx to offer the developer more time for custom designed requirements. The design of the framework is a multi-tier architecture made of a database tier, an enterprise tier with an integrated communication server and a technological independent presentation tier. The development of such technologically independent application screens allows the developer to start the same application as a desktop or a web application without changing the source code.

In order to increase the attractiveness of the JVx framework, the implementation of a known and established specification can help. Java Enterprise Edition (Java EE) is the specification for a software-architecture, for transaction-based application development. The integration of a generic persistence environment based on Java EE 6 increases the attractiveness of the framework and provides the motivation for further extensions. This master thesis aim is to analyse the specification and to plan and implement the integration of Java EE 6 parts into JVx.

Keywords: JVx, Java Enterprise Edition, Java Persistence API

Danksagung

Ein besonderer Dank gilt dem Team von SIB Visions für die tatkräftige Unterstützung bei der Entwicklungsarbeit. Vor allem möchte ich mich aber bei meinen beiden Betreuern Herrn DI Dr. Markus Schordan und Herrn Roland Hörmann für deren Engagement und das konstruktive Feedback an meiner Arbeit bedanken.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Aufbau der Arbeit . . . 1

2 Grundlagen zu Java EE **3**
2.1 Java Plattformen . . . 3
2.2 Client-Server Systeme . . . 3
2.3 Architektur von Java EE . . . 4
2.3.1 Mehrschichtenarchitektur . . . 5
2.4 Infrastruktur von Java EE . . . 7
2.4.1 Java EE Container . . . 7

3 Persistenzumgebung der Java Enterprise Edition **9**
3.1 Objektrelationale Abbildung . . . 9
3.2 Entitäten . . . 10
3.2.1 Primärschlüssel einer Entität . . . 11
3.2.2 Lebenszyklus einer Entität . . . 14
3.2.3 Beziehungen zwischen Entitäten . . . 15
3.3 Verwaltung der Entitäten . . . 17
3.3.1 Entitäten-Manager . . . 18
3.3.2 Persistenzkontext . . . 19
3.3.3 Persistenzeinheit . . . 19
3.3.4 Persistenzanbieter . . . 20
3.3.5 Abfragen der Entitäten . . . 20

4 Das quelloffene Framework JVx **25**
4.1 Infrastruktur und Architektur von JVx . . . 25
4.1.1 Darstellungsschicht . . . 26
4.1.2 Anwendungsschicht . . . 26
4.1.3 Datenhaltungsschicht . . . 27
4.2 Persistenzumgebung von JVx . . . 27
4.2.1 Aufbau der Metadaten . . . 28
4.2.2 Funktionen von IStorage . . . 29
4.2.3 Einschränkungskriterien und Sortierung von Abfragen . . . 29
4.3 Funktionalität von JVx anhand eines Beispiels . . . 31
4.3.1 Aufbau und Funktionalität der Beispielanwendung . . . 31
4.3.2 Entwicklung der Client-Schicht . . . 32
4.3.3 Entwicklung der Server-Schicht . . . 39

5 Generische Integration von JPA in JVx **42**
5.1 Gegenüberstellung von JVx und Java EE . . . 42

5.1.1 JVx Client-Schicht zu Java EE Web-Schicht 43
5.1.2 JVx Server-Schicht zu Java EE Business-Schicht 43
5.2 Qualitätskriterien zur Integration von JPA in JVx 45
5.2.1 Unabhängigkeit des JDBC Treibers 45
5.2.2 Datenbankunabhängkeit der Abfragesprache 46
5.2.3 Optimierung der Performance 46
5.2.4 Erzeugung und Änderung des Domänenmodells aus den Entitäten 46
5.2.5 Beibehalten der Architektur einer JVx Anwendung 46
5.2.6 Unterstützung der bestehenden JVx Funktionalitäten 47
5.2.7 Unterstützung verschiedenster Persistenzanbieter 47
5.2.8 Datenbankunabhängige Erzeugung der Metadaten 47
5.2.9 Einfache Verwendung von JPA in JVx 47
5.2.10 Verwendung von eigenen DAOs bzw. EAOs 47
5.3 Integration von JPA in JVx 48
5.3.1 Die Klasse JPAStorage 49
5.3.2 Die Klasse JPAAccess 49
5.3.3 Externes EAO 50
5.3.4 Das generische EAO Design 52
5.3.5 Die Klasse JPAServerMetaData 53

6 Beispielanwendung für den Einsatz von JPA in JVx 55
6.1 Entwickeln der Entitäten 55
6.2 Entwickeln der EAOs 57
6.3 Erzeugen der Persistenzeinheit 58
6.4 Erzeugen des Entitäten-Managers 58
6.5 Funktionen für den Zugriff auf JPAStorage 59
6.6 Evaluierung der Qualitätskriterien zur Integration von JPA in JVx 61

7 Zusammenfassung und Ausblick 63

Literatur 65

Internetquellen 68

Abbildungsverzeichnis 70

Tabellenverzeichnis 71

Abkürzungsverzeichnis 72

1 Einleitung

Die immer größer werdenden Anforderungen an eine zu entwickelnde Software erfordern den Einsatz von Software Spezifikationen und unterstützenden Frameworks. Software Frameworks liefern bereits fertige Komponenten für typische Anforderungen und verringern dadurch den Entwicklungsaufwand. Dies führt zu kürzeren Projektlaufzeiten und geringeren Projektkosten.

Es gibt bereits eine Vielzahl an unterschiedlichsten Software Frameworks, welche für verschiedenste Anforderungen geeignet sind. Die Etablierung eines kleinen, noch unbekannten Frameworks am Markt ist daher kein leichtes Unterfangen. Das JVx Framework ist ein quelloffenes Framework für Java Plattformen, welches in diese Kategorie der kleineren und eher unbekannteren Frameworks fällt. JVx ermöglicht die Entwicklung von Datenbankanwendungen mit wenig Quellcode in kürzester Zeit und beruht auf einem Drei-Schichten-Modell: Auf einer Datenhaltungsschicht, einer Anwendungsschicht über einen integrierten Kommunikationsserver und einer technologieunabhängigen Darstellungsschicht.

Um ein quelloffenes Framework für Entwicklungen attraktiv zu gestalten, kann neben einer ausgezeichnete Qualität, guter und verständlicher Dokumentation und einer engagierten Community im Hintergrund, die Umsetzung einer bereits etablierten und bekannten Spezifikation wesentlich von Vorteil sein. Spezifikationen wie Java Enterprise Edition (Java EE) bieten einige Vorteile und sind in der Java Welt sehr bekannt. Es gibt bereits zahlreiche Java EE Frameworks, welche auf dieser Spezifikation aufsetzen und die Entwicklung von Java Anwendung wesentlich vereinfachen. Durch die Integration des Java Persistence API (JPA), welches eine Teilspezifikation von Java EE ist, wäre es für JVx Anwendungen möglich, einige dieser Java EE Frameworks sinnvoll einzusetzen. Ein Beispiel dafür wäre die Verwendung von Reporting Frameworks wie JasperReport, welche einen wesentlichen Mehrwert bei JVx Anwendungen bringen würden. Die Integration ist aber nicht nur für JVx Anwendungen ein Gewinn, sondern auch für Webanwendungen welche JPA als Persistenz verwenden. Mittels JVx ist dadurch eine einfache Entwicklung von Backendanwendungen möglich, ohne das zugrundeliegende Domänenmodell anpassen zu müssen. Der Einsatz von JPA in JVx soll also die Attraktivität des Frameworks steigern und die Motivation für weitere Erweiterungen liefern.

1.1 Aufbau der Arbeit

Die Masterarbeit gliedert sich in sechs Kapitel mit einigen Unterkapiteln.

Das erste Kapitel **Grundlagen zu Java EE** bietet einen groben Überblick über den Aufbau, die Architektur und Infrastruktur von Java Enterprise Edition.

Im Kapitel **Persistenzumgebung der Java Enterprise Edition** wird die Funktionalität des Java Persistence API, welches eine Teilspezifikation von Java EE ist, näher beschrieben. Es werden die Themen objektrelationale Abbildung, Entitäten, Beziehungen zwischen Entitäten,

Lebenszyklus einer Entität und die Verwaltung dieser über den Entitäten-Manager erklärt.

Das Kapitel **Das quelloffene Framework JVx** beschreibt den Aufbau, die Architektur und die Funktionsweise von JVx. Besonderer Fokus wird auf das Persistenz API von JVx gelegt, welches die Schnittstelle zur Anknüpfung an das Java Persistence API ist.

Generische Integration von JPA in JVx stellt die beiden Technologien Java EE und JVx gegenüber und beschreibt die generische Integration von JPA in das Framework JVx. Es werden Qualitätskriterien aufgelistet, welche durch die Integration ermöglicht bzw. verbessert werden. Diese Kriterien werden dann in Kapitel 6 evaluiert.

Das vorletzten Kapitel **Beispielanwendung für den Einsatz von JPA in JVx** beschreibt wie das Framework durch Nutzung des JPA Standards verwendet werden kann.

Das letzte Kapitel **Zusammenfassung und Ausblick** liefert einen Überblick und beschreibt aufgetretene und bestehende Probleme. Es werden Ansätze und Ideen für weitere Entwicklungen bzw. Integrationsmöglichkeiten in JVx geliefert.

2 Grundlagen zu Java Enterprise Edition

Bei Java handelt es sich um eine objektorientierte Programmiersprache und ein eingetragenes Markenzeichen der Firma Sun Microsoft (seit 2010 von Oracle) [1]. Java ist aber auch eine Plattform, welche eine Umgebung in der in Java programmierte Anwendungen laufen zur Verfügung stellt. Diese Plattformen, Java SE, Java EE und Java ME, werden kurz im nachfolgenden Abschnitt 2.1 angeführt. Nachfolgend wird in dieser Masterarbeit aber nur näher auf die Java Plattform Java EE eingegangen. Alle anderen Plattformen sind in den angeführten Spezifikationen beschrieben.

Die Java EE ist der Industriestandard für transaktionsbasierte, modular verteilte, mehrschichtige Java Anwendungen. Die erste Java EE Version wurde im Dezember 1999 veröffentlicht und war bis zur Version 5 unter dem Namen J2EE (Java 2 Platform, Enterprise Edition) bekannt. Die aktuelle Version von Java EE ist die Version 6.0. [2]

2.1 Java Plattformen

Die Java Plattformen ermöglichen den Einsatz von Java Anwendungen auf unterschiedlichsten Endgeräten mit unterschiedlichsten Charakteristiken. Siehe dazu Tabelle 2.1.

Alle diese Java Plattformen bestehen aus einer Java Virtuellen Maschine (JVM) [4] und aus Programmschnittstellen (*engl.: Application Programming Interfaces, API*). Die JVM ist dabei die Schnittstelle zwischen der Maschine und dem auf dieser Maschine laufenden Betriebssystem. Dadurch wird die Plattformunabhängigkeit von Java Anwendungen gewährleistet. Ein API besteht aus einzelnen Softwarekomponenten, welche die Erstellung von anderen Softwarekomponenten bzw. Anwendungen ermöglichen bzw. erleichtern.

2.2 Client-Server Systeme

Die Java EE dient dazu, verteilte Client-Server Systeme auf Basis von Java zu bauen. Unter einem verteilten System versteht man eine Anwendung, welche als Gesamtheit nicht in einem einzelnen Programm agiert, sondern sich aus der Interaktion mit unterschiedlichsten Diensten und Programmen ergibt. Ein Client-Server System ist ein verteiltes System, welches darauf beruht, dass von einem Client (z.B.: Webbrowser) eine Anfrage an einen Server über ein bestimmtes Kommunikationsprotokoll (TCP/IP, HTTP, usw.) gesendet wird. Dieser Server bearbeitet die Anfrage und sendet eine Antwort an den Client zurück. [5]

Natürlich ist eine verteilte Anwendung um einiges komplexer zu erstellen als normale Desktop-Anwendungen. Daher bringt dieses Prinzip auch einige Änderungen hinsichtlich der Programmierung mit sich. Die Java EE stellt ein standardisiertes Modell bereit, um diese Herausforderungen zu lösen.

Java Plattform	Verwendung
Standard Edition (Java SE)	Java Platform, Standard Edition (Java SE): Dient als Grundlage für Java EE und Java ME und ermöglicht durch die enthaltenen Programmschnittstellen die Entwicklung von Anwendungen für den direkten Einsatz auf Computern. [31] "*Java Platform, Standard Edition (Java SE) lets you develop and deploy Java applications on desktops and servers, as well as in today's demanding embedded environments. Java offers the rich user interface, performance, versatility, portability, and security that today's applications require.*"[32]
Enterprise Edition (Java EE)	Enthält zusätzlich zu den Funktionalitäten von Java SE Programmschnittstellen für transaktionsbasierte, mehrschichtige Unternehmens- und Webanwendungen. [3] "*Java Platform, Enterprise Edition (Java EE) 6 is the industry standard for enterprise Java computing.*"[33]
Micro Edition (Java ME)	Ist eine Plattform zur Entwicklung von Java Anwendungen für mobile Endgeräte. [34] "*Java Platform, Micro Edition (Java ME) provides a robust, flexible environment for applications running on mobile and other embedded devices: mobile phones, personal digital assistants (PDAs), TV set-top boxes, and printers. Java ME includes flexible user interfaces, robust security, built-in network protocols, and support for networked and offline applications that can be downloaded dynamically. Applications based on Java ME are portable across many devices, yet leverage each device's native capabilities.*"[35]

Tabelle 2.1: Beschreibung der Java Plattformen

2.3 Architektur von Java EE

Das Architektur Design von Java EE ermöglicht es, eine Anwendung in von sich aus isolierte funktionale Bereiche zu gliedern und diese auf unterschiedlichsten Servern auszuführen. Aber auch wenn es vorkommt, dass die gesamte Anwendung auf nur einem Server ausgeführt wird, ist es von Vorteil, diese Anwendung in mehrere Schichten mit klar getrennten Aufgabenbereichen zu unterteilen.

2.3.1 Mehrschichtenarchitektur

Eine typische Mehrschichtenanwendung ist die Dreischichtenanwendung, welche aus einer Darstellungsschicht (Client Maschine), einer Anwendungsschicht (Java EE Server) und einer Datenhaltungsschicht (Datenbank Server) besteht. Sie dazu Abbildung 2.1.

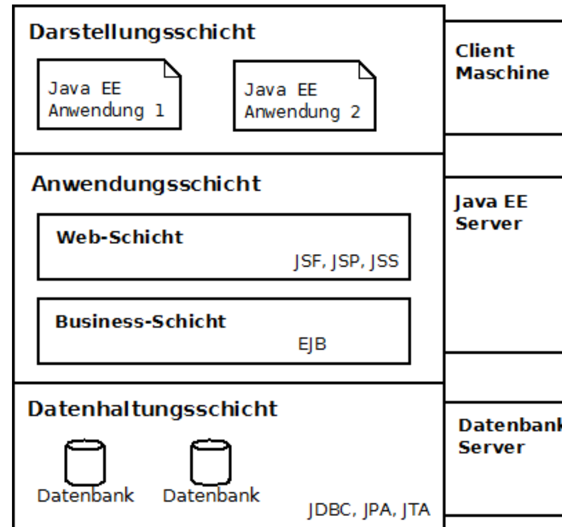

Abbildung 2.1: Mehrschichtenarchitektur unter Java EE
[6], S. 41, eigene Überarbeitung

Datenhaltungsschicht

Die unterste Schicht ist die Datenhaltungsschicht und sie ist für die permanente Speicherung der Daten verantwortlich. Diese Schicht wird unter Java EE auch als EIS-Schicht (Enterprise Information System) bezeichnet, welche aus Datenbank Servern, ERP (Enterprise Resource Planning) Systemen oder anderen Datenspeichern besteht. Die gespeicherten Daten werden in Form von Java-Objekten an die Anwendungsschicht übergeben. Tabelle 2.2 enthält Java EE Technologien, welche in der sogenannten EIS Schicht verwendet werden.

Anwendungsschicht

Die Anwendungsschicht enthält die Geschäftslogik und ist für die Aufbereitung der Daten verantwortlich. Diese Schicht gliedert sich bei Java EE in eine Business-Schicht und eine Web-Schicht, welche auf dem Java EE Server ausgeführt werden.

Die Web-Schicht kann als serverseitige Darstellungsschicht betrachtet werden und enthält Komponenten und Technologien, um eine Schnittstelle zwischen dem Client und der Business-Schicht zu ermöglicht. Folgende in der Tabelle 2.3 angeführten Java EE Technologien kommen in dieser Schicht zum Einsatz.

Die Business-Schicht stellt die Businessfunktionalitäten zur Verfügung und deckt damit die Geschäftslogik der Anwendung ab. In dieser Schicht kommen die in Tabelle 2.4 angeführten Java EE Technologien zum Einsatz.

Java EE Komponente	Verwendung
Java Database Connectivity API (JDBC)	Bietet eine einheitliche Schnittstelle für relationale Datenbanken zu unterschiedlichsten Datenbankanbietern. [7]
Java Persistence API	Ist eine einheitliche und datenbankunabhängige Schnittstelle für objektrelationale Abbildung. Mehr dazu in Kapitel 3.
Java EE Connector Architecture	Ist eine Programmschnittstelle, um Systeme transparent in die Java EE Plattform zu integrieren. [8]
Java Transaction API (JTA)	Bildet die Basis für ein verteiltes Transaktionsmanagement. [9]

Tabelle 2.2: Java EE 6 Komponenten der EIS-Schicht

Java EE Komponente	Verwendung
Java Servlet API (JSS)	Ist die Basistechnologie zum Aufbau dynamischer Webseiten. Diese Servlets liegen im »Web Container« (siehe Abschnitt 2.4.1) am Java EE Server und beantworten die Anfragen vom Webbrowser. [10]
Java Server Faces (JSF)	Ist ein Standard zur Entwicklung von grafischen Benutzeroberflächen für Webanwendungen. [11]
Java Server Pages (JSP)	Diese Technologie baut auf der Servlet Technologie auf. Es handelt sich dabei um Templates, welche aus statischem Text und auch dynamischen Textelementen (JSP-Elemente) bestehen. Diese JSP Seiten werden dann vom »Web Container« (siehe Abschnitt 2.4.1) in ein Servlet umgewandelt. [12]

Tabelle 2.3: Java EE 6 Komponenten der Web-Schicht

Java EE Komponente	Verwendung
Enterprise Java Beans (EJB)	EJBs dienen dazu die Geschäftslogik einer Anwendung abzubilden. Diese Beans werden in einem eigenen »EJB Container« (siehe Abschnitt 2.4.1) ausgeführt. Es gibt drei Typen von EJBs. Diese sind Session Beans (zustandslos und zustandsbehaftet), Message Driven Beans und Entity Beans. [13]
Java API for RESTful Web Services (JAX-RS)	Ermöglicht die Verwendung der Representational State Transfer (REST) Architektur in Java EE. [14]
Java API for XML Web Services (JAX-WS)	Ermöglicht die Erstellung von Web Services aufbauend auf XML. [15]

Tabelle 2.4: Java EE 6 Komponenten der Business-Schicht

Darstellungsschicht

Die Darstellungsschicht ist die Schnittstelle zum Anwender und ermöglicht diesem die Interaktion mit dem System (Java EE Server). Ein Java EE Client ist, zum Beispiel, ein Webbrowser, welcher auf der Client Maschine ausgeführt wird.

2.4 Infrastruktur von Java EE

Die in Abschnitt 2.3 angeführten Java EE Komponenten benötigen zur Ausführung eine spezielle Infrastruktur, einen sogenannten Java EE Server. Dieser EE Server ist in logische Systeme, welche als Container bezeichnet werden, unterteilt.

2.4.1 Java EE Container

Die Überlegung hinter diesem Komponenten-Container-Prinzip ist die Zerlegung der Verantwortlichkeit einer Serveranwendung in technische und infrastrukturelle Aufgaben auf der einen und der eigentlichen Logik auf der anderen Seite. Die Implementierung dieser Logik erfolgt in den einzelnen Komponenten und wird vom Entwickler durchgeführt, wobei die Infrastruktur von einem Container zur Verfügung gestellt wird. Der Container kann also als Lebensraum der Komponenten gesehen werden. Dieses Prinzip ist dasselbe wie jenes der JVM. Hier wirkt die JVM als Container, um ein Java Programm auf einer Maschine lauffähig zu machen.

Der sogenannte Deployment-Prozess installiert die Java EE Komponenten in den Java EE Containern. Die verschiedenen Java EE Container und deren Komponenten werden in Abbildung 2.2 dargestellt.

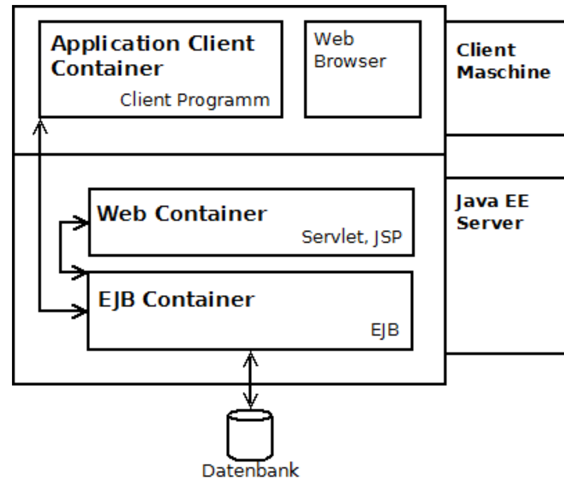

Abbildung 2.2: Java EE Server und Container
[6], S. 48, eigene Überarbeitung

Web Container

Der »Web Container« ist die Schnittstelle zwischen den Webkomponenten und dem Webserver. Eine Webkomponente ist eine Servlet, ein Java Server Faces oder eine Java Server Page. Der Container verwaltet den Lebenszyklus der Komponenten und leitet Anfragen an die Komponenten des »Application Client Container« weiter.

EJB Container

Der »EJB Container« läuft so wie der »Web Container« am Java EE Server. Er ist die Schnittstelle zwischen den »Enterprise Java Beans«, welche die Geschäftslogik abbilden und dem Java EE Server.

Application Client Container

Der »Application Client Container« wird auf der Client Maschine ausgeführt und bildet die Schnittstelle zwischen dem Client Programm und den Java EE Server Komponenten. Ein »Application Client« Container ermöglicht also einer alleinstehenden Java Anwendung die Kommunikation mit dem Java EE Server.

3 Persistenzumgebung der Java Enterprise Edition

Das Java Persistence API (JPA) ist eine Teilspezifikation von Java EE und dient als Schnittstelle, um die Zuordnung von Objekten in einer Datenbank zu vereinfachen. JPA 2.0 ist die aktuelle Version und der Standard für Java EE 6. Es erfolgt in den nachfolgenden Abschnitten keine vollständige Beschreibung des Standards, da dies den Rahmen der Arbeit sprengen würde. Es werden lediglich die wichtigsten Begriffe und Funktionalitäten beschrieben, um die in dieser Arbeit behandeltet Themen zu verstehen. Genauere Informationen können in der Spezifikation von JPA [16] nachgelesen werden.

"*The Java Persistence API provides a POJO persistence model for object-relational mapping. The Java Persistence API was developed by the EJB 3.0 software expert group as part of JSR 220, but its use is not limited to EJB software components. It can also be used directly by web applications and application clients, and even outside the Java EE platform, for example, in Java SE applications.*"[17]

Mit Hilfe von JPA ist es möglich, Java Objekte zur Laufzeit persistent in einer relationalen Datenbank zu speichern, obwohl diese relationalen Datenbanken nicht für objektorientierte Strukturen vorgesehen sind. Dies wird über objektrelationale Abbildungen ermöglicht. Siehe dazu Abschnitt 3.1. Bei den persistenten Java Objekten handelt es sich um ganz normale Java Objekte, den sogenannte POJOs (Plain Old Java Objects).

JPA ist aber nicht auf den Einsatz in einer Java EE Umgebung begrenzt, sondern kann auch außerhalb eines Java EE Containers eingesetzt werden. Dies ist ein wesentlicher Vorteil, um den JPA Standard auch in andere Frameworks bzw. Anwendungen, welche nicht auf Java EE aufbauen, integrieren zu können. Innerhalb eines Java EE Servers wird JPA auch als EJB-Persistenz bezeichnet. [16]

JPA ist aber erst durch Ideen anderer Persistenz-Frameworks wie Hibernate, TopLink oder JDO entstanden. Einer der an der Definition dieses Standards maßgeblich beteiligten Personen war Gavin King, der Gründer von Hibernate [18]. Daher verwendet JPA in etwa den gleichen Ansatz wie Hibernate. Nachfolgend werden diese Ansätze näher beschrieben.

3.1 Objektrelationale Abbildung

Objektrelationale Abbildung (*engl.: Object relational mapping, ORM*) ermöglicht einem objektorientierten Programm, Objekte in einer relationalen Datenbank abbilden zu können. Relationale Datenbanken sind Sammlungen von Tabellen, welche in unterschiedlichsten Beziehungen zueinander stehen. Jede Zeile in einer Tabelle entspricht einem Datensatz, welcher aus verschiedensten Attributen besteht [19]. Objekte eines objektorientierten Programms kapseln diese Daten und ein

entsprechendes Verhalten dazu. Dies bedeutet, dass Objekte im Gegensatz zu einem Datensatz aus einer Tabelle eine Reihe von eigenen Funktionen und Eigenschaften bereitstellen. Bei der Abbildung von Objekten auf die relationalen Datenbanktabellen entstehen daher Widersprüche, welche als objektrelationale Unverträglichkeit (*engl.: Objekt-relational Impedance Mismatch*) bekannt sind. Genauere Informationen darüber können in [20] nachgelesen werden.

Diese objektrelationale Abbildung hat den Vorteil, dass bereits etablierte Techniken von relationalen Datenbanken weiterverwendet werden können und Programmiersprachen nicht erweitert werden müssen. Jedoch können durch diese Abbildung die Fähigkeiten von relationalen Datenbanken nicht komplett ausgenutzt werden, was sich dann auf geringere Leistung und Geschwindigkeit niederschlägt.

Abbildung 3.1 zeigt die Eingliederung einer objektrelationalen Abbildung zwischen den Tabellen der Datenbank und den Objekten.

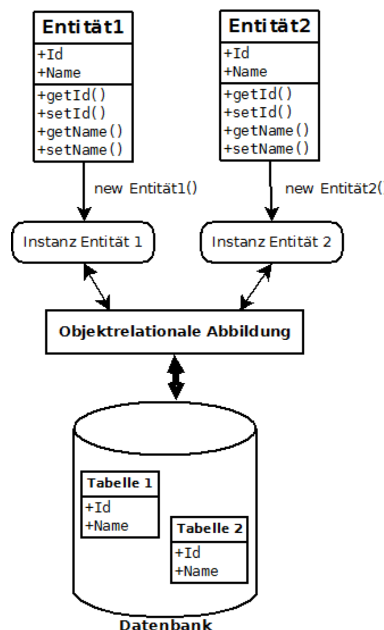

Abbildung 3.1: Funktionsweise einer objektrelationale Abbildung

3.2 Entitäten

Eine Entität (*engl.: Entity*) ist ein leichtgewichtiges, persistentes Objekt und der zentrale Teil der JPA. Normalerweise repräsentiert eine Klasse einer Entität eine Tabelle in einer relationalen Datenbank. Welche Funktionen bzw. persistente Attribute der Entität-Klasse Teil der Persistenz sind, wird über eigens definierte Eigenschaften festgelegt. Diese Eigenschaften werden in Form von Annotationen angegeben, um die Entitäten bzw. die Beziehungen zwischen den Entitäten in das relationale Datenabbild zu übertragen.

Nachfolgende Bedingungen müssen von Entität-Klassen eingehalten werden [16]:

- Die Klasse muss mit der Annotation javax.persistence.Entity versehen werden.
- Die Klasse muss einen parameterlosen *public* oder *protected* Konstruktor enthalten. Weitere Konstruktoren sind erlaubt.
- Die Klasse muss eine Top-Level-Klasse sein und darf nicht mit *final* deklariert werden. Auch keine als persistent markierte Attribute bzw. Funktionen dürfen als *final* deklariert sein.
- Die Klasse muss, wenn Instanzen der Klasse als Wertparameter (*engl.: by value*) übertragen werden, das *serializeable* Interface implementieren.
- Vererbungen sind von einer Entität-Klasse als auch von einer nicht Entität-Klasse möglich. Nicht Entität-Klassen können ebenfalls von Entität-Klassen erben.
- Die Persistenz Attribute der Klasse müssen als *private* oder *protected* deklariert werden und können dadurch nur direkt von der Entität-Klasse selbst angesprochen werden. Für den Zugriff auf die Persistenz Attribute werden öffentliche (*engl.: public*) Getter und Setter Methoden verwendet.
- Die Klasse muss einen Primärschlüssel enthalten.

Listing 3.1 zeigt ein Beispiel einer Entität *Kunde* mit den Attributen *Id* und *name*. Die Annotation *@Entity* markiert die Klasse *Kunde* als Entität. *@Id* markiert den Primärschlüssel und *@GeneratedValue* generiert automatisch eine ID.

```
@Entity
public class Kunde implements Serializable {

    private int id;
    private String name;

    @Id
    @GeneratedValue
    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Listing 3.1: Beispiel einer Entität

3.2.1 Primärschlüssel einer Entität

Jede Entität benötigt einen eindeutigen Schlüssel. Bei einer Kunden Entität ist dieser zum Beispiel die Kundennummer. Dieser eindeutige Schlüssel bzw. Primärschlüssel ermöglicht es, eine

bestimmte Entität zu finden. Der Schlüssel kann entweder ein einzelner oder ein kombinierter, bestehend aus mehreren Attributen, sein.

Ein einzelner Primärschlüssel wird mit der Annotation javax.persistence.Id in der Entität-Klasse gekennzeichnet. Kombinierte Primärschlüssel müssen in einer eigenen Primärschlüsselklasse definiert werden. Diese Primärschlüsselklasse muss folgende Bedingungen erfüllen [16]:

- Die Klasse muss *public* sein.
- Die Klasse muss einen *public* default Konstruktor haben.
- Es müssen die Methoden *equals* und *hashcode* vorhanden sein.
- Die Klasse muss das *serializeable* Interface implementieren.

Es gibt zwei Möglichkeiten eine Primärschlüsselklasse zu integrieren: Eingebettete kombinierte Primärschlüsselklassen, oder nicht eingebettete Primärschlüsselklassen.

Eingebettete kombinierte Primärschlüsselklassen

Sogenannte eingebettete kombinierte Primärschlüsselklassen werden mit der Annotation *javax.persistence.Embedded* markiert (siehe Listing 3.2). Diese eingebetteten Primärschlüsselklassen werden in der Entität-Klasse mit der Annotation *javax.persistence.EmbeddedId* gekennzeichent (siehe Listing 3.3). Solche eingebetteten Klassen gehören direkt zur Entität, was bedeutet, dass die persistenten Attribute auf die gleiche Tabellenzeile wie die der sie umschließenden Entität abgebildet werden. [21]

```
@Embeddable
public class KundePK implements Serializable {
    private String name;
    private long id;

    public KundePK() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }

    public int hashCode() {
        return (int) name.hashCode() + id;
    }

    public boolean equals(Object obj) {
        if (obj == this) return true;
```

```
        if (!(obj instanceof KundePK)) return false;
        if (obj == null) return false;
        KundePK pk = (KundePK) obj;
        return pk.id == id && pk.name.equals(name);
    }
}
```

Listing 3.2: Beispiel für eine eingebettete Primärschlüsselklasse

```
@Entity
public class Kunde implements Serializable {
    KundePK kundeKey;

    public Kunde() {
    }

    @EmbeddedId
    public KundePK getKundeKey() {
        return kundeKey;
    }

    public void setKundeKey(KundePK pk) {
        kundeKey = pk;
    }

    ...
}
```

Listing 3.3: Entität-Klasse mit einer eingebetteten Primärschlüsselklasse

Eingebettete Klassen können in einer Entität auch enthalten sein, wenn es sich nicht um einen Primärschlüsses handelt.

Nicht eingebettete Primärschlüsselklassen

Listing 3.4 zeigt eine nicht eingebettete Primärschlüsselklasse. Bei dieser entfällt die Annotation *javax.persistence.Embedded*. Die Attribute der Primärschlüsselklasse und der Entität-Klasse müssen jedoch hinsichtlich Name und Typ zusammenpassen und die Entität-Klasse muss mit der Annotation *javax.persistence.IdClass* versehen werden (siehe Listing 3.5). [21]

```
public class KundePK implements Serializable {
    private String name;
    private long id;

    public KundePK() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public long getId() {
        return id;
    }

    public void setId(long id) {
        this.id = id;
    }
```

```
    public int hashCode() {
        return (int) name.hashCode() + id;
    }

    public boolean equals(Object obj) {
        if (obj == this) return true;
        if (!(obj instanceof KundePK)) return false;
        if (obj == null) return false;
        KundePK pk = (KundePK) obj;
        return pk.id == id && pk.name.equals(name);
    }
}
```

Listing 3.4: Entität-Klasse mit einer nicht eingebettete Primärschlüsselklasse

```
@IdClass(EmployeePK.class)
@Entity
public class Kunde implements Serializable {
    @Id
    private String name;
    @Id
    private long id;

    ...
}
```

Listing 3.5: Beispiel für eine nicht eingebettete Primärschlüsselklasse

3.2.2 Lebenszyklus einer Entität

Jede Zeile einer Datenbanktabelle ist wenn diese richtig abgebildet wird, ein Objekt einer Entität-Klasse. Entitäten sind also Objekte, welche eindeutig identifiziert sind und bestimmte Informationen und Funktionalitäten bereitstellen. Bei der JPA haben diese Entitäten einen bestimmten Lebenszyklus bzw. verschiedene Zustände.

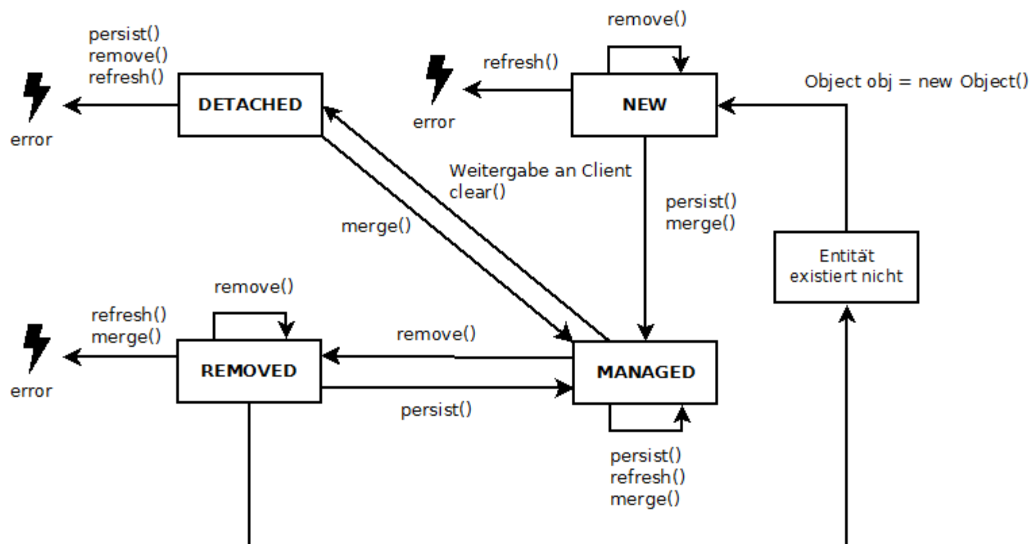

Abbildung 3.2: Lebenszyklus einer Entität
[22], eigene Überarbeitung

Abbildung 3.2 zeigt den Lebenszyklus einer Entität. Eine noch nicht existierende Entität wird mittels *new* erzeugt. In diesem Zustand **NEW** ist dem Objekt noch kein eindeutiger Schlüssel zugewiesen und es hat dadurch auch noch keinen Eintrag in der Datenbank. Über einen Entitäten-Manager (*engl. Entity-Manager*) wird die Entität in der Datenbank über die Methode *persist* des Entitäten-Managers gespeichert und erlangt dadurch den Zustand **MANAGED**. Entitäten werden in diesem Zustand vom Entitäten-Manager verwaltet (siehe Abschnitt 3.3). Entitäten welche von der Datenbank über den Entitäten-Manager gelesen werden, befinden sich ebenfalls im Zustand **MANAGED**. Über die Methode *remove* des Entitäten-Managers wird die Entität in den Zustand **REMOVED** versetzt und nach einem *commit* physikalisch von der Datenbank gelöscht. Im Zustand **DETACHED** befindet sich eine Entität, wenn diese vom Entitäten-Manager abgekapselt wird. Dieser Zustand tritt zum Beispiel ein, wenn die Entität auf ein anders Gerät bzw. eine andere Schicht übertragen wird oder der Entitäten-Manager geschlossen wird. Über den Befehlt *merge* des Entitäten-Managers kann eine Entität vom Zustand **DETACHED** wieder in den Zustand **MANAGED** zurückversetzt werden.

3.2.3 Beziehungen zwischen Entitäten

Die JPA unterstützt Beziehungen zwischen den Entitäten. Nachfolgend werden diese Beziehungstypen und die damit verbundenen Eigenschaften näher betrachtet [16].

Typen von Beziehungen

- **Eins-zu-eins Beziehung** (*engl.: One-to-one Relationsship*): Bei diesem Beziehungstyp ist eine Instanz einer Entität genau einer Instanz einer anderen Entität zugewiesen. Ein Beispiel dafür ist: Eine Entität *KFZ Kennzeichen* gehört genau zu einer Entität *Fahrzeug* und auch umgekehrt. Eins-zu-eins Beziehungen werden mit der Annotation *javax.persistence.OneToOne* angegeben.

- **Eins-zu-viele Beziehung** (*engl.: One-to-many Relationsship*): Eine Instanz einer Entität kann vielen Instanzen anderer Entitäten zugewiesen werden. Ein Beispiel dafür wäre eine *Rechnung-Artikel*-Beziehung: Eine Entität *Rechnung* besitzt eine Liste mit Entitäten von *Artikel*. Eins-zu-viele Beziehungen werden mit der Annotation *javax.persistence.OneToMany* angegeben.

- **Viele-zu-eins Beziehung** (*engl.: Many-to-one Relationsship*): Viele Instanzen einer Entität können einer Instanz einer anderen Entität zugewiesen sein. Ein Beispiel dafür ist eine *Rechnung-Kunde*-Beziehung. Ein Kunde hat viele *Rechnungen*, jedoch benötigt diese Entität keine Liste von allen *Rechnungen*. Die Entität *Rechnung*, welche Teil der Viele-zu-eins Beziehung ist, hat jedoch genau einen *Kunden* und muss diesen daher referenzieren. Viele-zu-eins Beziehungen werden mit der Annotation *javax.persistence.ManyToOne* angegeben.

- **Viele-zu-viele Beziehung** (*engl.: Many-to-many Relationsship*): Viele Instanzen einer Entität können gegenseitig vielen Instanzen einer anderen Entität zugewiesen sein. Ein Beispiel dafür wäre eine *Lehrveranstaltung-Studenten*-Beziehung. Jede *Lehrveranstaltung* hören viele *Studenten* und jeder *Student* besucht mehrere *Lehrveranstaltungen*. Viele-zu-viele Beziehungen werden mit der Annotation *javax.persistence.ManyToMany* angegeben.

- **Vererbung** (*engl.: Inheritance*): Entitäten können Eigenschaften von anderen Entitäten erben. Es handelt sich dabei um eine ganz normale Vererbung von Java Objekten. Ein Beispiel

dafür wäre eine Entität *Auto* und eine Entität *Motorrad*, welche die Eigenschaften der Entität Fahrzeug erben.

- **Eingebettete Klassen** (*engl.: Embedded Class*): Um Klassen einer Entität schmal zu halten bietet JPA die Möglichkeit von eingebetteten Klassen. Ein Beispiel dafür wäre eine eingebettete Klasse *Adresse* in der Entität *Kunde*. Bei der Klasse *Adresse* handelt es sich um keine Entität. Diese wird lediglich mit der Annotation *javax.persistence.Embeddable* gekennzeichnet. In der Entität *Kunde* erfolgt die Kennzeichnung der eingebetteten Klasse *Adresse* mit der Annotation *javax.persistence.Embedded*. Ein Beispiel dazu findet sich in Listing 5.5.

Richtungen in Beziehungen

Die Richtung in einer Beziehung kann entweder Bidirektional oder Unidirektional sein [16]:

- **Bidirektionale Beziehung**: Bei einer bidirektionalen Beziehung besitzt jede Entität ein Feld mit einer Beziehung zur anderen Entität. Über das Beziehungsfeld kann von der einen Entität auf die referenzierende Entität geschlossen werden. Wenn zum Beispiel die Entität *Rechnung* die zugewiesenen *Artikel* und *Artikel* die Beziehung zu den referenzierten *Rechnungen* kennt, so spricht man von einer bidirektionalen Beziehung.

- **Unidirektionale Beziehung**: Bei einer unidirektionalen Beziehung hat nur eine Entität ein Beziehungsfeld zu der referenzierenden Entität. Die Entität *Rechnung* besitzt eine Liste mit allen *Artikeln*, die Entität *Artikel* hat jedoch keine Referenz auf die Entität *Rechnung*, so spricht man von einer unidirektionalen Beziehung.

Kaskadierende Datenbankoperationen

Unter kaskadieren versteht man das Weiterreichen der Datenbankoperation an das verknüpfende Objekt. Entitäten die in Beziehungen zueinander stehen, haben oft Abhängigkeiten, welche die Existenz der anderen Entitäten betreffen. Zum Beispiel: Eine *Ordner* Entität enthält mehrere *Datei* Entitäten. Wird nun die *Ordner* Entität gelöscht, so sollen auch die enthaltenen *Datei* Entitäten gelöscht werden. Dies wird als eine kaskadierende Datenbankoperation bezeichnet. Eine kaskadierende Datenbankoperation gibt an, was mit verknüpften Objekten passieren soll, wenn ein Objekt durch eine Datenbankoperation manipuliert wird. In der Tabelle 3.1 werden die *javax.persistence.CascadeType* Operationen welche durch JPA spezifiziert sind angeführt.

Kaskadierende Operation	Beschreibung
ALL	Diese Operation bedeutet, dass alle Operationen DETACH, MERGE, PERSIST, REFRESH und REMOVE an die in Beziehung zu dieser Entität stehenden Entitäten weitergeleitet werden.
DETACH	Wird die Entität in den Status *DETACHED* überführt, so werden auch alle verknüpften Entitäten in den Status *DETACHED* versetzt.
MERGE	Wird die Entität in den Status *MERGED* überführt, so werden auch alle verknüpften Entitäten in den Status *MERGED* versetzt.
PERSIST	Wird die Entität in den Persistenzkontext gespeichert, so werden auch alle verknüpften Entitäten gespeichert.
REFRESH	Werden die Daten der Entität aktualisiert, so werden auch die Daten der verknüpften Entität aktualisiert.
REMOVE	Wird die Entität vom Persistenzkontext entfernt, so werden auch alle verknüpften Entitäten entfernt.

Tabelle 3.1: Kaskadierende Operationen für Entitäten
[6], S. 571

3.3 Verwaltung der Entitäten

Die beteiligten Komponenten zur Verwaltung der Entitäten sind in Abbildung 3.3 dargestellt.

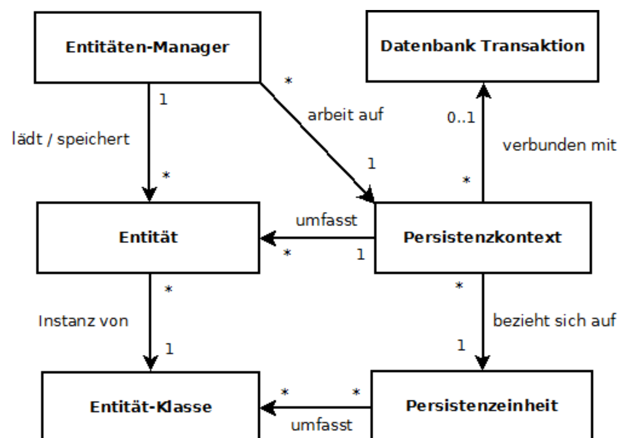

Abbildung 3.3: Komponenten zur Verwaltung der Entitäten
[22], eigene Überarbeitung

Ein Entitäten-Manager arbeitet genau mit einem Persistenzkontext und lädt und speichert Entitäten. Der Persistenzkontext umfasst mehrere Entitäten und eine Entität ist immer genau einem Persistenzkontext zugeordnet. Der Persistenzkontext bezieht sich genau auf eine Persistenzeinheit und verwaltet den transaktionsbasierten Zugriff auf die Datenbank. Die Persistenzeinheit verweist auf mehrere Entitäten-Klassen, von denen dann die Entitäten instanziert werden. Diese angeführten Komponenten werden in den nachfolgenden Abschnitten näher beschreiben. Die Komponenten Entität und Entität-Klasse wurden in Abschnitt 3.2 behandelt.

3.3.1 Entitäten-Manager

Die Verwaltung der Entitäten übernimmt der sogenannte Entitäten-Manager (engl.: Entity-Manager). Dieser erstellt, verändert, löscht und findet persistente Entitäten und ist für den Lebenszyklus der Entitäten verantwortlich. Der Entitäten-Manager stellt die API für den Zugriff auf den Persistenzkontext bereit. Siehe dazu Abschnitt 3.3.2.

Typen von Entitäten-Managern

Bei den Entitäten-Managern wird zwischen zwei Typen unterschieden [22]:

- **Container-MANAGED**: Dieser Typ kommt innerhalb eines Java EE Containers zum Einsatz. Dabei ist der Container für das Erzeugen und Schließen des Entitäten-Managers verantwortlich und er verwaltet die JTA (Java Transaction API) Transaktionen. Den Zugriff auf den Entitäten-Manager erhält die Anwendung über Dependency Injection oder per JNDI über die Annotation *javax.persistence.PersistenceContext*.

- **Application MANAGED**: Bei diesem Typ von Entitäten-Manager wird dieser von der Anwendung selbst verwaltet. Die Erzeugung und Zerstörung des Entitäten-Managers und die Verknüpfung mit dem Persistenzkontext werden direkt von der Anwendung übernommen. Für die Erzeugung wird die *EntityManagerFactory* und für Transaktionen kann JTA verwendet werden.

Grundfunktionalitäten des Entitäten-Managers

Der Entitäten-Manager besitzt eine Reihe von Funktionalitäten um die Entitäten zu verwalten. Wie wir bereits wissen, befinden sich Entitäten in einem dieser vier Zustände: *NEW*, *MANAGED*, *DETACHED* oder *REMOVED*. Diese Zustände werden durch nachfolgende Funktionen erreicht [22].

- ***T find(Class<T> entityClass, Object primaryKey)***: Diese Funktion sucht eine Entität anhand des übergebenen Primärschlüssels. Es wird zuerst im Persistenzkontext und danach erst in der Datenbank gesucht. Dies bringt den Vorteil mit sich, dass ein wiederholtes Abrufen derselben Entität keine nennenswerte Zeit kostet. Gibt es für den angegebenen Primärschlüssel keinen Datensatz, wird *null* geliefert. Eine IllegalArgumentException wird geworfen, wenn der Primärschlüssel nicht zum Typ der Primärschlüsseldefinition der Entität-Klasse passt.

- ***T getReference(Class<T> entityClass, Object primaryKey)***: Dies Funktion gibt ein nicht initialisiertes Objekt, welches nur den Primärschlüssel eines Datensatzes bereitstellt, zurück. Dies bedeutet, dass die Daten erst beim ersten Zugriff automatisch nachgeladen werden. Diese Funktion ist sehr gut einzusetzen, wenn die Entität große Datenmengen enthält.

- ***void persist(Object entity)***: Diese Funktion dient dazu eine neue Entität zum Persistenzkontext hinzuzufügen und diese in den Zustand *MANAGED* zu versetzen. Nach Beendigung der Transaktion werden die Daten der Entität in die Datenbank übernommen. Wird die *persist* Funktion auf eine bereits als gelöscht gekennzeichnete Entität angewandt, so wird diese wieder in den Status *MANAGED* zurückversetzt. Wenn sich die übergebene Entität im Zustand *DETACHED* befindet, wird eine *IllegalArgumentException* geworfen.

- ***T merge(T entity)***: Mit dieser Funktion können Entitäten, welche sich im Status *DETACHED* befinden, wieder in den Status *MANAGED* überführt werden. In diesem Status wird die Entität vom Entitäten-Manager verwaltet. Nach dem Aufruf sollte mit der als Rückgabe Parameter gesetzten Entität weiter gearbeitet werden. Auf eine neue Entität wirkt diese Funktion genauso wie die Funktion *persist*. Der Aufruf auf eine gelöschte Entität ist jedoch nicht erlaubt.

- ***void remove(Object entity)***: Sich im Status *MANAGED* befindende Entitäten werden über diese Funktion für das Löschen der zugehörigen Daten in der Datenbank vorgemerkt. Wird die *remove* Funktion auf eine Entität im Zustand *REMOVED* oder *New* angewandt, so wird der Aufruf ignoriert. Zu einer *IllegalArgumentException* kommt es, wenn die Funktion auf eine Entität im Zustand *DETACHED* angewandt wird.

- ***void refresh(Object entity)***: Diese Funktion fragt die Daten einer Entität im Zustand *MANAGED* erneut aus der Datenbank ab. Diese Funktion ist sinnvoll, wenn mehrere Anwendungen gleichzeitig mit derselben Datenbank arbeiten. Handelt es sich um keine *MANAGED* Entität, wird eine *IllegalARgumentException* geworfen.

- ***boolean contains(Object entity)***: Der Rückgabewert ist wahr, wenn die übergebene Entität bereits im Persistenzkontext enthalten ist.

- ***void clear()***: Alle im Persistenzkontext enthaltenen Entitäten werden entfernt und befinden sich danach im Zustand *DETACHED*.

3.3.2 Persistenzkontext

Der Persistenzkontext ist nichts anderes als eine Menge von Entitäten, die im Rahmen einer Transaktion gelesen, erzeugt und verändert werden und am Ende der Transaktion mit der Datenbank abgeglichen werden müssen. Jede in diesem Persistenzkontext enthaltene Entität ist eindeutig identifizierbar und genau nur einmal darin enthalten. Der Persistenzkontext kann im Grunde als Cache, welcher die Entitäten enthält und darauf achtet, dass Aktualisierungen oder Abfragen gegenüber der Datenbank nicht unnötigerweise mehrfach ausgeführt werden, angesehen werden. Diese Entitäten werden nun über den Entitäten-Manager in den Persistenzkontext eingefügt oder entfernt. [23]

3.3.3 Persistenzeinheit

Die Persistenzeinheit (*engl.: Persistence-Unit*) gibt an, welche Entität-Klassen vom Entitäten-Manager verwalten werden und in der Datenbank abgebildet werden sollen. Die erforderlichen Meta-Informationen wie die Benennung der Entität-Klassen oder die Verbindung zur Datenbank werden XML-basiert abgebildet in der *persistence.xml* Datei. Ein Beispiele einer Persistenzeinheit ist in Abschnitt 6.3 angeführt. Abbildungsregeln der Entitäten zu den Datenbanktabellen können

in Form von Annotationen in der Entität-Klasse selbst angegeben werden. Der Entitäten-Manager wird aufbauend auf dieser Persistenzeinheit gebildet und ist daher immer genau einer Persistenzeinheit zugeordnet. [23]

3.3.4 Persistenzanbieter

Ein Persistenzanbieter (engl.: Persistence Provider) ist eine Technologie, welche das Java Persistence API implementiert und um Funktionalitäten erweitert. Bei den nachfolgend in Tabelle 3.2 angeführten Persistenzanbietern handelt es sich um bekannte Implementierungen von JPA.

Technologie	Beschreibung
Hibernate	Bei Hibernate handelt es sich um ein quelloffenes Framework für Java, welches im Jahr 2001 entwickelt wurde. Hibernate gab es bereits vor JPA und es war daher maßgeblich an der Definition des Standards von JPA beteiligt. Daher weißt JPA auch viele Ähnlichkeiten mit Hibernate auf. [36]
TopLink	TopLink Essentials war die quelloffene Edition von TopLink und es dient als Referenzimplementierung von JPA 1.0. [24]
EclipseLink	EclipseLink basiert auf TopLink und es ist ebenfalls ein quelloffenes Persistenzframework. Es dient als Referenzimplementierung für JPA 2.0. [37]
OpenJPA	OpenJPA ist eine quelloffene Implementierung von JPA. [38]

Tabelle 3.2: Bekannte Persistenzanbieter

3.3.5 Abfragen der Entitäten

Um Entitäten aus der Datenbank abzufragen, unterstützt JPA zwei Möglichkeiten: Die **Java Persistence Query Language (JPQL)** [16] und das **Criteria API** [16]. Es besteht natürlich auch die Möglichkeit, direkt in SQL die Abfragen durchzuführen, jedoch werden dadurch nicht all die Vorteil der Objektorientierung ausgenutzt. Normale SQL Abfragen können über die Funktion *createNativeQuery()* des Entitäten-Managers abgesetzt werden.

Java Persistence Query Language

Die Java Persistence Query Language (JPQL) erstellt die Abfragen auf Basis des abstrakten Schemas der Entitäten. Dieser Vorteil betrifft vor allem, dass die Abhängigkeiten zwischen ver-

schiedenen Datenbankherstellern und deren unterschiedlichen SQL Dialekten nicht berücksichtigt werden müssen und dass objektorientierte Methoden eingesetzt werden können.

Die Abfragesprache von JPA ist der von SQL sehr ähnlich und daher für SQL erfahrene Benutzer leicht anzuwenden. Durch Abfragen ist es möglich, bestehende Entitäten aufzufinden, zu ändern oder zu löschen. Die einzige Möglichkeit, Entitäten auch ohne eine Abfragesprache zu lesen, stellt die Methode *find* des Entitäten-Managers über den Primärschlüssel dar.

Der Entitäten-Manager ist der Ausgangspunkt für die Abfrage von Entitäten. Als Ergebnis einer Abfrage können sowohl Entitäten als auch einfache Java Objekte geliefert werden. Werden Entitäten gelesen, so werden diese nach der Abfrage gleich mit dem Persistenzkontext verknüpft, damit nachträgliche Änderungen an den Entitäten automatisch mit der Datenbank synchronisiert werden. Die nachfolgenden Methoden des Entitäten-Managers werden verwendet, um Abfrage über die JPQL auf den Datenspeicher anzuwenden.

javax.persistence.Query createQuery(String qlString): Diese Methode erzeugt eine dynamische Abfrage. Der Ausdruck wird in Form eines String-Datentyps übergeben. Das Ergebnis ist ein Objekt vom Typ *javax.persistence.Query*, welches über Funktionen für die eigentliche Ausführung der Abfrage verfügt. Ein Beispiel einer solchen Abfrage ist in Listing 3.6 dargestellt. Die Abfrage liefert alle *Kunde* Entitäten. Über die Funktion *getResultList()* werden diese in Form einer *List* returniert.

```
javax.persistence.Query query = entityManager.createQuery("SELECT k FROM Kunde");
List<Kunde> kunden = query.getResultList();
```

Listing 3.6: Beispiel einer Abfrage

Es werden auch parametrisierte Abfragen, welche in Listing 3.7 zu sehen sind, unterstützt. Dadurch wird ein umständlicher Zusammenbau von Abfragestrings umgangen.

```
String name = "Mustermann";
javax.persistence.Query query = entityManager.createQuery("SELECT k FROM Kunde
                                                          WHERE k.name LIKE :name");
query.setParameter("name", name);
List<Kunde> kunden = query.getResultList();
```

Listing 3.7: Beispiel einer parametrisierten Abfrage

Es werden von JPQL auch Änderungs- und Löschoperationen unterstützt. Listing 3.8 setzt alle Postleitzahlen aller Kunden auf 12345. Änderungs- und Löschoperationen werden über die Funktion *executeUpdate* des Query Objekts durchgeführt. Rückgabewert ist die Anzahl der geänderten bzw. gelöschten Datensätze.

```
public int setztePLZ() {
 javax.persistence.Query query = entityManager.createQuery("UPDATE Kunde k
                                                          SET k.plz = 12345");
 return query.executeUpdate();
}
```

Listing 3.8: Beispiel einer Massenaktualisierung

javax.persistence.Query createNamedQuery(String name): Diese Methode erzeugt eine statische Abfrage. Ein Vorteil bei dieser Methode ist vor allem, dass der Sourcecode leserlicher wirkt. Dabei können Abfragen in den Metadaten von Entitäten formuliert werden. Listing 2.1

zeigt eine solche statische Abfrage.

```
@NamedQuery(name="alleKunden", query="SELECT k FROM Kunde k")
@Entity
public class Kunde implements Serializeable {
...
}

public List<Kunde> alleKunden() {
 javax.persistence.Query query = entityManager.createNamedQuery("alleKunden");
 return query.getResultList();
}
```

Listing 3.9: Beispiel einer benannten Abfrage

JPQL unterstützt natürlich viel mehr Funktionalitäten für Abfragen, als jene die bis jetzt beispielhaft angeführt wurden. Funktionen, welche bereits aus SQL bekannt sind, wie zum Beispiel *GROUP-BY*, *HAVING*, *ORDER-BY* und *JOINS*, werden in dieser Arbeit nicht näher beschrieben und können in [6] nachgelesen werden.

Criteria API

Ähnlich wie JPQL basiert das Criteria API auf dem abstrakten Schema der Entitäten. Das Criteria API ermöglicht jedoch streng typisierte Abfragen. Bei JPQL werden die Abfragen stringbasiert zusammengestellt, was zur Folge hat, dass Fehler erst zur Laufzeit erkannt werden. Durch das Cirteria API können diese Fehler bereits bei der Kompilierung festgestellt werden.

Listing 3.10 zeigt ein Beispiel einer Abfrage über das Criteria API.

```
 CriteriaBuilder cb = entityManager.getCriteriaBuilder();
 CriteriaQuery<Kunde> cq = cb.createQuery(Kunde.class);
 Root<Kunde> kunde = cq.from(Kunde.class);
 cq.select(kunde);
 TypedQuery<Kunde> q = entityManager.createQuery(cq);
 List<Kunde> kunden = q.getResultList();
```

Listing 3.10: Beispiel einer Abfrage über das Criteria API

Der Entitäten-Manager erstellt über die Funktion *getCriteriaBuilder()* ein *CriteriaBuilder* Objekt. Dieser *CriteriaBuilder* ermöglicht es über die Funktion *createQuery(Class class)* ein *CriteriaQuery* Objekt zu erzeugen. Über den Parameter *class* kann die Klasse der abzufragenden Entität angegeben werden. Über die Funktion *from(Class class)* wird ein sogenanntes *Root* Objekt, welches ähnlich der *FROM* Klausel in einer JPQL Abfrage entspricht, generiert. Dieses *Root* Objekt wird über die Methode *select(Root<?> root)* im *CriteriaQuery* Objekt gesetzt.

Das Criteria API ermöglicht es auch, die Resultate einer Abfrage über bestimmte Methoden einzugrenzen bzw. Abfragen zu gruppieren oder zu sortieren. Um Abfragen einzugrenzen stellt das CriteriaBuilder Interface die in Tabelle 3.3 enthaltenen Methoden zur Verfügung.

Methode	Beschreibung
equal	Überprüft, ob zwei Ausdrücke gleich sind
notEqual	Überprüft, ob zwei Ausdrücke nicht gleich sind
gt	Überprüft, ob der erste größer als der zweite Ausdruck ist
ge	Überprüft, ob der erste größer oder gleich dem zweiten Ausdruck ist
lt	Überprüft, ob der erste kleiner als der zweite Ausdruck ist
ge	Überprüft, ob der erste kleiner oder gleich dem zweiten Ausdruck ist
between	Überprüft, ob der erste zwischen dem zweiten und dritten Ausdruck liegt
like	Überprüft, ob der Ausdruck einem bestimmten Muster entspricht

Tabelle 3.3: Eingrenzung einer Abfrage
[6], S. 648

Mehrere Abfragebedingungen können über die in Tabelle 3.4 angeführten Methoden des *CriterialBuilder* verknüpft werden.

Methode	Beschreibung
and	Entspricht einer logischen Konjunktion von zwei Boolschen Ausdrücken
or	Entspricht einer logischen Disjunktion von zwei Boolschen Ausdrücken
not	Entspricht einer logischen Negation von zwei Booleschen Ausdrücken

Tabelle 3.4: Logische Verknüpfung von Abfragen
[6], S. 649

Listing 3.11 zeigt ein Beispiel für eine eingegrenzte Abfrage und der Verknüpfung mittels eines logischen Operators. Es werden alle Kunden, welche eine Id größer 12 und kleiner 20 haben, geliefert.

```
CriteriaBuilder cb = entityManager.getCriteriaBuilder();
CriteriaQuery<Kunde> cq = cb.createQuery(Kunde.class);
Root<Kunde> kunde = cq.from(Kunde.class);
cq.where(cb.gt(kunde.get("id"), 12)
        .and(cb.lt(kunde.get("id"), 20)));
cq.select(kunde);
TypedQuery<Kunde> q = entityManager.createQuery(cq);
List<Kunde> kunden = q.getResultList();
```

Listing 3.11: Beispiel einer Abfrage mittels Criteria API und logischer Verknüpfung

Genauere Details und Beispiele zum Criteria API können in [6] nachgelesen werden.

4 Das quelloffene Framework JVx

JVx ist ein quelloffenes Framework für Java Anwendungen, welches von der Firma SIB Visions GmbH [39] in Wien herausgegeben und verwaltet wird. JVx wurde entwickelt, um den Entwicklern von Datenbankanwendungen mehr Zeit für die Lösung von anwendungsspezifischen Anforderungen zu geben. Diese Eigenschaft wird dadurch erreicht, dass häufig wiederkehrende Aufgaben wie die Erstellung und Verwaltung von Stammdaten und dazugehöriger Arbeitsoberflächen weniger Zeit beanspruchen. Auf der Webseite von SIB Visions ist ein Vergleich von JVx zu anderen Java Frameworks, welcher anhand eines individuellen Softwareprojekts mittlerer Größe durchgeführt wurde, zu sehen [40]. Eine weitere sehr spezielle Eigenschaft von JVx ist die technologieunabhängige Darstellungsschicht. Für die Darstellung am Client existieren in JVx Implementierungen, für die Technologien Swing, QT Jambi [41] und GWT [42]. JVx bietet auch die Möglichkeit, über Erweiterungspakete mobile Android Anwendungen oder Silverlight [43] Anwendungen zu erstellen.

4.1 Infrastruktur und Architektur von JVx

JVx schafft keinen neuen Standard, sondern es ist ein Framework welches bekannte Techniken mit neuen Umsetzungsideen verbindet. JVx verfolgt wie Java EE eine Mehrschichtenarchitektur. Siehe dazu Abbildung 4.1.

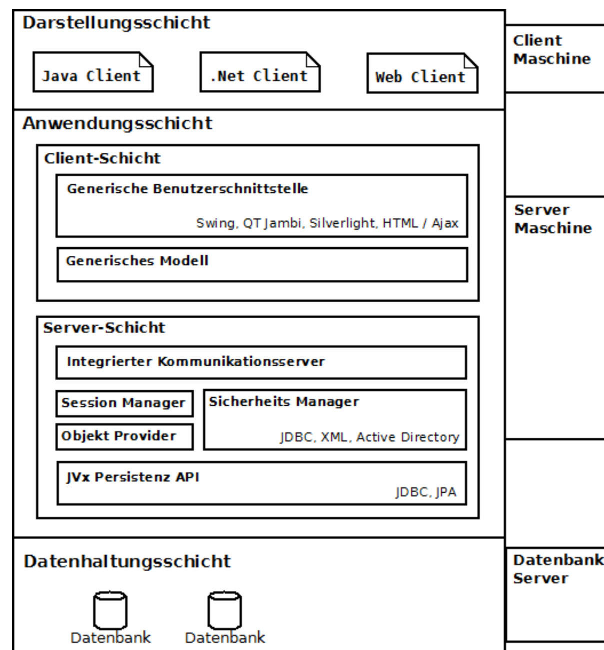

Abbildung 4.1: Systemarchitektur von JVx
[44], eigene Überarbeitung

Die Architektur von JVx gliedert sich in eine Darstellungsschicht (Client Maschine), eine Anwendungsschicht (Server Maschine) und eine Datenhaltungsschicht (Datenbank Server). Aber auch, wenn die gesamte JVx Anwendung als Desktop-Anwendung gestartet wird und nur lokal auf einem Rechner in einer virtuellen Maschine ausgeführt wird, ist es von Vorteil, diese Anwendung in mehrere Schichten mit klar getrennten Aufgabenbereichen zu unterteilen.

4.1.1 Darstellungsschicht

Die Darstellungsschicht ist für die Darstellung von Anwendermasken auf der Client Maschine verantwortlich und sie ermöglicht dem Benutzer die Interaktion mit dem Server. Eine JVx Anwendung kann aufgrund der technologieunabhängigen Client-Schicht als normale Desktop-Anwendung oder als Webanwendung ohne Codeänderung in einem Browser ausgeführt werden.

4.1.2 Anwendungsschicht

Die Anwendungsschicht betrifft die Entwicklung einer JVx Anwendung und sie gliedert sich in eine Client-Schicht und eine Server-Schicht. Details zur Entwicklung einer JVx Anwendung können aus Abschnitt 4.3 entnommen werden.

Client-Schicht

Die Client-Schicht wird als Darstellungsschicht des Servers bezeichnet und sie ist, wie bereits erwähnt, nicht auf eine bestimmte Technologie zur Darstellung von Anwendermasken beschränkt. Diese Eigenschaft ist ein besonderes und nützliches Merkmal des Frameworks und sie ermöglicht es ohne Codeänderungen eine geschriebene Anwendung als Java Desktop Anwendung oder als Web Anwendung zu verwenden. Ermöglicht wird dies durch eine generische Benutzerschnittstelle. Diese generische Benutzerschnittstelle ist im Quellcode [45] von JVx im Paket ***javax.rad.genui*** zu finden. Dieses Paket enthält konkrete Klassen von Elementen einer Benutzerschnittstelle (z.B.: *javax.rad.genui.component.UILabel*, *javax.rad.genui.component.UITextField*, usw.), welche als Hülle für die gewünschte Technologie fungieren. Die Anwendermasken in JVx werden nur mit diesen sogenannten »GUI Controls« erstellt. Listing 4.1 zeigt ein einfaches Beispiel für zwei dieser generischen Komponenten.

```
UILabel lblFirstName = new UILabel();
UITextField tfFirstName = new UITextField();

lblFirstName.setText("Vorname");
tfFirstName.setText("Stefan");
```

Listing 4.1: Beispiel einer einfachen Anwendermaske unter JVx

Bei der Erzeugung dieser »GUI Controls« wird je nach Start der Anwendung die passende technologieabhängige Komponente erzeugt. Diese befindet sich für Swing im Paket ***com.sibvisions.ui.swing*** oder für QT Jambi im Paket ***com.sibvisions.ui.qt***.

Alle diese »GUI Controls« verwenden im Hintergrund ein generisches Modell, um Daten darzustellen und zu manipulieren. Dieses generische Modell ist durch die Schnittstelle **javax.rad.model.IDataBook** definiert. Die Klasse

com.sibvisions.rad.model.remote.RemoteDataBook ist eine Implementierung der Schnittstelle *javax.rad.model.IDataBook*. Weitere Klassen dazu finden sich im Paket ***javax.rad.model*** und ***com.sibvisions.rad.model***. Diese Schicht hat nur über die Server-Schicht Zugriff auf die Daten der Datenhaltungsschicht.

Server-Schicht

Die Client-Schicht kommuniziert mit der Server-Schicht über eine Remoteverbindung. Ein Kommmunikationsserver nimmt die Anfragen des Clients entgegen, er verarbeitet diese und übermittelt das aufbereitete Ergebnis an den Client. Diese protokollunabhängige Kommunikation zwischen Client-Schicht und Server-Schicht ermöglicht also den Zugriff auf Daten der Datenhaltungsschicht. Dieser Zugriff wird über das Paket ***javax.rad.remote*** und ***com.sibvisions.rad.remote*** abgewickelt.

Die Verwaltung der Kommunikation, der Sessions (Master- und Subsessions) und der Serverobjekte wurde in den Paketen ***javax.rad.server*** und ***com.sibvisions.rad.server*** implementiert. Der Sicherheitsmanager ist für die Authentifizierung von berechtigten Anwendern verantwortlich. Die Authentifizierungsinformationen können aus verschiedenen Datenquellen, wie zum Beispiel einer Datenbank oder XML Datei, gelesen werden.

Der Zugriff auf die Daten unterschiedlichster Datenquellen wird in JVx über ein eigenes Persistence API geregelt. Die benötigten Interfaces und Klassen dafür befinden sich im Paket ***javax.rad.persist*** und ***com.sibvisions.rad.persist***. Siehe dazu Abschnitt 4.2.

4.1.3 Datenhaltungsschicht

Bei der Datenhaltungsschicht handelt es sich um einen beliebigen Datenspeicher. Dazu gehören sowohl relationale Datenbanksysteme wie Oracle oder MySql als auch bestimmte Datenformate wie zum Beispiel XML.

4.2 Persistenzumgebung von JVx

Das Persistenz API von JVx wird durch die Schnittstelle ***javax.rad.persist.IStorage*** definiert. Sogenannte »Storage Objekte« implementieren diese *javax.rad.persist.IStorage* Schnittstelle um CRUD (Create, Read, Update und Delete) Operationen auf eine bestimmte Datenquelle zur Verfügung zu stellen. Ein »Storage Objekt« repräsentiert also zum Beispiel eine Tabelle in der Datenbank. Auf die implementierten Schnittstellen der »Storage Objekte« wird über Remote Aufrufe von der Klasse *com.sibvisions.rad.model.remote.RemoteDataBook* zugegriffen. Die *com.sibvisions.rad.model.remote.RemoteDataBook* Klasse ist eine Implementierung der Schnittstelle *javax.rad.model.IDataBook* und sie entspricht dadurch einem generischen Modell, welches von den »GUI Controls« verwendet wird, um auf Daten zuzugreifen.

Der Austausch der Datensätze einer Datenquelle über das Persistenz API von JVx erfolgt über Felder von Objekten (*Object [ ]*). Ein *Object [ ]* entspricht genau einem Datensatz aus der Datenquelle. Der Aufbau und die Anordnung der einzelnen Werte eines Datensatzes entsprechen immer genau dem Aufbau der Metadaten (*javax.rad.persist.MetaData*) eines »Storage Objektes«.

4.2.1 Aufbau der Metadaten

Die Metadaten von JVx repräsentieren die Struktur der Datenquelle und dadurch die Eigenschaften der einzelnen Wertespalten dieser Datenquelle. Objekte der Klasse *javax.rad.persist.MetaData* werden am Client verwendet, um auf einzelne Werte eines Datensatzes zugreifen zu können und um dessen Eigenschaften zu berücksichtigen.

javax.rad.persist.MetaData enthält eine Liste aller *javax.rad.persist.ColumnMetaData*, welche die Spalten eines »Storage Objektes« mit deren Eigenschaften widerspiegeln. Abbildung 4.2 zeigt den Aufbau von *javax.rad.persist.MetaData* und die Beziehung zu der Klasse *javax.rad.persist.ColumnMetaData*, welche die Eigenschaften (Name, Label, Typ, usw.) der einzelnen Spalten kapselt.

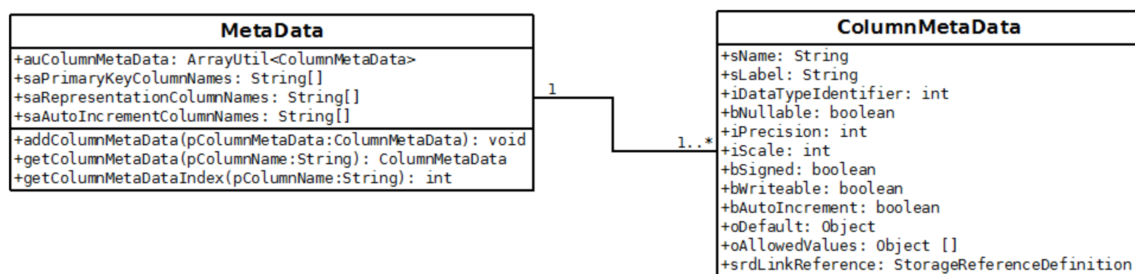

Abbildung 4.2: Klassendiagramm zu den Metadaten von JVx

In der Klasse *MetaData* der Abbildung 4.2 wurden nicht alle, sondern nur die für das Verständnis wichtigen Funktionen abgebildet. Jedem Objekt der Klasse *javax.rad.persist.ColumnMetaData* ist ein bestimmter Datentyp zugewiesen. JVx bietet für die Darstellung der Daten sechs verschieden Datentypen an, welche sich im Paket *javax.rad.model.datatype* befinden:

- **BigDecimalDataType**: Für die nummerische Darstellung von Werten.
- **BinaryDataType**: Für die Kapselung von binären Daten.
- **BooleanDataType**: Wahr und Falsch Werte
- **ObjectDataType**: Entspricht irgendeinem Java Objekt.
- **StringDataType**: Für die textuelle Darstellung von Werten
- **TimestampDataType**: Für die Kapselung eines Datums

Über diese Eigenschaften der Klasse *javax.rad.persist.ColumnMetaData* ist es direkt am Client möglich, Benutzereingaben zu beschränken. Ein Benutzer kann zum Beispiel für die Eingabe einer Zahl, welche auf vier Zeichen beschränkt ist, nur diese vier Zeichen eingeben. Handelt es sich um ein Datumsfeld (TimestampDataType), so kann dem Benutzer automatisch ein Kalender zur Eingabe des Datums angezeigt werden.

Die *StorageReferenceDefinition* in der Klasse *ColumnMetaData* wird für die automatische Verlinkung von Datenquellen verwendet. Eine genauere Erklärung dieser automatischen Verlinkung ist in Abschnitt 4.3 zu finden.

4.2.2 Funktionen von IStorage

Nachfolgende Funktionen werden durch die *javax.rad.persist.IStorage* Schnittstelle definiert.

- ***MetaData getMetaData()***: Liefert die Metadaten. Das Objekt *javax.rad.persist.MetaData* enthält eine Liste aller *javax.rad.persist.ColumnMetaData*, welche die Spalten eines »Storage Objektes« mit deren Eigenschaften (Typ, Länge, usw.) widerspiegeln.
- ***Object[] insert(Object[] pDataRow)***: Fügt den übergebenen Datensatz zur Datenquelle hinzu und liefert diesen eingefügten Datensatz retour.
- ***Object[] update(Object[] pOldDataRow, Object[] pNewDataRow)***: Ändert den alten Datensatz mit den Daten des neuen Datensatzes in der Datenquelle und liefert diesen retour.
- ***void delete(Object[] pDeleteDataRow)***: Löscht den übergebenen Datensatz aus der Datenquelle.
- ***int getEstimatedRowCount(ICondition pFilter)***: Liefert die Anzahl der Datensätze für das übergeben Einschränkungskriterium.
- ***List<Object[]> fetch(ICondition pFilter, SortDefinition pSort, int pFromRow, int pMinimumRowCount))***: Liefert die angeforderten Datensätze von der Datenquelle. Ein Eintrag der Liste entspricht genau einem Datensatz aus der Datenquelle. *javax.rad.model.condition.ICondition* definiert die Einschränkungskriterien. *javax.rad.model.SortDefinition* gibt die Sortierung der Ergebnisse an. Siehe dazu Abschnitt 4.2.3. Der Parameter *pFromRow* gibt an, ab welcher Datenzeile die Ergebnisse geliefert werden sollen und der Parameter *pMinimumRowCount* gibt an, wie viele Datensätze mindestens geliefert werden müssen, falls diese vorhanden sind.
- ***Object[] refetchRow(Object[] pDataRow)***: Liefert für den übergebenen Datensatz die aktuellen Werte.

4.2.3 Einschränkungskriterien und Sortierung von Abfragen

Das Persistenz API von JVx bietet die Möglichkeit Abfragen von Daten aus einer Datenquelle einzuschränken und zu sortieren. Die möglichen Einschränkungskriterien sind im Paket *javax.rad.model.condition* definiert und in Abbildung 4.3 abgebildet.

Das Interface *ICondition* repräsentiert die Schnittstelle für alle Bedingungen und ermöglicht die Gruppierung zu logischen UND- und ODER-Operationen. Die abstrakte Klasse *BaseCondition* ist die Defaultimplementierung von *ICondition*. Diese ermöglicht, die Operationen *And* und *Or* mit anderen Bedingungen zu verwenden. *OperatorCondition* ist eine abstrakte Klasse, welche die Defaultimplementierung für die logischen Operationen *And* und *Or* ist. Die Subklasse *Or* von *OperatorCondition* implementiert den logischen Oder-Operator (Disjunktion) und die Subklasse *And* den logischen Und-Operator (Konjunktion). Die Klasse *Not* ist die Implementierung des logischen Nicht-Operator (Negation). Die abstrakte Klasse *CompareCondition* bildet die Basisklasse für alle Vergleichs-Operationen. Siehe dazu Tabelle 4.1.

Listing 4.2 zeigt Beispiele zur Verwendung der Einschränkungskriterien. Diese kombinierbaren Bedingungen können dem Persistenz API in der *fetch* Methode übergeben werden, um die

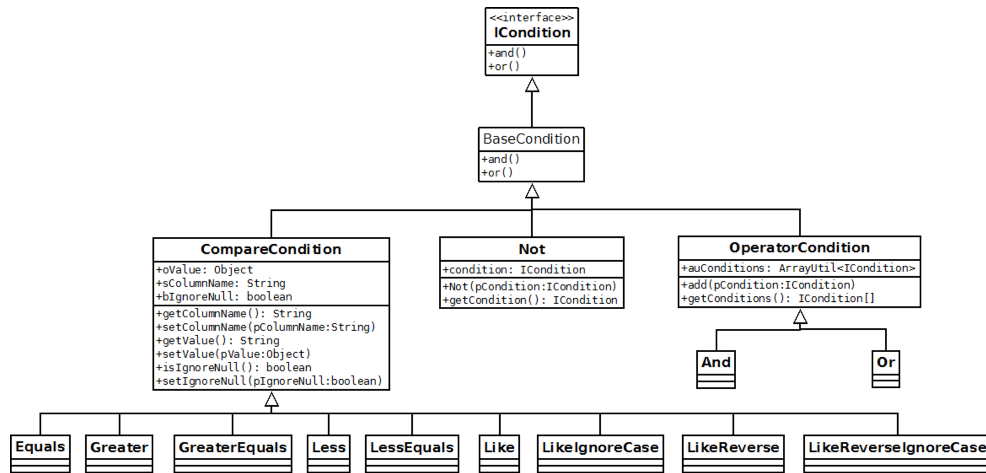

Abbildung 4.3: Klassendiagramm zu den Einschränkungskriterien von JVx

Klasse	Beschreibung
Equals	Ermöglicht den Vergleich von Werten auf deren Gleichheit.
Greater	Ist ein Wert größer als ein anderer Wert.
GreaterEquals	Ist ein Wert größer oder gleich als ein anderer Wert.
Less	Ist ein Wert kleiner als ein anderer Wert.
LessEquals	Ist ein Wert kleiner oder gleich als ein anderer Wert.
Like	Ermöglicht den Vergleich von Werten mit Stellvertretersymbolen (Wildcards). ? steht für irgendein Zeichen. * steht für eine unlimitierte Anzahl an Zeichen. Beispiel: *[SPALTENNAME] like 'M*'*
LikeIgnoreCase	Gleich wie *Like*, jedoch wird die Groß- und Kleinschreibung nicht berücksichtigt.
LikeReverse	Die umgekehrte Möglichkeit zu *Like*. Es wird jene Spalte angegeben, in der sich das Vergleichsmuster befindet. Beispiel: *'Max Mustermann' like [SPALTENNAME]*
LikeReverseIgnoreCase	Gleich wie *LikeReverse*, jedoch wird die Groß- und Kleinschreibung nicht berücksichtigt.

Tabelle 4.1: Subklassen der Klasse CompareCondition

bestimmten Daten aus der Datenquelle zu filtern.

```
ICondition c = new LikeIgnoreCase("ADR", "*STRASSE*").and(new Equals("PLZ", "1100"));

ICondition c = new Greater("EINKOMMEN", 3000).and(new LessEquals("EINKOMMEN", 4000));

ICondition c = new Equals("NAME", "Stefan").or(new Equals("NAME", "Markus"));
```

Listing 4.2: Beispiel zu Abfrage mittels ICondition

Um eine Abfrage zu sortieren gibt es die Klasse *javax.rad.model.SortDefinition*. Diese Klasse spezifiziert die zu verwendende Sortierreihenfolge. Listing 4.3 zeigt ein Beispiel. Der erste Parameter des Konstruktors gibt die zu sortierende Spalte an, der zweiter Parameter gibt an, ob absteigend (false) oder aufsteigend (true) sortiert werden soll.

```
SortDefinition s = new SortDefinition(new String[] {"ID"}, new boolean [] { true });
```

Listing 4.3: Beispiel für eine SortDefinition

4.3 Funktionalität von JVx anhand eines Beispiels

JVx ermöglicht, wie bereits des öfteren erwähnt, die Entwicklung von Datenbankanwendungen in kurzer Zeit und mit wenig Quellcode. Um diese Eigenschaft besser verstehen zu können, werden nachfolgend einige Funktionsweisen, Funktionalitäten und Besonderheiten von JVx anhand eines kleinen Beispiels erläutert. Das Beispiel dient vor allem dazu, den Aufbau und einige damit verbundene Besonderheiten einer JVx Anwendung kennenzulernen. Weitere Funktionen und Möglichkeiten von JVx können unter [40] nachgelesen werden.

Die Entwicklung einer JVx Anwendung gliedert sich grundsätzlich in die Entwicklung eines Clients und in die eines Servers. Der Client enthält jene Klassen, Komponenten und Funktionen zur Darstellung der Benutzeroberfläche und Entgegennahme der Benutzerinteraktionen. Der Server enthält jene Klassen und Funktionen, welche die Geschäftslogik, die Verbindung zur Datenquelle, die Benutzerverwaltung und die Verwaltung der Sessions abbilden.

Nachfolgend werden exemplarisch Client- und Serverfunktionalität einer JVx Anwendung beschrieben. Die genaue Verzeichnisstruktur, benötigte Entwicklungsumgebungen, Konfigurationsdateien und Pakete zur Entwicklung einer JVx Anwendung können unter [46] nachgelesen werden.

4.3.1 Aufbau und Funktionalität der Beispielanwendung

Nachfolgende Abbildung 4.4 stellt die Datenstruktur der Beispielanwendung dar. Dieses Klassendiagramm zeigt, dass einem *Kunden* eine *Anrede* (z.b.: 'Herr' oder 'Frau') zugewiesen ist. Jeder *Kunde* kann mehrere *Adressen* besitzen, wobei jede *Adresse* genau zu einem *Kunden* gehört. Zu jedem *Kunden* kann es mehrere *Ausbildungen* geben und jede *Ausbildung* kann mehreren *Kunden* zugewiesen sein.

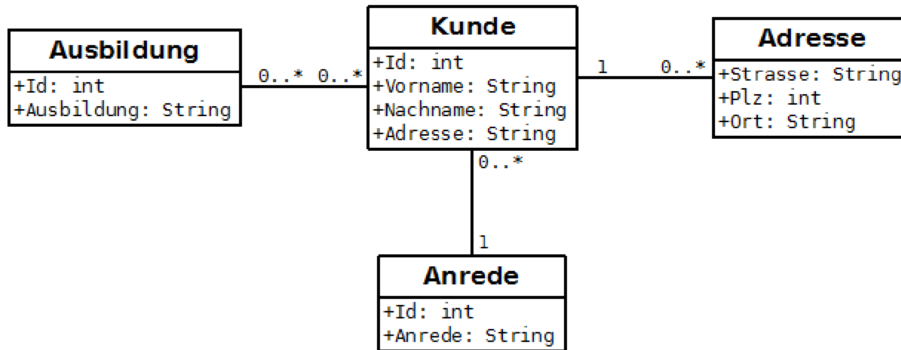

Abbildung 4.4: Datenstruktur der Beispielanwendung

Die Beispielanwendung ermöglicht dem Benutzer, Kunden inklusive deren Adressen anzuzeigen und zu verwalten. Zu jedem Kunden werden die zugehörigen Ausbildungen angezeigt. Die Abbildung 4.5 zeigt die Arbeitsoberfläche für die Kundenverwaltung.

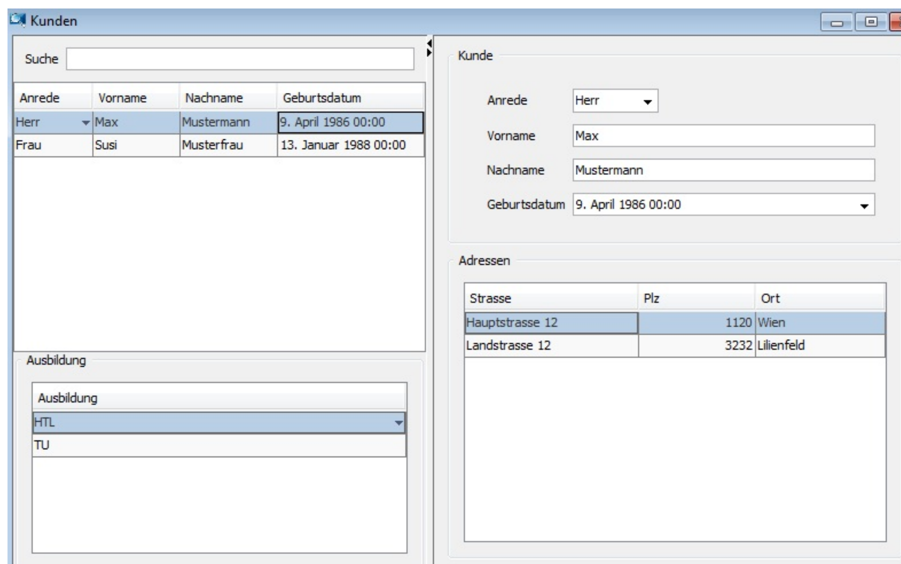

Abbildung 4.5: Arbeitsoberfläche der Beispielanwendung

4.3.2 Entwicklung der Client-Schicht

Der Client benötigt eine Klasse vom Typ *javax.rad.application.IApplication*. Diese Schnittstelle ist eine technologieunabhängige Definition einer Anwendung. JVx bietet dafür bereits eine Standardimplementierung über die Klasse *com.sibvisions.rad.application.Application* an. Diese Klasse stellt einige Komponenten, wie zum Beispiel eine Anmeldemaske und ein Menü inklusive einer Symbolleiste, zur Verfügung. Wird der Client von dieser Klasse abgeleitet, so können diese Funktionen direkt übernommen und durch zusätzliche Funktionen erweitert werden.

```
public class JVxBeispielAnwendung extends Application {

    public static final String VERSION = "1.2";

    public JVxBeispielAnwendung(UILauncher pLauncher) {
        super(pLauncher);
    }

    @Override
    protected IConnection createConnection() throws Exception {
        return new DirectServerConnection();
    }

    @Override
    protected String getApplicationName() {
        return "beispiel";
    }

    @Override
    protected void afterLogin() {
        super.afterLogin();

        UIMenu menuMasterData = new UIMenu();
        menuMasterData.setText("Daten");

        UIMenuItem miKontakte = createMenuItem("doOpenKunde",
                   null,
                   "Kunden",
                   UIImage.getImage("/images/contact.png"));

        menuMasterData.add(miKontakte);

        //vor dem Menuepunkt Hilfe einfügen
        getMenuBar().add(menuMasterData, 1);

        UIToolBar tbMasterData = new UIToolBar();

        UIButton butKunde = createToolBarButton
                          ("doOpenKunde", null, "Kunden",
                           UIImage.getImage("/images/contact.png"));

        tbMasterData.add(butKunde);

        getLauncher().addToolBar(tbMasterData);
    }

    public void doOpenKunde() throws Throwable
    {
        KundeFrame frame = new KundeFrame(this);

        configureFrame(frame);

        frame.setVisible(true);
    }

}
```

Listing 4.4: Beispielcode eines JVx Clients

Listing 4.4 zeigt als Beispiel eine Client Anwendung. Der Konstruktor dieser Klasse bekommt einen technologieabhängigen *Launcher* übergeben. Dieser *Launcher* kann beim Start der Anwendung angegeben werden. Soll der Client zum Beispiel als Java Swing Client gestartet werden, so muss als *Launcher* die Klasse *com.sibvisions.rad.ui.swing.impl.SwingApplication* übergeben

werden.

Über die Funktion *createConnection* wird das Kommunikationsprotokoll initialisiert. Wird zum Beispiel ein Anwendungsserver bei der Kommunikation zwischen Client und Server eingesetzt, so kann auch eine *HttpConnection* verwendet werden. In unserem kleinen Beispiel reicht eine *DirectServerConnection*, da Client und Server in derselben virtuellen Maschine laufen.

Die Funktion *afterLogin* wird aufgerufen, nachdem eine erfolgreiche Anmeldung durchgeführt wurde. In dieser Funktion wird der Client um einen Menüpunkt (siehe Funktion *createMenuItem*) und einen Eintrag (siehe Funktion *createToolBarButton*) in der Symbolleiste erweitert. Durch einen Klick auf den Menüeintrag bzw. Eintrag in der Symbolleiste wird die Funktion *doOpenKunde* aufgerufen, welche unsere Arbeitsoberfläche für die Kundenverwaltung (siehe Abbildung 4.5) öffnet.

Listing 4.5 zeigt einen Ausschnitt der Klasse *KundeFrame*. Diese bekommt im Konstruktor die Clientanwendung (siehe Listing 4.4) übergeben und initialisiert über die Funktionen *initializeModel()* (siehe Listing 4.7 und 4.8) die generischen Modelle und *initializeUI* (siehe Listing 4.10) die »GUI Controls« für diese Arbeitsoberfläche. Für jede angezeigte Datenquelle wird ein eigenes generisches Modell (*rdbKunde*, *rdbAdresse* und *rdbKundeAusbildung*) erzeugt.

```
public class KundeFrame extends UIInternalFrame {

    private Application application;

    private AbstractConnection connection;

    private RemoteDataSource dataSource = new RemoteDataSource();

    // Modell für die Datenquelle Kunde
    private RemoteDataBook rdbKunde = new RemoteDataBook();

    // Modell für die Datenquelle Adresse
    private RemoteDataBook rdbAdresse = new RemoteDataBook();

    // Modell für die Datenquelle KundeAusbildung
    private RemoteDataBook rdbKundeAusbildung = new RemoteDataBook();

    ...

    public KundeFrame(JVxBeispielAnwendung pApp) throws Throwable {
        super(pApplication.getDesktopPane());

        application = pApp;

        initializeModel();
        initializeUI();
    }

    ...
}
```

Listing 4.5: Beispielcode zur Kundenverwaltung

Listing 4.6 zeigt die Funktion *initializeModel()*, welche die Objekte für den Zugriff auf den Server bzw. die Daten instanziert. In dieser Funktion wird eine eigene Verbindung zum Server für diese Arbeitsoberfläche erstellt. Es wird von der *MasterConnection*, welche von der Clientan-

wendung *JVxBeispielAnwendung* verwaltet wird, eine *SubConnection* zum Server erzeugt. Dies bewirkt, dass für diese Arbeitsoberfläche am Server ein eigenes »Lifecycle Objekt« verwendet wird. Siehe dazu Listing 4.13. Das »Lifecycle Objekt« ist in diesem Beispiel über die Klasse *apps.beispiel.frames.DBEdit* spezifiziert. Dieses »Lifecycle Objekt« hält alle Objekte, welche von der Arbeitsoberfläche benötigt werden. Wird die Arbeitsoberfläche der Kundenverwaltung geschlossen, wird auch die *SubConnection* zum Server beendet und der benutzte Speicher wieder freigegeben. Die erzeugte *SubConnection* zum Server wird im *DataSource* Objekt gespeichert, welche sich um die Übertragung der Daten zwischen Client und Server kümmert.

```
private void initializeModel() throws Throwable {

    connection = ((MasterConnection)application.getConnection()).
                 createSubConnection("apps.beispiel.frames.DBEdit");
    connection.open();

    //data connection
    dataSource.setConnection(connection);
    dataSource.open();
    ...
}
```

Listing 4.6: Erzeugen einer Verbindung zum Server

Nach erfolgreicher Erzeugung der Verbindung zum Server werden die generischen Modelle zur Verwaltung der Daten initialisiert. Wie aus Abbildung 4.4 ersichtlich ist, besteht unsere Beispielanwendung aus den Tabellen *Anrede*, *Kunde*, *Adresse* und *Ausbildung*. Aufgrund der Viele-zu-viele Beziehung zwischen *Ausbildung* und *Kunde* existiert als Datenspeicher eine zwischen Tabelle *KundeAusbildung*, welche in dieser Abbildung nicht ersichtlich ist.

Listing 4.7 zeigt die Initialisierung des generischen Modells für die Kunden Tabelle. Es wird der *DataSource*, welcher die Verbindung zum Server hält, gesetzt und der Name jener Funktion am Server, welche den Zugriff auf die Datenquelle für die Kunden bereitstellt. Bei dieser Funktion handelt es sich um das »Lifecycle Objekt« der Klasse *apps.beispiel.frames.DBEdit* für die Arbeitsoberfläche der Kundenverwaltung. Siehe dazu Listing 4.13.

```
private void initializeModel() throws Throwable {
    ...
    rdbKunde.setDataSource(dataSource);
    rdbKunde.setName("kunde");
    rdbKunde.open();
    ...
}
```

Listing 4.7: Initialisiere RemoteDataBook für Kunde

Automatische Verlinkung

Eine weitere Besonderheit von JVx ist die automatische Verlinkung jener Datenquellen, welche eine Viele-zu-eins Beziehung zu einer Datenquelle aufweisen. In unserem Beispiel ist das die Beziehung zwischen der Datenquelle *Anrede* und *Kunde*. Es genügt die Definition des generischen Modells für *Kunde*. Die Verlinkung zu *Anrede* wird automatisch von JVx zur Verfügung gestellt

und als automatische Auswahlliste in den entsprechenden »GUI Controls« dargestellt. Siehe dazu Abbildung 4.6 und 4.7.

Abbildung 4.6: Automatische Auswahllisten in Tabellen

Abbildung 4.7: Automatische Auswahllisten in der Detailansicht

Listing 4.8 zeigt die Initialisierung des generischen Modells für die Adresse Tabelle. Es wird der *DataSource* gesetzt welcher die Verbindung zum Server hält und der Name jener Funktion am Server, welche den Zugriff auf die Datenquelle für die Adresse bereitstellt. Siehe dazu Listing 4.13.

```
private void initializeModel() throws Throwable {
    ...
    rdbAdresse.setDataSource(dataSource);
    rdbAdresse.setName("adresse");

    ReferenceDefinition rdAdresse = new ReferenceDefinition();
    rdAdresse.setColumnNames(new String[] {"KUNDE_ID"});
    rdAdresse.setReferencedDataBook(rdbKunde);
    rdAdresse.setReferencedColumnNames(new String[] {"ID"});

    rdbAdresse.setMasterReference(rdAdresse);
    rdbAdresse.open();
    ...
}
```

Listing 4.8: Initialisiere RemoteDataBook für Adresse

Master-Detail-Beziehung

Eine weitere Besonderheit von JVx ist die Master-Detail-Beziehung zwischen Datenquellen. Diese ermöglicht auf einfache Art und Weise, eine Verknüpfung zwischen zwei Datenquellen herzustellen. In unserem Beispiel ist eine Master-Detail-Beziehung die Beziehung zwischen *Kunde* und *Adresse*, aber auch zwischen *Kunde* und *KundeAusbildung*. Es werden also mittels dieser Master-Detail-Beziehung die Eins-zu-viele und Viele-zu-viele Beziehungen aufgelöst. In Listing 4.8 wird die Master-Detail-Beziehung zwischen der Datenquelle *Kunde* und *Adresse* über den

Fremdschlüssel *KUNDE_ID* und dem Primärschlüssel *ID* der Datenquelle *Kunde* hergestellt. Sobald die Auswahl einer Zeile aus dem generischen Modell des Masters registriert wird, werden die passenden Details aus der Datenquelle geladen und angezeigt. In unserem Beispiel erfolgt durch die Auswahl des Masters *Kunde* automatisch die Afnzeige der zugehörigen *Adressen*. Wird jetzt eine neue Adresse zu einem Kunden angelegt, wird durch die Master-Detail-Beziehung die korrekte Zuweisung zum richtigen Kunden gewährleistet.

Die Initialisierung des generischen Modells für die Datenquelle KundeAusbildung ist ähnlich wie Listing 4.8 und wird daher nicht weiter erläutert.

Unser Beispiel enthält auch eine Filterfunktion. Diese Einschränkung der Daten kann Clientseitig als auch Serverseitig durchgeführt werden. Die Clientseitige Filter der Daten wird im Hauptspeicher durchgeführt. Damit das möglich ist, müssen jedoch alle Daten am Client zur Verfügung stehen. Sollten das aufgrund des Lazy-Loading-Mechanismus noch nicht der Fall sein, werden die Daten automatisch ausgelesen. Der Vorteil liegt in der Reduzierung der Kommunikation. Standardmäßig ist jedoch die Datenbankseitige Filterung der Daten eingestellt. Dabei wird das erzeugte Einschränkungskriterium (siehe Abschnitt 4.2.3) an das Persistenz API von JVx gesendet, um die gefilterten Daten aus der Datenquelle zu lesen. Dadurch ist sichergestellt, dass immer auf die Echtdaten zugegriffen wird. In Listing 4.9 wird ein Suchfeld erzeugt und es wird dafür eine Client-Aktion registriert. Eine Client-Aktion definiert eine Funktion am Client, welche zu einem vordefinierten Zeitpunkt aufgerufen wird. Die Funktionen mit denen Aktionen erzeugt werden können, beginnen immer mit *event* (z.B.: eventValuesChanged, eventAction, usw.). In unserem Beispiel wird bei einer Änderung der Werte im Suchfeld die Funktion *doFilter* am Client aufgerufen.

```
private void initializeModel() throws Throwable {
  ...
  RowDefinition definition = new RowDefinition();
  definition.addColumnDefinition(new ColumnDefinition("SEARCH", new StringDataType()));

  drSearch = new DataRow(definition);
  drSearch.eventValuesChanged().addListener(this, "doFilter");
  ...
}

public void doFilter() throws ModelException
{
    String suche = (String)drSearch.getValue("SEARCH");
    if (suche == null)
    {
        rdbKunde.setFilter(null);
    }
    else
    {
        ICondition filter = new LikeIgnoreCase("VORNAME", "*" + suche + "*").or(
                            new LikeIgnoreCase("NACHNAME", "*" + suche + "*").or(
                            new LikeIgnoreCase("ADRESSE", "*" + suche + "*")));

        rdbKunde.setFilter(filter);
    }
}
```

Listing 4.9: UITable und UIEditor mit Modell verknüpfen

Listing 4.10 zeigt die Funktion *initializeUI()*, welche die Arbeitsoberfläche zur Kundenverwaltung erzeugt und die Verbindung der einzelnen »GUI Controls« zu den passenden generischen

Modellen herstellt. Das generische Modell von JVx erlaubt die Anzeige und Bearbeitung von Daten auf unterschiedlichste Weise. Egal, ob die Daten in einer Tabelle oder einem Text Editor bearbeitet werden, sie werden in allen Fällen vom Modell verwaltet. Eine Änderung der Daten in diesem Modell wirkt sich automatisch auf alle registrierten Komponenten aus. Dadurch sind die angezeigten Daten immer synchron.

In Listing 4.10 werden nicht alle benötigten Komponenten für dieses Beispiel dargestellt. Wichtig ist nur, dass die Verknüpfung der Editoren mit dem zugehörigen Modell erkannt wird. In den »GUI Controls« UITable genügt es, wenn das zugehörige Modell gesetzt wird. In den »GUI Controls« UIEditor muss zusätzlich zum zugehörigen Modell noch die passende Spaltenbezeichnung gesetzt werden.

```
private void initializeUI() throws Throwable
{
    UILabel lblAnrede = new UILabel();
    UILabel lblVorname = new UILabel();

    ...

    UIEditor edtAnrede = new UIEditor();
    UIEditor edtVorname = new UIEditor();

    ...

    UITable tabKunde = new UITable();
    tabKunde.setDataBook(rdbKunde);

    UITable tabKundeAusbildung = new UITable();
    tabKundeAusbildung.setDataBook(rdbKundeAusbildung);
    tabKundeAusbildung.setPreferredSize(new UIDimension(150, 150));

    UITable tabAdresse = new UITable();
    tabAdresse.setDataBook(rdbAdresse);
    tabAdresse.setPreferredSize(new UIDimension(150, 150));


    lblAnrede.setText("Anrede");
    lblVorname.setText("Vorname");
    lblNachname.setText("Nachname");
    lblGeburtsdatum.setText("Geburtsdatum");

    edtAnrede.setDataRow(rdbKunde);
    edtAnrede.setColumnName("ANREDE_ANREDE");
    edtVorname.setDataRow(rdbKunde);
    edtVorname.setColumnName("VORNAME");
    edtNachname.setDataRow(rdbKunde);
    edtNachname.setColumnName("NACHNAME");
    edtGeburtsdatum.setDataRow(rdbKunde);
    edtGeburtsdatum.setColumnName("GEBURTSDATUM");

    ...
    setSize(new UIDimension(600, 500));
}
```

Listing 4.10: UITable und UIEditor mit Modell verknüpfen

Ob es sich bei »GUI Controls« dann um Text, Nummern oder Datums Editoren handelt, hängt vom Datentyp der gesetzten Spalte ab. Es erfolgt eine automatisch Überprüfung anhand der definierten Metadaten (siehe Abschnitt 4.2.1) des zugehörigen Modells. Siehe dazu Abbildung 4.8.

Abbildung 4.8: UIEditor für Datumseingabe

4.3.3 Entwicklung der Server-Schicht

Die Serverklassen bilden die bereits erwähnten »Lifecycle Objekt«, welche die Geschäftslogik der Anwendung enthalten, ab. Grundsätzlich folgt eine JVx Anwendung dem in Abbildung 4.9 dargestellten Aufbau. Die in blau dargestellten Klassen müssen vom Entwickler einer JVx Anwendung selbst erzeugt werden.

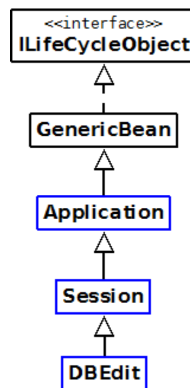

Abbildung 4.9: Serverklassen einer JVx Anwendung

Welches Objekt ein »Lifecycle Objekt« einer Anwendung und welches teil der Session ist wird in einer Konfigurationsdatei (config.xml) einer JVx Anwendung definiert. Siehe dazu Listing 4.11.

```
<?xml version="1.0" encoding="UTF-8"?>

<application>
  ...
  <lifecycle>
    <mastersession>apps.beispiel.Session</mastersession>
    <application>apps.beispiel.Application</application>
  </lifecycle>
  ...
</application>
```

Listing 4.11: Auszug aus der Konfugrationsdatei einer JVx Anwendung

Lifecycle Objekte einer Anwendung

Die Klasse *Application* repräsentiert das »Lifecycle Objekt« für eine Anwendung. Es existiert genau eine Instanz dieser Klasse und diese ermöglicht dadurch, auf Objekte unterschiedlicher Sessions zuzugreifen. Da in unserem Beispiel keine Session-übergreifenden Objekte notwendig sind, benötigt diese Klasse keine zusätzlich Funktionalität.

Lifecycle Objekte einer Session

Die Klasse *Session* repräsentiert das »Lifecycle Objekt« für eine Session. In unserem Beispiel beginnt eine Session mit der Anmeldung an der Anwendung und endet mit der Abmeldung. Es existiert pro Session genau eine Instanz dieses Objekts, welches die Verwendung von Objekten für die Dauer der Anwendung ermöglicht. Dieses Objekt kann zum Beispiel zum Aufbau einer Datenbankverbindung verwendet werden. Dadurch ist sichergestellt, dass pro Session genau eine Datenbankverbindung aufgebaut wird. Siehe dazu Listing 4.12. In diesem Beispiel wird über die Funktion *getDBAccess()* eine Verbindung zur HSQLDB Datenbank *jvxbeispiel* aufgebaut. Durch die Ableitung von *Application* ist es möglich auch die Objekte der Anwendung zu verwenden.

```
public class Session extends Application {

    public DBAccess getDBAccess() throws Exception
    {
        DBAccess dba = (DBAccess)get("dBAccess");

        if (dba == null)
        {
            dba = DBAccess.getDBAccess("jdbc:hsqldb:hsql://localhost/jvxbeispiel");

            dba.setUsername("SA");
            dba.setPassword("");
            dba.open();

            put("dBAccess", dba);
        }

        return dba;
    }

}
```

Listing 4.12: Beispiel eines Datenbankaufbaus unter JVx

Lifecycle Objekte einer Arbeitsoberfläche

Die Klasse *DBEdit* repräsentiert das »Lifecycle Objekt« für die Arbeitsoberfläche der Kundenverwaltung. Auf diese Objekte kann ausschließlich über die *SubConnection* der aktuellen Arbeitsoberfläche zugegriffen werden. Siehe dazu Listing 4.6. Durch die Ableitung von der Klasse *Session* können Objekte der übergeordneten Klasse *Session* bzw. der *Application* verwendet werden. Siehe dazu Listing 4.13.

```
public class DBEdit extends Session {

    public IStorage getKunde() throws Exception
```

```
    {
        DBStorage dbKunde = (DBStorage)get("kunde");

        if (dbKunde == null)
        {
            dbKunde = new DBStorage();
            dbKunde.setDBAccess(getDBAccess());
            dbKunde.setFromClause("KUNDE");
            dbKunde.setWritebackTable("KUNDE");
            dbKunde.open();

            put("kunde", dbKunde);
        }

        return dbKunde;
    }

    public IStorage getAdresse() throws Exception
    {
        DBStorage dbAdresse = (DBStorage)get("adresse");

        if (dbBestellung == null)
        {
            dbAdresse = new DBStorage();
            dbAdresse.setDBAccess(getDBAccess());
            dbAdresse.setFromClause("ADRESSE");
            dbAdresse.setWritebackTable("ADRESSE");
            dbAdresse.open();

            put("adresse", dbAdresse);
        }

        return dbAdresse;
    }

    public IStorage getKundeAusbildung() throws Exception
    {
        DBStorage dbKundeAusbildung = (DBStorage)get("kundeAusbildung");

        if (dbKundeAusbildung == null)
        {
            dbKundeAusbildung = new DBStorage();
            dbKundeAusbildung.setDBAccess(getDBAccess());
            dbKundeAusbildung.setFromClause("KUNDE_AUSBILDUNG");
            dbKundeAusbildung.setWritebackTable("KUNDE_AUSBILDUNG");
            dbKundeAusbildung.open();

            put("kundeAusbildung", dbKundeAusbildung);
        }

        return dbKundeAusbildung;
    }

}
```

Listing 4.13: Klasse zum >Lifecycle Objekt< der Arbeitsoberfläche

Die Funktionen *getKunde()*, *getAdresse()* und *getKundeAusbildung()* ermöglichen den Zugriff auf die Datenquelle. In unserem Fall ist die Datenquelle eine Datenbank mit den Tabellen *KUNDE*, *ADRESSE* und *KUNDE_AUSBILDUNG*. Die Namen der Funktionen müssen den Namen, welche bei der Initialisierung der generischen Modell gesetzt wurden, entsprechen. Siehe dazu Listing 4.7 und Listing 4.8. Der Rückgabewert dieser Funktionen ist ein Objekt vom Typ *javax.rad.persist.IStorage*. Siehe dazu Abschnitt 4.2.

5 Generische Integration von JPA in JVx

Generisch stammt von dem lateinischen Wort »genus« ab und bedeutet so viel wie Art, Gattung oder Stamm [47] und stellt somit einen Oberbegriff für eine große Gruppe bzw. Klasse von Objekten dar. In der Softwareprogrammierung ist eine mögliche Definition des Wortes »generisch« (engl. generic) folgende:

"*Generics is a form of abstraction in developing code. Instead of writing a function or a class for a particular type, it can be written generally to use any type. When an instance of the generic class is instantiated, the type is specified.*"[48]

In diesem Kapitel wird beschrieben, wie die generische Integration von JPA in JVx ermöglicht wird. Es erfolgt eine Gegenüberstellung der beiden Technologien Java EE und JVx, die Erklärung der Integration, sowie eine Auflistung der zugrundeliegenden Qualitätskriterien. Der Quellcode für die Integration von JPA in JVx, sowie eine Beispielanwendung, kann unter [59] heruntergeladen werden.

5.1 Gegenüberstellung von JVx und Java EE

Die beiden Architekturmodelle von Java EE und JVx wurden bereits in den Abschnitten 2.3 und 4.1 erläutert. Abbildung 5.1 stellt diese beiden Architekturmodelle gegenüber. Sie unterscheiden sich vor allem in der Anwendungsschicht und im logischen Aufbau einer Anwendung. Die Client-Schicht einer JVx Anwendung kann mit der Web-Schicht einer Java EE Anwendung und die Server-Schicht einer JVx Anwendung mit der Business-Schicht einer Java EE Anwendung verglichen werden.

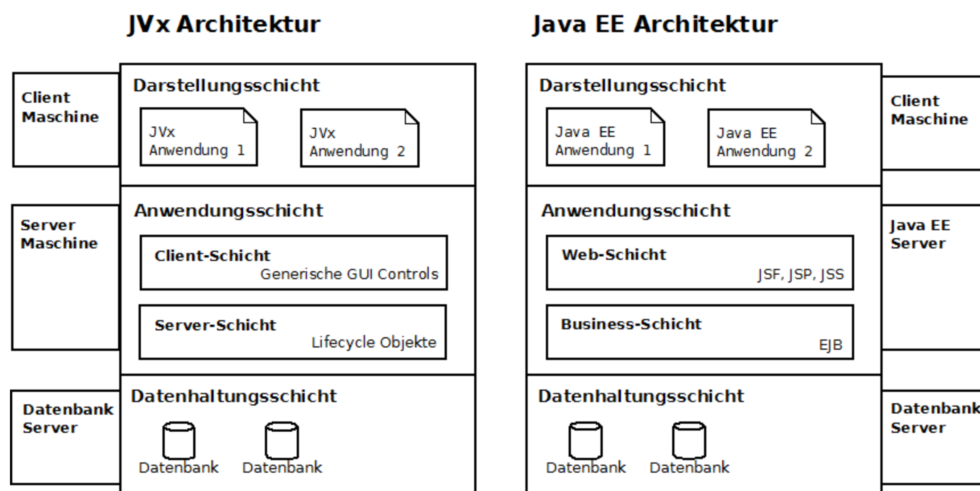

Abbildung 5.1: Gegenüberstellung der JVx und Java EE Architektur
[44], [6] S. 41, eigene Überarbeitung

Abbildung 5.2 stellt beispielhaft einen logischen Aufbau einer Java EE Anwendung und einer JVx Anwendung gegenüber. Ein ähnlicher Aufbau einer Java EE Anwendung mit den unterstützenden Frameworks Sruts, Spring und JPA ist in [25] ersichtlich. Dieser Aufbau wird in den nachfolgenden Abschnitten 5.1.1 und 5.1.2 näher beschrieben.

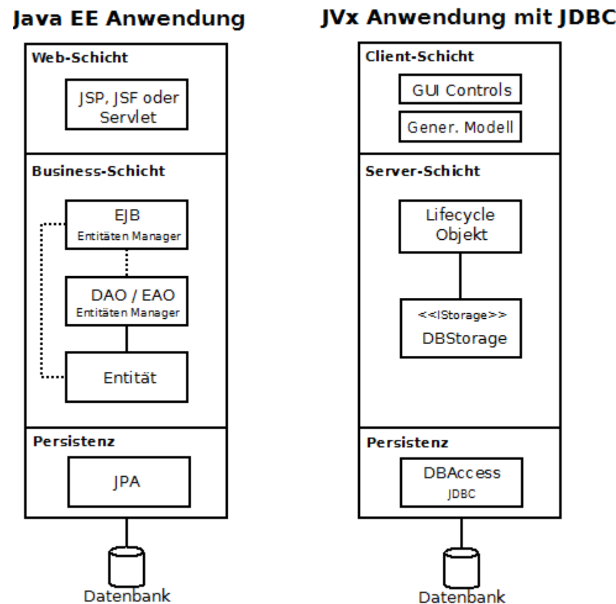

Abbildung 5.2: Gegenüberstellung einer JVx und Java EE Anwendung

5.1.1 JVx Client-Schicht zu Java EE Web-Schicht

Die Webkomponenten einer Java EE Web-Schicht können Java Servlets, Webseiten der Java Server Faces Technologie oder Java Server Pages, welche in einem »Web Container« (siehe Abschnitt 2.4.1) ausgeführt werden sein. Java Servlets sind Java Klassen, welche Anfragen (Request) des Clients entgegennehmen, Antworten (Response) an den Client versenden und daher eher für serviceorientierte Anwendungen zum Einsatz kommen. Java Technologien wie Java Server Faces werden für interaktive Webanwendung verwendet. Bei der Entwicklung einer Java EE Anwendung werden dann je nach Art der Anwendung die Benutzeroberflächen aufbauend auf einer dieser Technologien für die Darstellung am Client entwickelt. Zur Anzeige und Speicherung der Daten erfolgt eine Interaktion mit den Komponenten der Java EE Business-Schicht. [6]

Die Client-Schicht einer JVx Anwendung wird aufbauend auf den »GUI Controls« der generischen Benutzerschnittstelle entwickelt. Diese »GUI Controls« dienen als Hülle für die unterschiedlichsten Technologien. Zur Anzeige und Speicherung der Daten werden sie mit dem generischen Modell abgeglichen. Dieses Modell übernimmt dann die Kommunikation mit den Komponenten der JVx Server-Schicht, um Daten zu lesen bzw. zu speichern. Siehe dazu Abschnitt 4.1.2.

5.1.2 JVx Server-Schicht zu Java EE Business-Schicht

Die Java EE Business-Schicht besteht aus den sogenannten Enterprise Java Beans (EJB), welche im »EJB Container« eines Java EE Servers ausgeführt werden (siehe Abschnitt 2.4.1). Diese

Enterprise Beans bilden die Geschäftslogik einer Java EE Anwendung ab. Es existieren zwei Typen von Enterprise Beans, die sogenannten »Session Beans« und »Message Driven Beans«. Ein »Message Driven Bean« erlaubt Java EE Anwendungen Nachrichten asynchron abzufragen. Ein »Session Bean« kapselt die Geschäftslogik in Funktionen, welche vom Client aus aufgerufen werden. Es wird zwischen drei Arten von »Session Beans« unterschieden, welche als zustandsbehaftete (engl.: stateful), zustandslose (engl.: stateless) und einelementige (engl.: singleton) »Session Beans« bezeichnet werden. Siehe dazu Tabelle 5.1.

Java EE Komponente	Verwendung
Stateful Session Beans	Der Zustand eines Objekts wird durch die Werte seiner Variablen repräsentiert. In einem »Stateful Session Bean« speichern die Instanzvariablen einen bestimmten Status für einen bestimmten Client. Dies bedeutet, ein »Stateful Session Bean« ist eindeutig über eine ID einem Client zugewiesen und es ermöglicht Informationen für die Zeit einer Sitzung (Session) zu speichern. Änderungen des Zustands sind im Folgeaufruf durch den gleichen Client immer noch sichtbar. [6]
Stateless Session Beans	Ein »Stateless Session Bean« speichert keinen Zustand eines Clients innerhalb einer Sitzung, sondern nur für die Zeit des Aufrufs einer Funktion. Daher müssen für die Abarbeitung einer Funktion immer alle benötigten Information als Parameter übergeben werden. [6]
Singleton Session Beans	Ein »Singleton Session Bean« existiert pro Anwendung und es wird beim Start der Anwendung instanziert. Es hat die Eigenschaft, dass kein clientspezifischer Zustand, sondern ein globaler Zustand, welcher für alle Clients sichtbar ist gespeichert wird. [6]

Tabelle 5.1: Java EE 6 Komponenten der Business-Schicht

Die Server-Schicht einer JVx Anwendung besitzt sogenannte »Lifecycle Objekte«. Diese »Lifecycle Objekte« enthalten die Geschäftslogik einer JVx Anwendung. JVx bietet die Möglichkeit »Lifecycle Objekte« pro Anwendung, pro Session oder pro SubSession zu erzeugen. Siehe dazu Abschnitt 4.3.3. Ein »Lifecycle Objekt« einer Anwendung kann mit einem »Singleton Session Bean« einer Java EE Anwendung und ein »Lifecycle Objekt« einer Session mit einem »Stateful Session Bean« verglichen werden.

Zum Lesen, Speichern, Ändern und Löschen der Daten aus einer Datenbank wird in Java EE Anwendungen das Java Persistence API (siehe Kapitel 3) verwendet. Dieses benötigt Entitäten, um Daten aus der Datenbank zu kapseln und der Anwendung zur Verfügung zu stellen. In einigen Java EE Anwendungen gibt es zusätzlich zu den EJBs noch eine weitere Schicht, zum Zugriff auf die Entitäten. Es handelt sich hier um die sogenannten »Data Access Objekte« (DAO) bzw. »Entity Access Objekte« (EAO). Diese DAO bzw. EAO Objekte besitzen Funktionen zum Lesen,

Speichern, Ändern und Löschen von Daten bzw. Entitäten. DAOs und EAOs kommen aber in neuen Java EE 6 Anwendungen eher weniger zum Einsatz. Es werden nur mehr EJBs, welche den Zugriff auf die Entitäten direkt über den Entitäten-Manager (siehe Abschnitt 3.3.1) regeln, verwendet. Im Prinzip kann ein EJB, welches den Entitäten-Manager zum Zugriff auf eine Entität verwendet, auch als EAO angesehen werden.

JVx verwendet für den Zugriff auf die Daten einer Datenbank das *DBStorage* Objekt. Die Klasse dieses »Storage Objekts« ist eine Implementierung der *javax.rad.persist.IStorage* Schnittstelle und somit auch eine Implementierung des Persistenz API von JVx. Siehe dazu Abschnitt 4.2. Die Datenbankanbindung erfolgt dann über JDBC.

5.2 Qualitätskriterien zur Integration von JPA in JVx

Der Zugriff auf Daten aus der Datenbank erfolgt in JVx direkt über JDBC. JDBC erlaubt dem Entwickler, eine Datenbankverbindung aufzubauen, zu verwalten, SQL Anfragen an die Datenbank zu senden und die Ergebnisse dem Java Programm zur Verfügung zu stellen. Die Verwendung von JDBC hat einige Nachteile gegenüber JPA [26]. Durch die Integration von JPA in JVx können Vorteile dieser Technologie gegenüber JDBC auch in JVx ausgenutzt werden. Diese Vorteile können gleichzeitig als Qualitätskriterien für die Verwendung von JPA in JVx angesehen werden. Die Integration von JPA in JVx soll aber auch keine architektonische Veränderung im Aufbau einer JVx Anwendung mit sich bringen. Sie soll die bereits etablierten Besonderheiten und Funktionalitäten einer JVx Anwendung ebenfalls unterstützen. Diese Eigenschaften führen zu einer Liste von Qualitätskriterien, welche bei der Integration von JPA in JVx berücksichtigt werden fmüssen und Verbesserungen bzw. Erweiterungen bei der Verwendung von JVx bringen.

5.2.1 Unabhängigkeit des JDBC Treibers

In JPA erfolgt die Konfiguration des Zugriffs auf eine Datenbank (Datenbankanbieter, Datenbankname, Server, Username und Passwort) über die Persistenzeinheit (siehe Abschnitt 3.3.3). JPA kümmert sich um den korrekten Auf- und abbau der Verbindung und um die datenbankanbieterspezifischen Merkmale. Über JDBC muss der Verbindungsauf- und Abbau über den Quellcode speziell für einen bestimmten Datenbankanbieter geregelt werden [27]. In JVx ist diese Verwaltung für die Datenbankverbindung über die Kasse *com.sibvisions.rad.persist.jdbc.DBAccess* geregelt und sie ermöglicht so dem Entwickler einen relativ einfachen Zugang zu einer Datenbank. Die Klasse *DBAccess* ist eine Standardimplementierung für alle ANSI Datenbanken. JVx bietet für die gängigsten Datenbankanbieter Subklassen der Klasse *DBAccess*. In diesen Klassen ist bereits der richtige JDBC Treiber definiert und es werden JDBC Probleme zu einzelnen Datenbanken behoben. Für folgende Datenbankanbieter stehen bereits fertige Implementierungen bereit:

- DB2 [49] über die Klasse *com.sibvisions.rad.persist.jdbc.DB2DBAccess*
- Derby [50] über die Klasse *com.sibvisions.rad.persist.jdbc.DerbyDBAccess*
- HSQLDB [51] über die Klasse *com.sibvisions.rad.persist.jdbc.HSQLDBAccess*
- MSSQL [52] über die Klasse *com.sibvisions.rad.persist.jdbc.MSSQLDBAccess*
- MySql [53] über die Klasse *com.sibvisions.rad.persist.jdbc.MySQLDBAccess*

- Oracle [54] über die Klasse *com.sibvisions.rad.persist.jdbc.OracleDBAccess*
- PostgreSQL [55] über die Klasse *com.sibvisions.rad.persist.jdbc.PostgreSQLDBAccess*

Die Klasse *com.sibvisions.rad.persist.jdbc.DBAccess* bietet Funktionen für die verschiedensten Datenbankoperationen (z.B.: Absetzen von SQL Anfragen, Aufruf einer Prozedur am Datenbankserver). Es werden zwar die gängigsten Datenbankanbieter abgedeckt, jedoch unterstützen Persistenzanbieter (siehe Abschnitt 3.3.4) wie TopLink mehr Datenbankanbieter [56] als zum Beispiel Sybase [57]. Sogar Objektdatenbanken wie ObjectDB [58] werden durch JPA unterstützt.

5.2.2 Datenbankunabhängigkeit der Abfragesprache

JDBC unterstützt nur native SQL Ausdrücke. Dabei ist der Entwickler für die Erstellung von datenbankspezifischen effizienten Anfragen und für das Mappen der Daten in die passende Objektstruktur verantwortlich. JPA verwendet für die Abfrage von Daten eine eigene Abfragesprache (siehe 3.3.5). Diese ist im Gegensatz zu nativen SQL Anfragen datenbankunabhängig [27]. Abfragen auf die Datenbank müssen im Quellcode nicht geändert werden, wenn auf einen anderen Datenbankanbieter gewechselt wird. Enthält eine JVx Anwendung in der Geschäftslogik spezifische SQL Ausdrücke, so könnte es sein, dass diese bei einem Wechsel zu einem anderen Datenbankanbieter angepasst werden müssen. Ein weiterer Vorteil von JPA liegt darin, dass auch native SQL Ausdrücke unterstützt werden.

JPA bieten neben der eigenen datenbankunabhängigen Abfragesprache einen effizienten automatischen Weg zum Lesen und Bearbeiten von Daten. Siehe dazu auch den nächsten Abschnitt 5.2.3.

5.2.3 Optimierung der Performance

JPA bietet standardmäßig Cachingmechanismen an, um die Zugriffe auf die Datenbank zu minimieren. Der unterste Cache (First Level Cache) von JPA wird durch den Persistenzkontext abgebildet (siehe Abschnitt 3.3.2). Bei mehrmaligem Zugriff auf eine Entität hintereinander wird nur einmal auf die DB zugegriffen. JPA ermöglicht auch die Performancesteigerung einer Anwendung über einen übergeordneten Cachingmechanismus (Second Level Cache). Diese wird vom jeweiligen Persistenzanbieter zur Verfügung gestellt. Nähere Informationen können in [6] nachgelesen werden. Die Funktionsweise eines Cachingmechanismus ist unter [28] beschrieben.

5.2.4 Erzeugung und Änderung des Domänenmodells aus den Entitäten

JPA verfolgt den Ansatz, den Entwickler von datenbankspezifischen Entwicklungsaufgaben zu entlasten. Darunter fällt auch die Erzeugung und Änderung von Tabellen eines Datenbankschemas. Existiert noch keine Datenbank für eine Anwendung, kann der Entwickler diese durch die zuvor erzeugten Entitäten generieren lassen. Nachträgliche Änderungen in der Datenbank können durch Änderung der Entitäten durchgeführt werden. Unter JVx muss diese Aufgabe vom Entwickler direkt über den Datenbankserver mittels SQL-Statements vorgenommen werden.

5.2.5 Beibehalten der Architektur einer JVx Anwendung

Durch die Integration von JPA in JVx soll der bestehende Aufbau einer JVx Anwendung beibehalten werden. Siehe dazu Abschnitt 4.3. Die Anbindung von JPA soll also über die bestehenden

Schnittstellen von JVx ermöglicht und gegebenenfalls um erforderliche Schnittstellen erweitert werden. Dadurch kann sichergestellt werden, dass bestehende Mechanismen von JVx nicht in deren Funktionalität eingeschränkt werden.

5.2.6 Unterstützung der bestehenden JVx Funktionalitäten

Die Integration von JPA soll die bestehende Funktionalität einer JVx Anwendung unterstützen. Darunter fallen auch die in Abschnitt 4.3 angeführten Funktionen:

- Automatische Verlinkung: Datenquellen, welche eine Viele-zu-eins Beziehung zu einer Datenquelle aufweisen, werden automatisch verlinkt.
- Master-Detail-Beziehung: Diese ermöglicht auf einfache Art und Weise eine Verknüpfung zwischen zwei Datenquellen, welche eine Eins-zu-viele oder eine Viele-zu-viele Beziehung zueinander besitzen, herzustellen.

5.2.7 Unterstützung verschiedenster Persistenzanbieter

JPA soll so in JVx integriert werden, dass unterschiedlichste Persistenzanbieter wie EclipseLink oder OpenJPA verwendet werden können. Es kann also jeder Persistenzanbieter für den Zugriff auf die Datenbank verwendet werden, wenn dieser auf dem Standard von JPA 2.0 aufbaut.

5.2.8 Datenbankunabhängige Erzeugung der Metadaten

Die Metadaten, welche den Aufbau einer Tabelle in einem Datenbankschema beschreiben, sind ein wesentlicher Teil einer JVx Anwendung. Siehe dazu Abschnitt 4.2.1. Diese Metadaten werden über datenbankanbieterspezifische Abfragen aus der Datenbank ermittelt, was eine sehr aufwändige Prozedur darstellt. Durch die Integration von JPA können diese Metadaten aufbauend auf den Entitäten und deren Beziehungen zueinander erzeugt werden. Dadurch ist eine Datenbankunabhängigkeit gewährleistet.

5.2.9 Einfache Verwendung von JPA in JVx

Die Verwendung von JPA in JVx soll für den Entwickler einer JVx Anwendung einfach und verständlich sein. Es soll auch ermöglicht werden, JPA in bereits vorhandene JVx Anwendungen durch wenig Codeänderungen zu integrieren. Ein Framework bzw. eine Frameworkerweiterung darf nicht kompliziert zu verwenden sein, um von den Anwendern auch eingesetzt zu werden.

5.2.10 Verwendung von eigenen DAOs bzw. EAOs

JVx soll ermöglichen, dass eigene »Data Access Objekte« (DAO) bzw. »Entity Access Objekte« (EAO) für den Zugriff auf die Entitäten in einer JVx Anwendung verwendet werden können. Bei den EAOs kann es sich auch um EJBs, welche den Zugriff auf die Entitäten direkt über den Entitäten-Manager ermöglichen, handeln. Wichtig ist nur, dass diese EAOs Funktionen zum Lesen, Speichern, Ändern und Löschen von Entitäten besitzen.

5.3 Integration von JPA in JVx

Die Integration von JPA in JVx erfolgt über die *IStorage* (siehe Abschnitt 4.2) Schnittstelle von JVx. Dieses Persistenz API von JVx wird von der Client-Schicht verwendet, um die CRUD (Create, Read, Update und Delete) Operationen auf die Daten zu ermöglichen. Abbildung 5.3 zeigt, wie die Integration ermöglicht wurde. Es wurde dafür ein eigenes »Storage Objekt« *JPAStorage*, welches eine Ableitung der *IStorage* Schnittstelle ist, erzeugt. Das Objekt *JPAAccess* verwaltet den Zugriff auf die Entitäten entweder über das interne generische EAO *GenericEAO* oder über ein externes DAO bzw. EAO, welches vom Entwickler einer JVx Anwendung zur Verfügung gestellt wird.

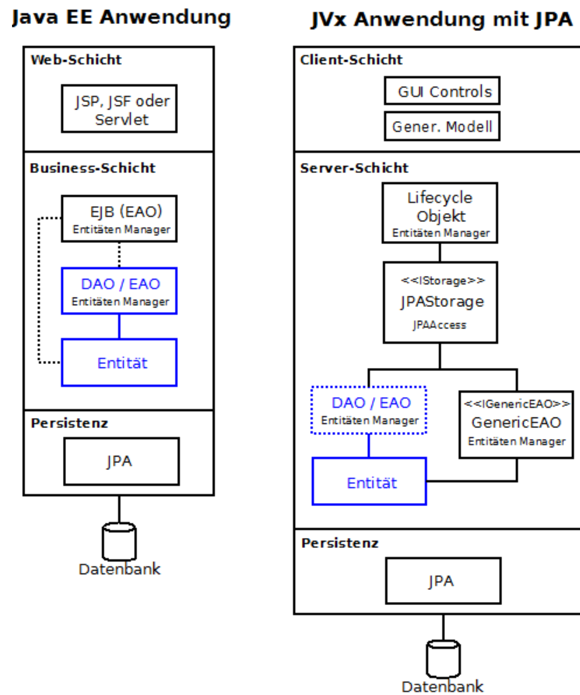

Abbildung 5.3: Integration von JPA in JVx

Abbildung 5.4 stellt diese Integration etwas genauer in einem abstrakten Klassendiagramm gegenüber.

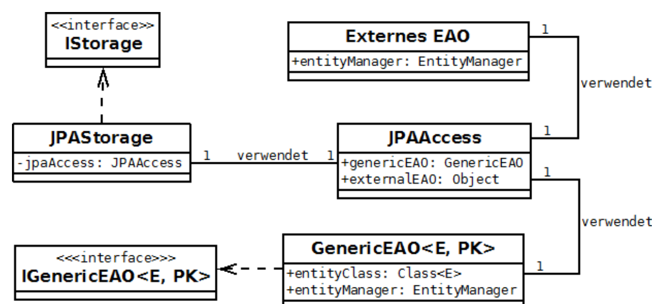

Abbildung 5.4: Klassendiagramm zur Integration von JPA in JVx

5.3.1 Die Klasse JPAStorage

Die Klasse *com.sibvisions.rad.persist.jpa.JPAStorage* ist eine Ableitung der Schnittstelle *javax.rad.persist.IStorage* und sie kann dadurch vom generischen Modell einer JVx Anwendung verwendet werden. Abbildung 5.5 stellt die wichtigsten Funktionen und Attribute dieser Klasse in einem Klassendiagramm dar. *JPAStorage* ist für das Mappen der Daten aus dem generischen Modell in die passende Entität und umgekehrt verantwortlich, je nach dem welche Funktion des »Storage Objekts« aufgerufen wird. Die Abbildung der Entitäten zum Austausch der Daten zwischen JPA und JVx, ist in der Klasse *JPAServerMetaData* dargestellt. Siehe dazu Abschnitt 5.3.5.

Ein »Storage Objekt« der Klasse *JPAStorage* ermöglicht also im Grunde den Zugriff auf Objekte einer bestimmten Entität-Klasse. Daher wird bei der Erzeugung die Klasse jener Entität übergeben, welches von dem »Storage Objekt« verwaltet wird. Dieses wird im Attribut *masterEntity* gespeichert. Das Attribut *detailEntity* dient dazu, ein »Storage Objekt« der Klasse *JPAStorage* zu erzeugen, welches eine Viele-zu-viele Beziehung zwischen zwei Entitäten abbildet. Siehe dazu Listing 6.12.

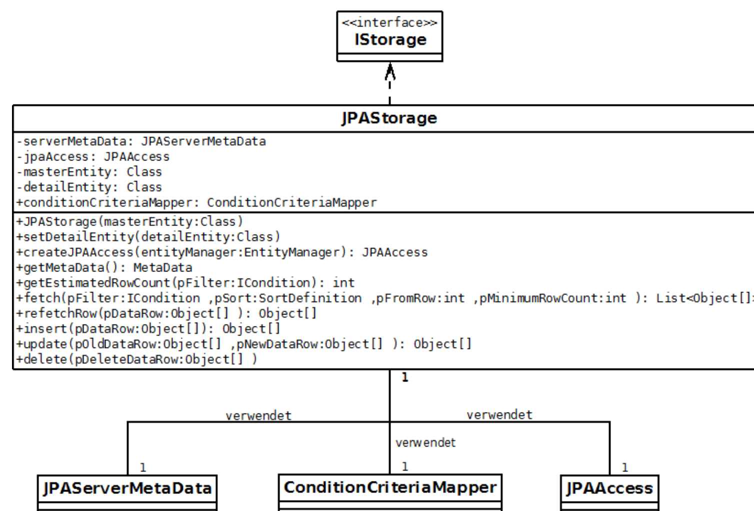

Abbildung 5.5: Klassendiagramm der JPAStorage Klasse

Die Funktion *List<Object[]> fetch(ICondition pFilter, SortDefinition pSort, int pFromRow, int pMinimumRowCount)* ermöglicht Einschränkungskriterien und Sortierungen von Daten. Damit diese Einschränkungs- und Sortierungskriterien auch auf die Abfrage von Entitäten angewandt werden können, muss eine Umwandlung der JVx Einschränkungskriterien (siehe Abschnitt 4.2.3) in passende JPA Einschränkungskriterien (siehe Abschnitt 3.3.5) erfolgen. Diese Funktionalität wird von der Klasse *com.sibvisions.rad.persist.jpa.ConditionCriteriaMapper* übernommen. *JPAStorage* kapselt außerdem noch ein Objekt der Klasse *JPAAccess*. Siehe dazu Abschnitt 5.3.2.

5.3.2 Die Klasse JPAAccess

Die Klasse *JPAAccess* kapselt ein Objekt der Klasse *GenericEAO* (siehe Abschnitt 5.3.4) und eventuell ein externes EAO zum Zugriff auf die Entitäten. Abbildung 5.6 stellt die wichtigsten Funktionen und Verweise grafisch in einem Klassendiagramm dar.

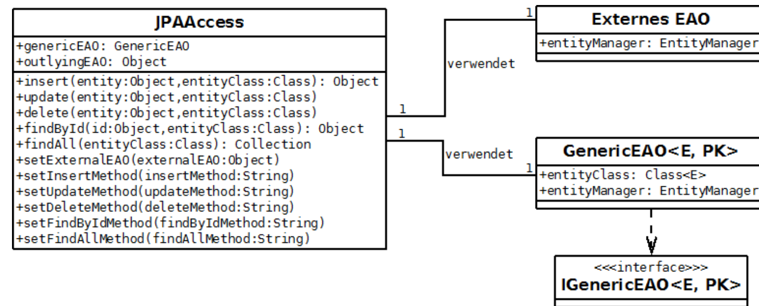

Abbildung 5.6: Klassendiagramm der JPAAccess Klasse

Gibt es ein externes EAO (wurde über die Funktion *setExternalEAO(Object externalEAO)* gesetzt), so werden die passenden Funktionen für den Zugriff auf die Entitäten von diesem Objekt beim Aufruf einer der Funktionen *insert*, *update*, *delete*, *findById*, *findAll* von *JPAAccess* verwendet. Existiert kein externes EAO bzw. tritt ein Fehler beim Zugriff auf eine Funktion des externen EAOs auf, so wird die passende Funktion von *genericEAO* aufgerufen.

5.3.3 Externes EAO

Unter einem externen »Entity Access Objekt« (EAO) wird jedes Objekt verstanden, welches Funktionen zum Lesen, Speichern, Ändern und Löschen von Entitäten besitzt. Es kann von einem Entwickler einer JVx Anwendung selbst erstellt und für den Zugriff auf die Entitäten gesetzt werden. Der Vorteil eines externen EAOs liegt darin, dass es dem Entwickler ermöglicht, Entitäten noch zu verändern bzw. zu überprüfen, bevor diese mit der Datenbank abgeglichen werden. Listing 5.1 zeigt ein Beispiel für ein EAO.

```
public class KundeEAO  {
    ...
    @EAOMethod(methodIdentifier = EAO.INSERT)
    public Kunde insertKunde(Kunde kunde) {

        if(kunde.getTelefonnummer() == null) {
            kunde.setTelefonnummer("geheim");
        }

        entityManager.getTransaction().begin();

        entityManager.persist(kunde);

        entityManager.getTransaction().commit();

        return kunde;
    }
    ...
}
```

Listing 5.1: Beispiel für ein EAO

Das Problem bei externen EAOs ist, dass nicht bekannt ist, welche Funktionen die benötigten Grundfunktionalitäten von *JPAAccess* *insert*, *update*, *delete*, *findById* und *findAll* bereitstellen. Dieses Problem kann auf drei verschiedene Arten gelöst werden.

EAO implementiert IGenericEAO

Implementiert das externe EAO das Interface *com.sibvisions.rad.persist.jpa.IGnerericEAO*, so sind die benötigten Funktionen bekannt. Siehe dazu Listing 5.2.

```
public class KundeEAO implements IGenericEAO<Kunde, Integer> {

    @Override
    public Kunde insert(Kunde newEntity) {
        ...
    }

    @Override
    public void update(Kunde entity) {
        ...
    }

    @Override
    public void delete(Kunde entity) {
        ...
    }

    @Override
    public Kunde findById(Integer id) {
        ...
    }

    @Override
    public Collection<Kunde> findAll() {
        ...
    }
}
```

Listing 5.2: EAO implementiert IGenericEAO

EAO verwendet Annotationen

Es wird auch die Möglichkeit geboten die entsprechenden Funktionen über eine Annotation zu kennzeichnen. Dafür gibt es ein eigenes Interface *com.sibvisions.rad.persist.jpa.EAOMethod*. Listing 5.4 zeigt ein Beispiel für den Einsatz dieser Annotation.

```
public class KundeEAO  {

    ...

    @EAOMethod(methodIdentifier = EAO.INSERT)
    public Kunde insertKunde(Kunde kunde) {
        ...
    }

    @EAOMethod(methodIdentifier = EAO.DELETE)
    public void deleteKunde(Kunde kunde) {
        ...
    }

    ...
}
```

Listing 5.3: Beispiel für ein EAO mit Annotation

Die Funktionen können über die in Tabelle 5.2 definierten Typen identifiziert werden.

Funktions Typ	Beschreibung
INSERT	Bei Aufruf dieser Funktion wird die übergebene Entität eingefügt.
UPDATE	Bei Aufruf dieser Funktion wird die übergebene Entität gespeichert.
DELETE	Bei Aufruf dieser Funktion wird die übergebene Entität gelöscht.
FIND_BY_ID	Diese Funktion findet eine Entität anhand der übergebenen Id.
FIND_ALL	Diese Funktion findet alle Entitäten einer bestimmten Klasse.

Tabelle 5.2: Funktions Typen von com.sibvisions.rad.persist.jpa.EAOMethod

Setzen der Funktionsnamen des EAOs in JPAAccess

Da es sein kann, dass ein externes EAO aus einem bereits bestehenden Java Archiv verwendet werden soll und der Code für dieses EAO nicht verändert werden kann, so wird auch die Möglichkeit gegeben, die Funktionsnamen des EAO für die entsprechenden Funktionen direkt im Objekt der Klasse *JPAAccess* zu setzen. *JPAAccess* besitzt die Funktionen *setInsertMethod(String insert)*, *setUpdateMethod(String update)*, *setDeleteMethod(String delete)*, *setFindByIdMethod(String findById)* und *setFindAllMethod(String findAll)*, über die es möglich ist, die Funktionen, welche aufgerufen werden sollen, zu definieren.

```
JPAAccess jpaAccess = jpaKunde.createJPAAccess(getEntityManager());
jpaAccess.setInsertMethod("insertKunde");
jpaAccess.setUpdateMethod("updateKunde");
jpaAccess.setDeleteMethod("deleteKunde");
jpaAccess.setFindByIdMethod("findKunde");
jpaAccess.setFindAllMethod("findAllKunde");
```

Listing 5.4: JPAAccess setter-Funktionen für Funktionsnamen

5.3.4 Das generische EAO Design

Das generische EAO Design [29] besteht aus einem generischen Interface *com.sibvisions.rad.persist.jpa.IGenericEAO<E, PK>*, welches die grundlegenden Funktionen für den Zugriff auf Entitäten definiert und die generische Java Technologie [30] verwendet. Der erste Typparameter E definiert den Typ der Entität und der zweite Parameter den Typ des Primärschlüssels der Entität. Die Klasse *com.sibvisions.rad.persist.jpa.GenericEAO<E, PK>* implementiert das Interface. Abbildung 5.7 stellt dieses Design in einem Klassendiagramm dar.

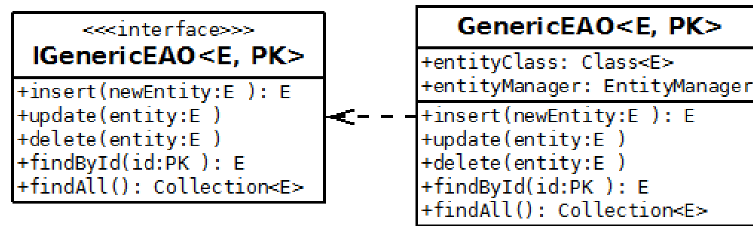

Abbildung 5.7: Klassendiagramm des generischen EAO Designs

5.3.5 Die Klasse JPAServerMetaData

Pro *JPAStorage* Objekt gibt es ein Objekt der Klasse *JPAServerMetaData*. Diese Klasse kapselt jene Informationen, welche für den Austausch der Daten zwischen JPA und JVx benötigt werden. Der Aufbau des generischen Modells ist durch die Klasse *MetaData* (siehe Abschnitt 4.2.1) bestimmt. Die benötigten Klassen und die wichtigsten Funktionen und Attribute dafür werden in Abbildung 5.8 dargestellt.

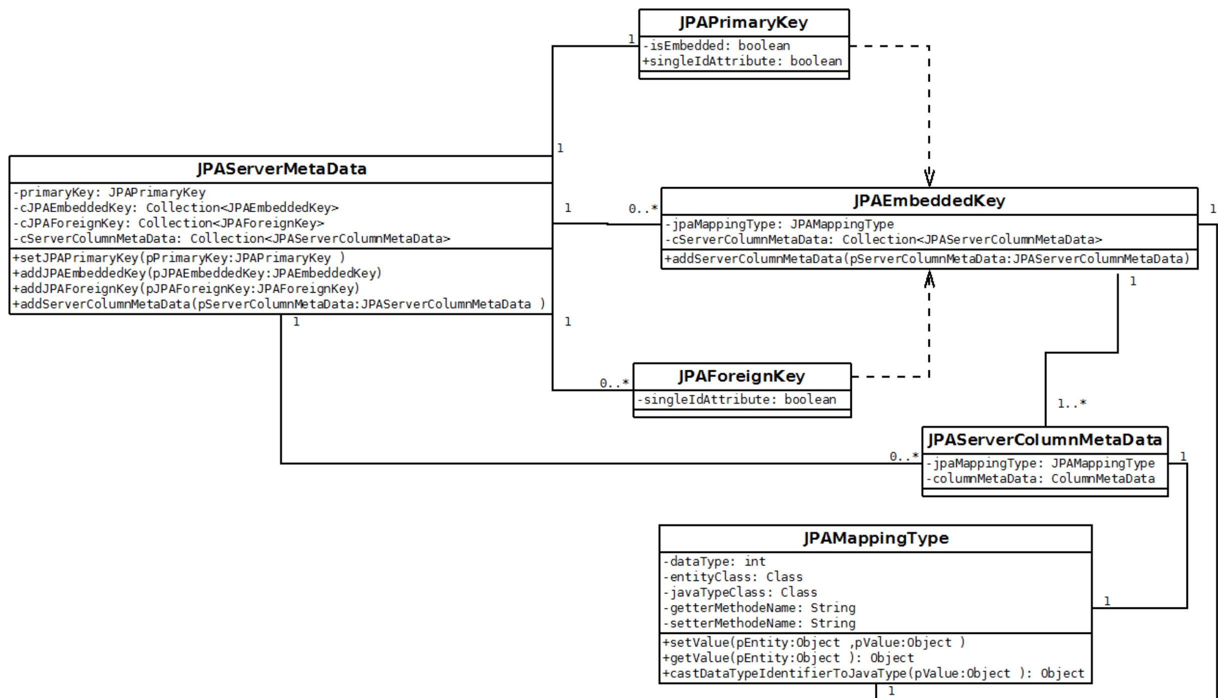

Abbildung 5.8: Klassendiagramm zur Klasse JPAServerMetaData

Der Aufbau einer Entität wird in der *JPAServerMetaData* Klasse abgebildet. Eine Entität kapselt genau einen Primärschlüssel (*JPAPrimaryKey*), eventuell mehrere Fremdschlüssel (*JPAForeignKey*) und mehrere eingebettete Objekte (*JPAEmbeddedKey*). *JPAPrimaryKey* und *JPAForeignKey* sind eine Ableitung von *JPAEmbeddedKey*, da diese ähnlich wie eingebettete Objekte in einer Entität behandelt werden können. Listing 5.5 zeigt ein Beispiel einer Entität und Abbildung 5.9 das Abbildungsmodell.

```
@Entity
public class Kunde implements Serializable {

    @Id
    private int id;

    @Id
    private int svn;

    @ManyToOne
    private Anrede anrede;

    private String vorname;

    private String nachname;

    @Embedded
    private Adresse adresse;

    ...
 }

@Embeddable
public class Address {

    private String strasse;
    private String plz;

    ...
}
```

Listing 5.5: Beispiele einer Entität mit Primär- Fremdschlüssel und eingebetteter Klasse

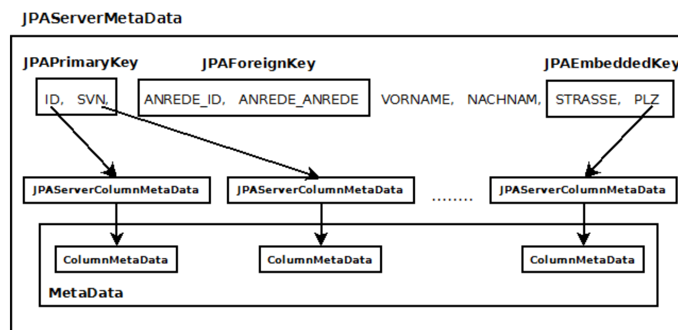

Abbildung 5.9: Abbildungsmodell von JPAServerMetaData

Die Klasse *JPAServerColumnMetaData* kapselt ein Objekt der Klasse *ColumnMetaData* und repräsentiert dadurch genau ein einzelnes primitives Attribut einer Entität (z.B.: String vorname, String strasse). Die Klasse *JPAMappingType* kapselt dann zu jedem Attribut einer Entität die Informationen, um die Daten korrekt zwischen Entitäten und dem generischen Modell zu mappen. Genauere Erklärungen zu diesen Klassen finden sich in der Java Dokumentation im Quellcode der Integration von JPA in JVx [59].

6 Beispielanwendung für den Einsatz von JPA in JVx

In diesem Kapitel wird gezeigt, wie JPA in einer JVx Anwendung verwendet werden kann. Als Beispiel wird die in Abschnitt 4.3 beschriebene Anwendung herangezogen und es werden die benötigten Entwicklungsschritte beschrieben. Die Client-Schicht einer JVx Anwendung bleibt bei der Verwendung von JVx unverändert. Es sind nur wenige Entwicklungsschritte in der Server-Schicht notwendig. Danach erfolgt eine Evaluierung der in Abschnitt 5.2 angeführten Qualitätskriterien zur Integration von JPA in JVx.

6.1 Entwickeln der Entitäten

Abbildung 4.4 zeigt die benötigte Datenstruktur für die Beispielanwendung. Aus diesem Klassendiagramm können die Entitäten und deren Beziehung zueinander abgelesen werden. Daraus ist zu erkennen, dass vier Entitäten zu erzeugen sind. Diese werden in der Server-Schicht einer JVx Anwendung abgelegt.

Listing 6.1 zeigt die Entität *Anrede*. Die Id der *Anrede* wird über die Annotation *javax.persistence.Id* festgelegt. Die Annotation *javax.persistence.GeneratedValue* ermöglicht, dass diese Id automatisch erzeugt wird.

```
@Entity
public class Anrede implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String anrede;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getAnrede() {
        return anrede;
    }
    public void setAnrede(String anrede) {
        this.anrede = anrede;
    }

}
```

Listing 6.1: Entität Anrede der Beispielanwendung

Listing 6.2 zeigt die Entität *Adresse* der Beispielanwendung. Die Annotation *javax.persistence.ManyToOne* bildet die Viele-zu-eins Beziehung zwischen *Adresse* und *Kunde* ab.

```
@Entity
public class Adresse implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String strasse;
    private int plz;
    private String ort;

    @ManyToOne
    private Kunde kunde;


    //Getter- und Setter-Funktionen

    ...
}
```

Listing 6.2: Entität Adresse der Beispielanwendung

Listing 6.3 zeigt die Entität *Kunde*. Die Annotation *javax.persistence.ManyToOne* bildet die Viele-zu-eins Beziehung zwischen *Kunde* und *Anrede* ab. Die Annotation *javax.persistence.ManyToMany* stellt die Viele-zu-viele Beziehung von *Kunde* zu *Ausbildung* und die Annotation *javax.persistence.OneToMany* die Eins-zu-viele Beziehung zwischen *Kunde* und *Adresse* dar. Der *FetchType.LAZY* gibt an, dass beim Laden der Entität *Kunde* die *Collection* für *Ausbildung* und *Adresse* nicht initialisiert werden soll. Erst beim Zugriff auf die *Collection* werden die benötigten Daten geladen. Diese Eigenschaft verbessert die Performance beim Laden der Entität *Kunde*. Würde *FetchType.EAGER* verwendet werden, so würde beim Laden eines *Kunden* auch die *Collection* befüllt werden. *cascade=ALL* ist in Abschnitt 3.2.3 beschrieben.

```
@Entity
public class Kunde implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    @ManyToOne
    private Anrede anrede;

    private String vorname;
    private String nachname;

    private Date geburtsdatum;

    @ManyToMany(cascade=ALL, fetch = FetchType.LAZY)
    private Collection<Ausbildung> ausbildung = new ArrayList<Ausbildung>();

    @OneToMany(cascade=ALL, fetch = FetchType.LAZY, mappedBy="kunde")
    private Collection<Adresse> adresse = new ArrayList<Adresse>();

    //Getter- und Setter-Funktionen
```

```
    ...
}
```

Listing 6.3: Entität Kunde der Beispielanwendung

Listing 6.4 zeigt die Entität *Ausbildung* der Beispielanwendung.

```
@Entity
public class Ausbildung implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private int id;

    private String ausbildung;

    //Getter- und Setter-Funktionen

    ...
}
```

Listing 6.4: Entität Ausbildung der Beispielanwendung

6.2 Entwickeln der EAOs

Dieser Schritt ist nicht unbedingt notwendig, außer es wird zusätzliche Funktionalität benötigt, bevor die Entitäten mit der Datenbank abgeglichen werden. Abschnitt 5.3.3 zeigt die verschiedenen Möglichkeiten für die Verwendung von eigenen EAOs in JVx. In diesem Beispiel wurde ein eigenes EAO für den Zugriff auf die Entität *Kunde* erzeugt. Siehe dazu Listing 6.5. Dieses EAO wirft eine *DataSourceException*, wenn beim Aufruf der Funktion *insertKunde* der Entität Kunde kein Geburtsdatum vergeben wurde.

```
public class KundeEAO {

    private EntityManager entityManager;

    public void setEntityManager(EntityManager entityManager) {
        this.entityManager = entityManager;
    }

    @EAOMethod(methodIdentifier = EAO.INSERT)
    public Kunde insertKunde(Kunde kunde) throws DataSourceException {

        if(kunde.getGeburtsdatum() == null) {
            throw new DataSourceException("Es muss ein Geburtsdatum gesetzt werden");
        }

        entityManager.getTransaction().begin();
        entityManager.persist(kunde);
        entityManager.getTransaction().commit();

        return kunde;
    }
}
```

Listing 6.5: EAO für die Entität Kunde der Beispielanwendung

6.3 Erzeugen der Persistenzeinheit

Listing 6.6 zeigt die Persistenzeinheit der Beispielanwendung. Als Persistenzanbieter (siehe Abschnitt 3.3.4) wurde EclipseLink verwendet. Die benötigten Bibliotheken für den gewählten Persistenzanbieter müssen in die JVx Anwendung integriert werden. Die Persistenzeinheit enthält alle Entitäten und die Verbindungsdaten zur Datenbank. In diesem Beispiel wurde eine HSQLDB verwendet. Die Eigenschaft *eclipselink.ddl-generation* mit dem Wert *create-tables* gibt an, dass die Datenbank aus den Entitäten erzeugt werden soll. Die Persistenzeinheit wird in der Datei *persistence.xml* im Verzeichnis *META-INF* abgelegt.

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                        http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
    version="2.0" xmlns="http://java.sun.com/xml/ns/persistence">

    <persistence-unit name="pu" transaction-type="RESOURCE_LOCAL">

    <class>entity.Anrede</class>
    <class>entity.Kunde</class>
    <class>entity.Adresse</class>
    <class>entity.Ausbildung</class>

    <properties>
      <property name="javax.persistence.jdbc.driver" value="org.hsqldb.jdbcDriver" />
      <property name="javax.persistence.jdbc.url"
                              value="jdbc:hsqldb:hsql://localhost/jvxbeispiel" />

      <property name="javax.persistence.jdbc.user" value="sa" />
      <property name="javax.persistence.jdbc.password" value="" />

      <property name="eclipselink.ddl-generation" value="create-tables" />
    </properties>

    </persistence-unit>

</persistence>
```

Listing 6.6: Persistenzeinheit der Beispielanwendung

6.4 Erzeugen des Entitäten-Managers

Die Verwaltung des Entitäten-Managers kann entweder von einem Java EE Container (*Container-MANAGED*) oder von der Anwendung (*Application MANAGED*) selbst vorgenommen werden. Siehe dazu Abschnitt 3.3.1. Wird die JVx Anwendung auf einem Java EE Server (z.b.: JBoss Application Server [60]) ausgeführt, so könnte ein *Container-MANAGED* Entitäten-Manager verwendet werden. Siehe dazu Listing 6.7 und 6.9.

```
@PersistenceContext(unitName="pu")
public EntityManager em;
```

Listing 6.7: Erzeugen eines *Container-MANAGED* Entitäten-Managers

Alternativ kann der *Container-MANAGED* Entitäten-Manager auch über JNDI erzeugt werden.

```
@PersistenceContext(unitName="pu")
@Resource
SessionContext ctx;

EntityManager em = (EntityManager)ctx.lookup("KundenverwaltungEM");
```

Listing 6.8: Erzeugen eines *Container-MANAGED* Entitäten-Managers über JNDI

In diesem Beispiel wurde ein *Application MANAGED* Entitäten-Manager über die Klasse *javax.persistence.EntityManagerFactory* erzeugt.

```
public EntityManager getEntityManager() throws Exception {

  EntityManagerFactory emf = Persistence.createEntityManagerFactory("pu1");
  EntityManager entityManager = emf.createEntityManager();

  return entityManager;

}
```

Listing 6.9: Erzeugen eines *Application MANAGED* Entitäten-Managers über die *EntityManager-Factory*

6.5 Funktionen für den Zugriff auf JPAStorage

In der Beispielanwendung in Abschnitt 4.3 wurde ein *DBStorage* Objekt für den Zugriff auf die Daten verwendet. Diese Funktionen müssen nun für das *JPAStorage* angepasst werden. Listing 6.10 zeigt den Zugriff auf die Entität *Kunde* über die Klasse *JPAStorage*. Die Klasse der Entität *Kunde* wird dem *JPAStorage* Objekt bei der Erzeugung im Konstruktor übergeben. Über die Funktion *setEntityManager* wird der Entitäten-Manager gesetzt und ein Objekt der Klasse *JPAAccess* erzeugt. Diesem Objekt kann das externe EAO *KundeEAO* übergeben werden.

```
public class DBEdit extends Session {

    public IStorage getKunde() throws Exception {

        JPAStorage jpaKunde = (JPAStorage)get("kunde");

        if (jpaKunde == null) {

            jpaKunde = new JPAStorage(Kunde.class);
            jpaKunde.setEntityManager(getEntityManager());

            KundeEAO kundeEAO = new KundeEAO();
            kundeEAO.setEntityManager(getEntityManager());

            JPAAccess jpaAccess = jpaKunde.getJPAAccess();
            jpaAccess.setExternalEAO(kundeEAO);
            jpaKunde.open();

            put("kunde", jpaKunde);
        }
```

```
        return jpaKunde;
    }
}
```

Listing 6.10: Funktionen für den Zugriff auf die JPAStorage Kunde

Listing 6.11 zeigt den Zugriff auf die Entität *Adresse* über die Klasse *JPAStorage*. Da für diesen Zugriff kein eigenes EAO verwendet wird, genügt es, nur den Entitäten-Manager zu setzen.

```
public class DBEdit extends Session {
    ...
    public IStorage getAdresse() throws Exception {

        JPAStorage jpaAdresse = (JPAStorage)get("adresse");

        if (jpaAdresse == null) {

            jpaAdresse = new JPAStorage(Adresse.class);
            jpaAdresse.setEntityManager(getEntityManager());
            jpaAdresse.open();

            put("adresse", jpaAdresse);
        }

        return jpaAdresse;
    }
    ...
}
```

Listing 6.11: Funktionen für den Zugriff auf die JPAStorage Adresse

Listing 6.12 zeigt den Zugriff auf die Beziehung zwischen der Entität *Kunde* und der Entität *Ausbildung*. Diese stellt eine Viele-zu-viele Beziehung dar. In der Datenbank wird diese Viele-zu-viele Beziehung in Form einer Zwischentabelle abgebildet. Bei den Entitäten wird nicht zwingend eine Zwischen-Entität benötigt. In unserem Beispiel gibt es diese Zwischen-Entität nicht und daher muss die Beziehung durch Angabe beider Entitäten angegeben werden. Die Master-Entität *Kunde* wird ganz normal im Konstruktor übergeben. Die zweite Entität wird über die Funktion *setDetailEntity* von *JPAStorage* gesetzt. Dadurch ist es auf einfache Art und Weise möglich, eine Viele-zu-viele Beziehung in einem *JPAStorage* Objekt abzubilden.

```
public class DBEdit extends Session {
    ...
    public IStorage getKundeAusbildung() throws Exception {

        JPAStorage jpaKundeAusbildung = (JPAStorage)get("kundeAusbildung");

        if (jpaKundeAusbildung == null) {

            jpaKundeAusbildung = new JPAStorage(Kunde.class);
            jpaKundeAusbildung.setDetailEntity(Ausbildung.class);
            jpaKundeAusbildung.setEntityManager(getEntityManager());
            jpaKundeAusbildung.open();

            put("bestellungArtikel", jpaKundeAusbildung);
        }

        return jpaKundeAusbildung;
```

```
    }
}
```

Listing 6.12: Funktionen für den Zugriff auf die JPAStorage KundeAusbildung

6.6 Evaluierung der Qualitätskriterien zur Integration von JPA in JVx

Nachfolgend werden die in Abschnitt 5.2 angeführten Qualitätskriterien evaluiert und es wird geprüft ob diese durch die Integration von JPA in JVx erfüllt wurden.

Kriterium	Evaluierung
Unabhängigkeit des JDBC Treibers	Die Persistenzeinheit für den datenbankunabhängigen Zugriff kann im META-INF Verzeichnis einer JVx Anwendung angegeben werden. Siehe dazu Abschnitt 6.3. Über diese Persistenzeinheit ist es möglich, den Entitäten-Manager für den Zugriff und die Verwaltung der Entitäten zu erzeugen. Durch Austausch der Persistenzeinheit kann auf einfache Art und Weise auf einen anderen Datenbankanbieter gewechselt werden.
Datenbankunabhängigkeit der Abfragesprache	Über den Entitäten-Manager können innerhalb einer JVx Anwendung datenbankunabhängige SQL Abfragen erzeugt werden. Enthält eine JVx Anwendung in der Geschäftslogik spezielle SQL Ausdrücke, so müssen diese bei einem Wechsel zu einem anderen Datenbankanbieter nicht angepasst werden, sofern dieser Datenbankanbieter vom verwendeten Persistenzanbieter korrekt unterstützt wird.
Optimierte Performance	Verwendet der Entwickler die Cachingmechanismen des jeweiligen Persistenzanbieters korrekt, so kann dadurch ein Performancegewinn erzielte werden. JVx bietet durch eigene Lazy-Loading Mechanismen und Cachen der Daten bereits eine gute Performance. Durch eine Kombination und der korrekten Verwendung der Cachingmechanismen von JPA und JVx wäre eine Optimierung der Performance denkbar.
Erzeugung und Änderung des Domänenmodelles aus den Entitäten	Bietet der gewählte Persistenzanbieter die Möglichkeit, die Datenbank und die dazugehörigen Tabellen aus den Entitäten zu erzeugen bzw. zu ändern, ist dieses Kriterium erfüllt. In der Beispielanwendung wurde dies über den Persistenzanbieter EclipseLink erreicht. Siehe dazu Abschnitt 6.3.

Tabelle 6.1: Evaluierung der Qualitätskriterien zur Integration von JPA in JVx

Kriterium	Evaluierung
Beibehalten der Architektur einer JVx Anwendung	Kommt JPA in einer JVx Anwendung zum Einsatz, so ergeben sich keine Änderungen in der Client-Schicht. In der Server-Schicht müssen nur die in Kapitel 6 angeführten Entwicklungsschritte durchgeführt werden. Diese haben aber keine Auswirkung auf die definierte Architektur einer JVx Anwendung. Es müssen nur die passenden »Storage Objekte« erzeugt werden.
Unterstützung der bestehenden JVx Funktionalitäten	JPA wurde so in JVx integriert, dass die bereits vorhandenen Funktionalitäten wie die automatische Verlinkung und die Master-Detail-Beziehung wie gewohnt zum Einsatz kommen bzw. verwendet werden können.
Unterstützung verschiedenster Persistenzanbieter	Die Integration von JPA in JVx wurde so vorgenommen, dass innerhalb des Frameworks nur Klassen, Schnittstellen und Annotationen des JPA Standards verwendet wurden. Dadurch ist es möglich, jeden Persistenzanbieter einzusetzen, wenn dieser auf dem Standard von JPA 2.0 aufbaut.
Datenbankunabhängige Erzeugung der Metadaten	In der Klasse *JPAStorage* werden die Metadaten anhand der Entitäten und deren Beziehungen zueinander erzeugt. Dadurch ist eine datenbankunabhängige Erzeugung der Metadaten gewährleistet.
Einfache Verwendung von JPA in JVx	Wie aus den in Kapitel 6 angeführten Entwicklungsschritten zu erkennen ist, ist die Verwendung von JPA in JVx mit wenig Quellcode und auf einfache Weise möglich.
Verwendung von eigenen DAOs bzw. EAOs	JVx ermöglicht die Verwendung von eigenen DAOs bzw. EAOs, um den Zugriff auf die Entitäten zu steuern. Diese Eingliederung der eigenen DAOs bzw. EAOs in die JVx Anwendung ist in Abschnitt 5.3.3 erklärt.

Tabelle 6.2: Evaluierung der Qualitätskriterien zur Integration von JPA in JVx

7 Zusammenfassung und Ausblick

Das quelloffene Framework JVx verbindet bekannte Architekturmodelle und Technologien mit neuen Ideen und Eigenschaften. Für einen reinen Anwendungsentwickler bietet JVx eine gute Dokumentation und es ermöglicht benötigte Funktionalitäten mit wenig Quellcode in kurzer Zeit zu entwickeln. Meines Erachtens handelt es sich um ein Framework, welches von einem Softwareentwickler in wenigen Tagen erlernt und angewendet werden kann, um tolle Java Anwendungen zu erstellen.

Aber auch im Bezug auf Weiterentwicklungen, welche das Framework direkt betreffen, wurde großer Wert auf Offenheit gelegt. Eine übersichtliche Architektur und offen gehaltene Schnittstellen ermöglichen die Integration bzw. Erweiterung von Funktionalitäten und auch die Eingliederung des Frameworks in bestehende Anwendungen. Ausreichend kommentierter und sauberer Quellcode verhilft uns dazu, die interne Funktionsweise zu verstehen und gegebenenfalls zu erweitern.

In dieser Arbeit wurde die Persistenz Schnittstelle von JVx mit dem Java Persistence API (JPA) 2.0 von Java EE 6 kombiniert. In der aktuellen JVx Version gab es bis jetzt nur eine Anbindung der Persistenz Schnittstelle an JDBC. JDBC bietet jedoch keine Datenbankunabhängigkeit und muss daher für jeden Datenbankanbieter speziell verwendet werden. Um eine gewisse Datenbankunabhängigkeit auch in JVx zu gewährleisten, werden die bekanntesten Anbieter (DB2, Derby, HSQLDB, MSSQL, MySql, Oracle und PostgreSQL) von JVx unterstützt. Soll ein anderer Datenbankanbieter in JVx eingesetzt werden und ist dieser nicht ANSI konform, so ist womöglich ein großer Programmieraufwand notwendig, um diesen Datenbankanbieter zur Verfügung zu stellen. Vor allem auch deshalb, weil JVx den Aufbau einer JVx Anwendermaske über die zugrundeliegenden Metadaten einer Datenbanktabelle regelt. Diese Metadaten sind aber datenbankanbieterspezifisch zu ermitteln. JPA bzw. Persistenzanbieter von JPA bietet diese Datenbankunabhängigkeit bzw. unterstützt mehr Datenbankanbieter als die zuvor erwähnten. Die Ermittlung der Metadaten erfolgt dann nicht aufbauend auf den datenbankanbieterspezifischen Tabellen, sondern den zugrundeliegenden Entitäten von JPA.

Damit JPA in JVx überhaupt verwendet werden kann, mussten 10 Qualitätskriterien (siehe Abschnitt 5.2) erfüllt werden. Durch die Integration sollten die bestehende Architektur und auch die bestehende Funktionalität von JVx weiterhin aufrecht erhalten bleiben. Auch die einfache Verwendung von JPA in JVx soll gewährleistet werden, um das Interesse für den Gebrauch von JPA in einer JVx Anwendung zu erzeugen. Die Qualitätskriterien wurden in Abschnitt 6.6 evaluiert. Diese Evaluierung ergab, dass alle 10 Qualitätskriterien zur Gänze erfüllt werden.

Die Schwierigkeiten bei der Entwicklung lagen vor allem auf einer einfachen und sauberen Abbildung der Entitäten in JVx, sodass vorhandene Funktionalitäten (automatische Verlinkung und Master-Detail-Beziehung) nicht gefährdet waren. Aber auch die Möglichkeit der Abbildung einer Viele-zu-viele Beziehung zwischen zwei Entitäten in JVx war mit Schwierigkeiten verbunden.

In der Datenbank wird eine Viele-zu-viele Beziehung mit einer Zwischentabelle gelöst. In JPA muss es diese Zwischen-Entität jedoch nicht unbedingt geben. Auch die unterschiedlichen Möglichkeiten für die Primärschlüssel und die sich daraus ergebenden Fremdschlüssel bei Beziehungen mussten berücksichtigt werden.

Die Integration wurde so umgesetzt, dass diese angeführten Schwierigkeiten gelöst wurden und der Einsatz von JPA in JVx somit ermöglicht wird. Durch die Verwendung von JPA können bekannte Vorteile dieses Standards auch in JVx genutzt werden. Dem Einsatz von anderen Java EE Frameworks in einer JVx Anwendung sind dadurch keine Grenzen gesetzt. Java Frameworks wie JasperReport [61] können aufgrund dieser Integration nun problemlos verwendet werden. Für Webanwendungen, welche auf dem Java EE Standard basieren, bietet JVx durch die Integration eine einfache Möglichkeit, Backendanwendungen zu entwickeln, ohne das zugrundeliegende Domänenmodell anzupassen. Die Integration von JPA in JVx bietet also im Grunde für beide Seiten, JVx und Java EE Anwendungen, einen Mehrwert.

Da es sich bei JVx um ein quelloffenes Framework handelt sind der Erweiterung keine Grenzen gesetzt. Die Integration von anderen Java Standards wie den Webservice Technologien JAX-RS oder JAX-WS und die Kopplung an das Persistenz API von JVx wären sinnvolle Erweiterungen. Aber auch die Verwendung von EJBs in der Server-Schicht bzw. der Einsatz des unterstützenden Frameworks Spring [62] innerhalb einer JVx Anwendung wären interessante Ansatzpunkte. Der umgekehrte Weg, also die Integration der Persistenz Schnittstelle von JVx in einer Java EE Anwendung, ist natürlich auch eine Möglichkeit.

Literatur

[1] J. Gosling, B. Joy, G. Stelle, G. Bracha, and A. Buckley, *The Java Language Specification*, Oracle Corporation Std., Juli 2011. [Online]. Available: http://docs.oracle.com/javase/specs/jls/se7/jls7.pdf

[2] R. Janas and W. Zabierowski, "Brief overview of jee," in *Modern Problems of Radio Engineering*, Telecommunications and . I. C. o. Computer Science (TCSET), Eds., 2010, pp. 174 – 176.

[3] I. Evans, *Your First Cup: An Introduction to the Java EE Platform*, Oracle Corporation, 500 Oracle Parkway Rewood City, CA 94065 U.S.A, Juni 2011. [Online]. Available: http://docs.oracle.com/javaee/6/firstcup/doc/

[4] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java Virtual Machine Specification*, Oracle Corporation Std., Juni 2011. [Online]. Available: http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

[5] D. Abts, *Masterkurs Client/Server-Programmierung mit Java*. Vieweg, 2007, no. 978-3-8348-0322-1.

[6] E. Jendrock, I. Evans, D. Gollapudi, K. Haase, and C. Srivathsa, *The Java EE 6 Tutorial*, Oracle Corporation, 500 Oracle Parkway Rewood City, CA 94065 U.S.A, Juli 2011. [Online]. Available: http://docs.oracle.com/javaee/6/tutorial/doc/javaeetutorial6.pdf

[7] B. Martin, *JDBC Developer's Guide and Reference*, Oracle Corporation Std., Maerz 2010. [Online]. Available: http://docs.oracle.com/cd/B19306_01/java.102/b14355.pdf

[8] W. Farrel, *Introduction to the J2EE Connector Architecture*, IBM. [Online]. Available: http://carfield.com.hk/document/java/tutorial/java_connector_architecture.pdf

[9] S. Cheung and V. Matena, *Java Transaction API (JTA)*, Sun Microsoftsystems Inc. Std., November 2002. [Online]. Available: http://download.oracle.com/otn-pub/jcp/jta-1.1-spec-oth-JSpec/jta-1_1-spec.pdf

[10] R. Mordani, *Java Servlet Specification*, Sun Microsystems Std., Dezember 2010. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr315/index.html

[11] R. K. Ed Burns, *Java Server Faces Speci*, Oracle Corporation Std., November 2010. [Online]. Available: http://download.oracle.com/otndocs/jcp/jsf-2.0-fr-eval-oth-JSpec/

[12] P. Delisle, J. Luehe, and M. Rot, *JavaServer Pages Specification*, Sun Microsoft Std., Mai 2006. [Online]. Available: http://jsp.java.net/spec/jsp-2_1-fr-spec.pdf

[13] L. DeMichiel and M. Keith, *EJB Core Contracts and Requirements*, Sun Microsystems Std., May 2006. [Online]. Available: http://download.oracle.com/otn-pub/jcp/ejb-3_0-fr-oth-JSpec/ejb-3_0-fr-spec-ejbcore.pdf

[14] M. Hadle and P. Sando, *JAX-RS: Java API for RESTful Web Service*, Sun Microsystem Std., September 2008. [Online]. Available: http://download.oracle.com/otn-pub/jcp/jaxrs-1.0-fr-eval-oth-JSpec/jaxrs-1.0-final-spec.pdf

[15] R. Chinnici, M. Hadley, and R. Mordan, *The Java API for XML Web Services (JAX-WS) 2.0*, Sun Microsystem Std., Oktober 2005. [Online]. Available: http://download.oracle.com/otn-pub/jcp/jaxws-2_0-pfd-spec-oth-JSpec/jaxws-2_0-pfd-spec.pdf

[16] L. DeMichiel, *JSR 317: Java Persistence API, Version 2.0*, Sun Microsystem Std., November 2009. [Online]. Available: http://jcp.org/aboutJava/communityprocess/final/jsr317/index.html

[17] Oracle. (letzter Zugriff 2012.04.27) Java persistence api. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html

[18] R. Hien and M. Kehle, *Hibernate und das Java Persistence API*. entwickler.press, 2007, no. 978-3-935042-96-3.

[19] D. Schmitt, "Techniken und werkzeuge fuer objekt-relationale abbildungen," Master's thesis, TU Wien, Juli 2006. [Online]. Available: http://club.black.co.at/david/thesis/Diplomarbeit.pdf

[20] C. Ireland, D. Bowers, M. Newton, and K. Waugh, "A classification of object-relational impedance mismatch," in *Proceedings of the 2009 First International Conference on Advances in Databases, Knowledge, and Data Applications*, ser. DBKDA '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 36–43. [Online]. Available: http://dx.doi.org/10.1109/DBKDA.2009.11

[21] Oracle, *Oracle Containers for J2EE Enterprise JavaBeans Developers Guid*, Oracle, Maerz 2007.

[22] W. Eberling and J. Lessner, *Enterprise JavaBeans 3.1*. HANSER, 2011, no. 987-3-446-42259-9.

[23] M. Keith and M. Schincariol, *Pro JPA 2, Mastering the Java Persistence API*. Apress, 2009, no. 978-1430219569.

[24] Oracle. (letzter Zugriff 2012.02.26) Toplink. [Online]. Available: http://www.oracle.com/technetwork/middleware/toplink/overview/index.html

[25] H. Pan, J. Chen, and C. Wu, "The network platform based on struts2 + jp a + spring framework," in *International Conference on Educational and Information Technology*, 2010.

[26] D. Phutela and M. Solutions, *Hibernate Vs JDBC*, Mindfire Solutions Std. [Online]. Available: http://www.mindfiresolutions.com/mindfire/Java_Hibernate_JDBC.pdf

[27] J. E. Lascano, *JPA implementations versus pure JDBC*, San Jose State University Std. [Online]. Available: http://www.espe.edu.ec/portal/files/sitiocongreso/congreso/c_computacion/PaperJPAversusJDBC_edisonlascano.pdf

[28] P. C. Linskey and M. Prud'hommeaux, "An in-depth look at the architecture of an object/relational mapper," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, ser. SIGMOD '07. New York, NY, USA: ACM, 2007, pp. 889–894. [Online]. Available: http://doi.acm.org/10.1145/1247480.1247581

[29] B. Song, Y. Zhang, and C.-S. Zhou, "Implementation on network teaching system based on java ee architecture," in *Proceedings of the 2010 Second International Conference on Information Technology and Computer Science*, ser. ITCS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 227–231. [Online]. Available: http://dx.doi.org/10.1109/ITCS.2010.62

[30] A. Donovan, A. Kiežun, M. S. Tschantz, and M. D. Ernst, "Converting java programs to use generic libraries," in *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 15–34. [Online]. Available: http://doi.acm.org/10.1145/1028976.1028979

Internetquellen

[31] Oracle, *Java Platform Standard Edition 7 Documentation*, Oracle Corporation Std., letzter Zugriff 2012.02.06. [Online]. Available: http://docs.oracle.com/javase/7/docs/index.html

[32] ——. (letzter Zugriff 2012.04.27) Java se at a glance. [Online]. Available: http://www.oracle.com/technetwork/java/javase/overview/index.html

[33] ——. (letzter Zugriff 2012.04.27) Java ee at a glance. [Online]. Available: http://www.oracle.com/technetwork/java/javaee/overview/index.html

[34] ——, *Java Platform Micro Edition Software Development Kit*, Oracle Corporation Std., April 2009. [Online]. Available: http://docs.oracle.com/javame/dev-tools/jme-sdk-3.0-win/html-helpset/index.html

[35] ——. (letzter Zugriff 2012.04.27) Java me and java card technology. [Online]. Available: http://www.oracle.com/technetwork/java/javame/index.html

[36] J. Community. (letzter Zugriff 2012.02.26) Hibernate. [Online]. Available: http://hibernate.org/

[37] E. Foundation. (letzter Zugriff 2012.02.26) Eclipselink. [Online]. Available: http://www.eclipse.org/eclipselink/

[38] A. Softw. (letzter Zugriff 2012.04.26) Openjpa. [Online]. Available: http://openjpa.apache.org/

[39] Sibvisions. (letzter Zugriff 2012.04.10) Sibvisions. [Online]. Available: http://www.sibvisions.com/

[40] ——. (2012, April) Jvx enterprise application framework. [Online]. Available: http://www.sibvisions.com/jvx

[41] Q. J. Team. (letzter Zugriff 2012.04.10) Qt jambi. [Online]. Available: http://qt-jambi.org/

[42] Google. (letzter Zugriff 2012.04.10) Google web toolkit. [Online]. Available: https://developers.google.com/web-toolkit/

[43] Microsoft. (letzter Zugriff 2012.04.10) Silverlight. [Online]. Available: http://www.microsoft.com/silverlight/

[44] S. V. GmbH. (letzter Zugriff 2012.04.26) Systemarchitektur von jvx. [Online]. Available: http://www.sibvisions.com/de/jvxmdocumentation/85-jvxsysarch

[45] Sib. (letzter Zugriff 2012.04.10) Jvx source code. [Online]. Available: http://www.sibvisions.com/de/jvxmdownload

[46] Sibvisions. (letzter Zugriff 2012.04.18) Schritt fuer schritt anleitung zur jvx applikation. [Online]. Available: http://www.sibvisions.com/de/jvxmdocumentation/86-jvxfirstappl

[47] A. Martin. (letzter Zugriff 2012.04.27) Latein woerterbuch. [Online]. Available: http://www.albertmartin.de/latein/?q=Genus&con=0

[48] D. Bolton. (letzter Zugriff 2012.04.27) Definition of generics. [Online]. Available: http://cplus.about.com/od/glossar1/g/genericsdefn.htm

[49] IBM. (letzter Zugriff 2012.04.22) Db2. [Online]. Available: http://www-01.ibm.com/software/data/db2/

[50] A. S. Foundation. (letzter Zugriff 2012.04.22) Derby. [Online]. Available: http://db.apache.org/derby/

[51] T. H. D. Group. (letzter Zugriff 2012.04.22) Hsqldb. [Online]. Available: http://hsqldb.org/

[52] Microsoft. (letzter Zugriff 2012.04.22) Mssql. [Online]. Available: http://www.microsoft.com/sqlserver/en/us/default.aspx

[53] O. Corporation. (letzter Zugriff 2012.04.22) Mysql. [Online]. Available: http://www.mysql.de/

[54] Oracle. (letzter Zugriff 2012.04.27) Oracle database. [Online]. Available: http://www.oracle.com/de/products/database/index.html

[55] P. G. D. Group. (letzter Zugriff 2012.04.22) Postgresql. [Online]. Available: http://www.postgresql.org/

[56] Oracle. (letzter Zugriff 2012.04.22) Oracle toplink 10.1.3: Database support. [Online]. Available: http://www.oracle.com/technetwork/middleware/ias/database1013-094118.html

[57] Sybase. (letzter Zugriff 2012.04.22) Sybase database. [Online]. Available: http://www.sybase.com/

[58] O. Software. (letzter Zugriff 2012.04.22) Objectdb. [Online]. Available: http://www.objectdb.com/

[59] S. Wurm. (letzter Zugriff 2012.05.11) Jvx ee source code. [Online]. Available: http://sourceforge.net/projects/jvxee/

[60] R. Hat. (letzter Zugriff 2012.04.26) Jboss application server. [Online]. Available: http://www.jboss.org/jbossas

[61] J. Soft. (letzter Zugriff 2012.04.10) Jasperreports. [Online]. Available: http://jasperforge.org/projects/jasperreports

[62] SpringSource. (letzter Zugriff 2012.04.27) Spring. [Online]. Available: http://www.springsource.org/

Abbildungsverzeichnis

2.1 Mehrschichtenarchitektur 5
2.2 Java EE Server und Container 8

3.1 Funktionsweise einer objektrelationale Abbildung 10
3.2 Lebenszyklus einer Entität 14
3.3 Komponenten zur Verwaltung der Entitäten 17

4.1 Systemarchitektur von JVx 25
4.2 Klassendiagramm zu den Metadaten von JVx 28
4.3 Klassendiagramm zu den Einschränkungskriterien von JVx 30
4.4 Datenstruktur der Beispielanwendung 32
4.5 Arbeitsoberfläche der Beispielanwendung 32
4.6 Automatische Auswahllisten in Tabellen 36
4.7 Automatische Auswahllisten in der Detailansicht 36
4.8 UIEditor für Datumseingabe 39
4.9 Serverklassen einer JVx Anwendung 39

5.1 Gegenüberstellung der JVx und Java EE Architektur 42
5.2 Gegenüberstellung einer JVx und Java EE Anwendung 43
5.3 Integration von JPA in JVx 48
5.4 Klassendiagramm zur Integration von JPA in JVx 48
5.5 Klassendiagramm der JPAStorage Klasse 49
5.6 Klassendiagramm der JPAAccess Klasse 50
5.7 Klassendiagramm des generischen EAO Designs 53
5.8 Klassendiagramm zur Klasse JPAServerMetaData 53
5.9 Abbildungsmodell von JPAServerMetaData 54

Tabellenverzeichnis

2.1 Beschreibung der Java Plattformen 4
2.2 Java EE 6 Komponenten der EIS-Schicht 6
2.3 Java EE 6 Komponenten der Web-Schicht 6
2.4 Java EE 6 Komponenten der Business-Schicht 7

3.1 Kaskadierende Operationen für Entitäten 17
3.2 Bekannte Persistenzanbieter 20
3.3 Eingrenzung einer Abfrage 23
3.4 Logische Verknüpfung von Abfragen 23

4.1 Subklassen der Klasse CompareCondition 30

5.1 Java EE 6 Komponenten der Business-Schicht 44
5.2 Funktions Typen von com.sibvisions.rad.persist.jpa.EAOMethod 52

6.1 Evaluierung der Qualitätskriterien zur Integration von JPA in JVx 61
6.2 Evaluierung der Qualitätskriterien zur Integration von JPA in JVx 62

Abkürzungsverzeichnis

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DAO	Data Access Object
EAO	Entity Access Object
EIS	Enterprise Information System
EJB	Enterprise Java Beans
ERP	Enterprise Resource Planning
HTTP	Hyper Text Transfer Protocol
J2EE	Java 2 Platform Enterprise Edition
Java EE	Java Enterprise Edition
Java ME	Java Micro Edition
Java SE	Java Standard Edition
JAX-RS	Java API for RESTful Web Services
JAX-WS	Java API for XML Web Services
JDBC	Java Database Connectivity
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSF	Java Server Faces
JSP	Java Server Pages
JSS	Java Servlet API
JTA	Java Transaction API
JVM	Java Virtual Maschine
JVx	Java Application Extention
POJO	Plain Old Java Object
TCP/IP	Transmission Control Protocol/Internet Protocol