entwickler.press

Java EE 6

Enterprise-Anwendungsentwicklung leicht gemacht

Dirk Weil

Dirk Weil

Java EE 6

Enterprise-Anwendungsentwicklung leicht gemacht

entwickler.press

Dirk Weil
Java EE 6
Enterprise-Anwendungsentwicklung leicht gemacht
ISBN: 978-3-86802-077-9

© 2012 entwickler.press
Ein Imprint der Software & Support Media GmbH

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de abrufbar.

Ihr Kontakt zum Verlag und Lektorat:
Software & Support Media GmbH
entwickler.press
Geleitsstr. 14
60599 Frankfurt am Main
Tel.: +49 (0)69 630089-0
Fax: +49 (0)69 930089-89
lektorat@entwickler-press.de
http://www.entwickler-press.de

Lektorat: Sebastian Burkart
Korrektorat: Frauke Pesch
Satz: Dominique Kalbassi

Alle Rechte, auch für Übersetzungen, sind vorbehalten. Reproduktion jeglicher Art (Fotokopie, Nachdruck, Mikrofilm, Erfassung auf elektronischen Datenträgern oder anderen Verfahren) nur mit schriftlicher Genehmigung des Verlags. Jegliche Haftung für die Richtigkeit des gesamten Werks kann, trotz sorgfältiger Prüfung durch Autor und Verlag, nicht übernommen werden. Die im Buch genannten Produkte, Warenzeichen und Firmennamen sind in der Regel durch deren Inhaber geschützt.

Inhaltsverzeichnis

Vorwort **9**

1 Java EE im Überblick **11**

1.1 Aufgabenstellung 11
1.2 Architekturmodell 11
1.3 Anwendungsbestandteile und Formate 12
1.4 Profile 15
1.5 Plattformen 16

2 CDI **17**

2.1 Was ist das? 17
2.2 Wozu braucht man das? 17
2.3 Bereitstellung und Injektion von Beans 20
2.4 Lifecycle Callbacks 25
2.5 Qualifier 26
2.6 Alternatives 28
2.7 Nutzung der Java-EE-Umgebung 30
2.8 Producer und Disposer 31
2.9 Kontexte und Scopes 35
2.10 Interceptors 40
2.11 Decorators 43
2.12 Stereotypes 45
2.13 Eventverarbeitung 47
2.14 Programmgesteuerter Zugriff auf CDI Beans 49
2.15 Integration von JPA, EJB und JSF 52
2.16 Plattformen und Ergänzungen 53

3 Java Persistence **57**

3.1 Worum geht's? 57
3.1.1 Lösungsansätze 58
3.1.2 Anforderungen an O/R-Mapper 59
3.1.3 Entwicklung des Standards 60
3.1.4 Architektur von Anwendungen auf Basis von JPA 61

3.2 Die Basics 62
3.2.1 Entity-Klassen 62
3.2.2 Konfiguration der Persistence Unit 64
3.2.3 CRUD 66
3.2.4 Detached Objects 68
3.2.5 Entity-Lebenszyklus 69
3.2.6 Mapping-Annotationen für einfache Objekte 70
3.2.7 Generierte IDs 76
3.2.8 Objektgleichheit 79
3.3 Objektrelationen 82
3.3.1 Unidirektionale n:1-Relationen 83
3.3.2 Unidirektionale 1:n-Relationen 86
3.3.3 Bidirektionale 1:n-Relationen 87
3.3.4 Uni- und bidirektionale 1:1-Relationen 91
3.3.5 Uni- und bidirektionale n:m-Relationen 93
3.3.6 Eager und Lazy Loading 94
3.3.7 Kaskadieren 96
3.3.8 Orphan Removal 98
3.3.9 Anordnung von Relationselementen 99
3.4 Queries 100
3.4.1 JPQL 100
3.4.2 Native Queries 112
3.4.3 Criteria Queries 114
3.5 Vererbungsbeziehungen 122
3.5.1 Mapping-Strategie „SINGLE_TABLE“ 123
3.5.2 Mapping-Strategie „TABLE_PER_CLASS“ 125
3.5.3 Mapping-Strategie „JOINED“ 126
3.5.4 Non-Entity-Basisklassen 127
3.6 Dies und das 128
3.6.1 Secondary Tables 128
3.6.2 Zusammengesetzte IDs 129
3.6.3 Dependent IDs 130
3.6.4 Locking 133
3.6.5 Callback-Methoden und Listener 137
3.6.6 Bulk Update/Delete 139
3.7 Caching 140

3.8 Erweiterte Entity Manager 143
3.9 Java Persistence in SE-Anwendungen 149
3.9.1 Konfiguration der Persistence Unit im SE-Umfeld 149
3.9.2 Erzeugung eines Entity Managers in SE-Anwendungen 150
3.9.3 Transaktionssteuerung in Java-SE-Anwendungen 152

4 Bean Validation **153**

4.1 Aufgabenstellung 153
4.2 Plattformen und benötigte Bibliotheken 154
4.3 Validation Constraints 155
4.4 Objektprüfung 161
4.5 Internationalisierung der Validierungsmeldungen 162
4.6 Validierungsgruppen 163
4.7 Integration in JPA und JSF 164
4.8 Bean Validation in SE-Umgebungen 166

5 JavaServer Faces **167**

5.1 Einsatzzweck von JSF 167
5.2 Die Basis: Java-Webanwendungen 167
5.2.1 Grundlegender Aufbau 167
5.2.2 Servlets 168
5.2.3 JavaServer Pages 170
5.3 JSF im Überblick 171
5.3.1 Model View Controller 171
5.3.2 Facelets 173
5.3.3 Request-Verarbeitung 173
5.4 Konfiguration der Webanwendung 175
5.5 Benötigte Bibliotheken und Plattformen 177
5.6 Programmierung der Views 177
5.6.1 JSF Tag Libraries 178
5.7 Managed Beans 184
5.8 Unified Expression Language 186
5.9 Navigation 191
5.10 Scopes 193
5.11 Verarbeitung tabellarischer Daten 194
5.12 Internationalisierung 196

5.13 Ressourcenverwaltung 199
5.14 GET Support 201
5.15 Event-Verarbeitung 203
5.16 Konvertierung 206
5.17 Validierung 210
5.18 Immediate-Komponenten 218
5.19 Ajax 218
5.20 Templating mit Facelets 222
5.21 Eigene JSF-Komponenten 226
5.22 Komponentenbibliotheken 233
5.23 Security 234

6 Enterprise JavaBeans 237

6.1 Aufgabenstellung 237
6.2 Aufbau von Enterprise JavaBeans 237
6.3 EJB Deployment 240
6.4 Lokaler Zugriff auf Session Beans 241
6.5 Remote-Zugriff 243
6.6 Transaktionssteuerung 246
6.7 Asynchrone Methoden 249
6.8 Timer 250
6.9 Security 252

7 Ein „Real World"-Projekt 255

7.1 Aufgabenstellung 255
7.2 Anwendungsarchitektur 257
7.3 Persistenz 259
7.4 Views 270
7.5 Fachliche Injektion 274

Stichwortverzeichnis 277

Vorwort

Java steht uns als leistungsfähige Basis für die Softwareentwicklung schon seit mehr als 15 Jahren zur Verfügung, einen großen Teil davon auch inklusive der Enterprise Edition, d. h. der Plattform für Unternehmensanwendungen, die verteilte Systeme mit oder ohne Browser als Client, (Web-) Services, Clustering, integrierte Systemlandschaften etc. beherrschbar machen soll. Warum dann jetzt ein Buch über die Softwareentwicklung mit Java EE?

Java EE ist zweifellos schon seit vielen Jahren eine verlässliche und tragfähige Plattform zur Entwicklung von Enterprise-Anwendungen. Die hohe Komplexität der ersten Versionen hat aber viele bewogen, sich ganz oder teilweise zugunsten anderer Frameworks abzuwenden. Die Kritik war bis zur Version 1.4 auch durchaus berechtigt – nun aber sind die Weichen neu gestellt in Richtung Einfachheit der Softwareentwicklung.

Ein Wert von Java EE besteht sicher in ihrer abwärtskompatiblen Weiterentwicklung. Für ältere Versionen erstellte Anwendungen bleiben also weiter lauffähig. Als Kehrseite der Medaille sind somit auch die alten Strukturen der Plattform noch vorhanden. Um in den Genuss der leichtgewichtigen Softwareentwicklung zu kommen, muss man sich somit auf die neuen Wege konzentrieren und alten Ballast rechts und links liegen lassen.

Dieses Buch soll Ihnen bei der Auswahl aus dem großen Angebot der Java EE eine Hilfe sein. Es hat nicht den Anspruch einer allumfassenden Darstellung, sondern beschränkt sich auf die Teile der Gesamtspezifikation, mit denen sich ein leistungsfähiger, aber überschaubarer Stack für Enterprise-Anwendungen zusammensetzen lässt. Es zeigt anhand vieler Beispiele, wie einfach Software für die Java-EE-Plattform erstellt werden kann. In einem durchgängigen „Real World"-Projekt werden am Ende alle behandelten Teile zu einer kompletten Anwendung zusammengesetzt und so das Zusammenspiel der Einzelteile verdeutlicht.

Begleitprojekte

Die im Buch gezeigten Beispiele stammen aus den Begleitprojekten zu den jeweiligen Buchkapiteln. Sie stehen unter dem URL *http://www.gedoplan.de/veroeffentlichungen/javaee6buch* zum Download bereit. Dort ist auch das erwähnte übergreifende Projekt enthalten.

Weiterführende Dokumentation

Im Sinne der Auswahl der zur leichtgewichtigen Softwareentwicklung benötigten Teile der Java EE geht das Buch nicht auf sämtliche Details der Plattformbestandteile ein, son-

dern verweist für weniger häufig genutzte Informationen auf die jeweiligen Spezifikationen. Sie können über *http://jcp.org* auf diese Texte zugreifen. Am Anfang der Buchkapitel finden Sie jeweils einen entsprechenden Link.

Wir haben weitgehend darauf verzichtet, die Java-Dokumentation der Plattformklassen im Buch abzudrucken, da sie online verfügbar ist. Sie kann unter *http://docs.oracle.com/javaee/6/api/* direkt gelesen werden und ist unter *http://www.oracle.com/technetwork/java/javaee/documentation/* downloadbar.

1 Java EE im Überblick

1.1 Aufgabenstellung

Java EE steht für „Java Platform, Enterprise Edition“ und ist eine Dachspezifikation mit gut drei Dutzend einzelnen Spezifikationen, die als Ergänzung der Standard Edition für unterschiedliche Bereich der Anwendungsentwicklung dienen. Im Kern geht es um verteilte Anwendungen, die ggf. auf mehreren Servern laufen, von Clients unterschiedlichen Typs genutzt werden und mit anderen Anwendungen im Unternehmensverbund vernetzt sind. Einige Teilspezifikationen sind aber auch im reinen Standard-Umfeld anwendbar.

Die Spezifikationen legen die Schnittstellen und Umgebungseigenschaften sehr strikt fest und definieren zudem eine Menge standardisierter Infrastrukturdienste wie bspw. Transaktionssteuerung oder Komponentenregistrierung. Entwickler erhalten dadurch einen vorgegebenen Rahmen, in dem sie Software erstellen können, ohne stets die komplexe Laufzeitumgebung vor Augen zu haben. Sie können sich idealerweise auf das konzentrieren, was ihre eigentliche Aufgabe ist: Fachlogik implementieren.

1.2 Architekturmodell

Java-EE-Anwendungen lassen sich grob aufteilen in eine Clientschicht, die serverseitige Implementierung und eine Schicht, in der Daten abgelegt oder von anderen Systemen weiterverarbeitet werden.

Als Clients von Java-EE-Anwendungen werden häufig Browser eingesetzt. Die Präsentationslogik wird in diesem Fall serverseitig bereitgestellt, vielfach durch JavaServer Faces in einem Web Container.

Desktopclients implementieren die Präsentationslogik i. d. R. selbst. Sie greifen entweder direkt auf die Geschäftslogik zu oder bedienen sich bspw. Web Services.

Die Geschäftslogik kann mittels CDI und/oder EJB programmiert werden und diverse weitere serverseitige Techniken wie Java Persistence, Connector, Messaging etc. verwenden.

Die verarbeiteten Daten werden häufig in Datenbanken abgelegt, können aber auch in anderen Backend-Systemen liegen, die auf unterschiedliche Weise angebunden werden können.

Abbildung 1.1 zeigt dieses Modell schematisch. Es ist aus Gründen der Übersichtlichkeit etwas vereinfacht:

- Neben Browser und Desktop sind andere Clients denkbar wie bspw. mobile Geräte
- In der Webschicht sind andere Framework als JavaServer Faces einsetzbar
- Aus der Geschäftslogikschicht ließen sich die eher technische Aspekte wie Connector, Messaging etc. in eine Infrastrukturschicht extrahieren

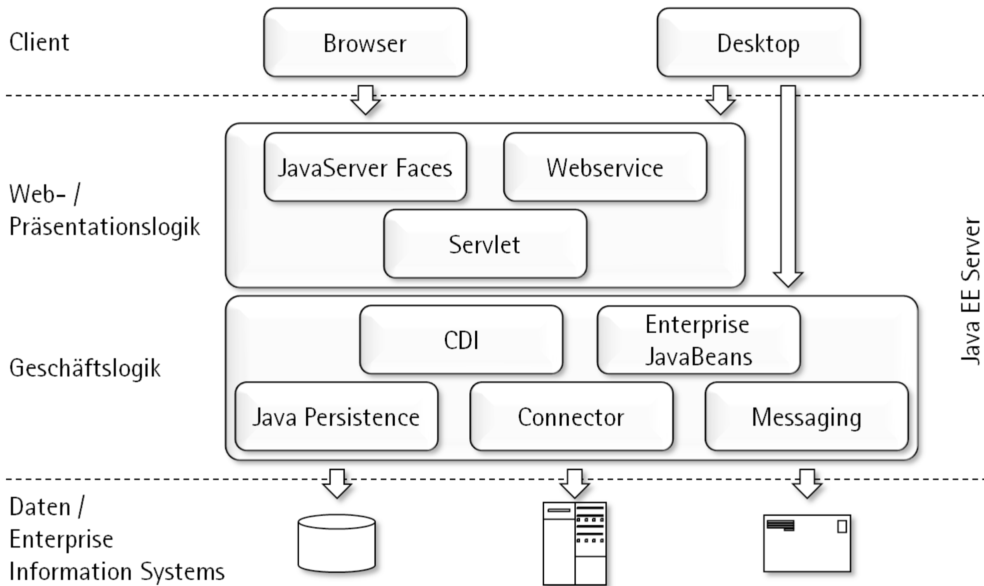


Abbildung 1.1: Java-EE-Architekturmodell

1.3 Anwendungsbestandteile und Formate

Java-EE-Anwendungen sind aus wohldefinierten Teilen aufgebaut, die sich grob an den Architekturbestandteilen orientieren. Technisch werden die Teile in separaten Jar-Dateien bereitgestellt.

Da in den weiteren Kapiteln des Buches näher auf die Bestandteile eingegangen wird, soll hier zunächst nur ein grober Überblick geschaffen werden.

Webanwendung

Dieser Anwendungsteil fasst die Webdokumente, Servlets etc. zusammen, die für die serverseitige Präsentationslogik der Anwendung benötigt werden, und zwar in ein Jar-File mit der Endung *.war*. Ein optionaler Deployment Descriptor *WEB-INF/web.xml* erlaubt die Konfiguration der Anwendung.

Enterprise JavaBeans

Im ursprünglichen Ansatz der Java EE waren EJBs der Platz für die serverseitige Geschäftslogik. Die Klassen und Interfaces mehrerer EJBs werden dann zu einem Jar-File zusammengepackt. Auch hier können Konfigurationswerte in einem Descriptor *META-INF/ejb-jar.xml* beigefügt werden. Die Endung *.jar* lässt leider keine direkte Unterscheidung von einer einfachen Bibliothek zu.

CDI

Die Aufgaben von EJBs können weitgehend von CDI übernommen werden. Java EE definiert dafür kein spezielles Anwendungsformat. Vielmehr werden die CDI-Klassen und Interfaces direkt oder als einfache Bibliothek in die gesamte Anwendung integriert. Hier ist ein Descriptor namens *beans.xml* notwendig.

Enterprise-Anwendung

Hierin werden die zuvor genannten Bestandteile nochmals zusammengefasst, und zwar zu einem Jar-File mit der Endung *.ear*. Zusätzlich können auch Konnektoren und Application Clients integriert werden, worauf hier aber nicht weiter eingegangen werden soll. Konfigurationsparameter dürfen in *META-INF/application.xml* eingetragen werden. In einem speziellen Verzeichnis – üblicherweise *lib* genannt – können Bibliotheken eingetragen werden, die der gesamten Anwendung zur Verfügung stehen sollen.

Abbildung 1.2 zeigt schematisch den Aufbau einer Enterprise Application, wobei die Inhalte der Applikationsteile – ihrerseits Jar-Files – exemplarisch angegeben sind.

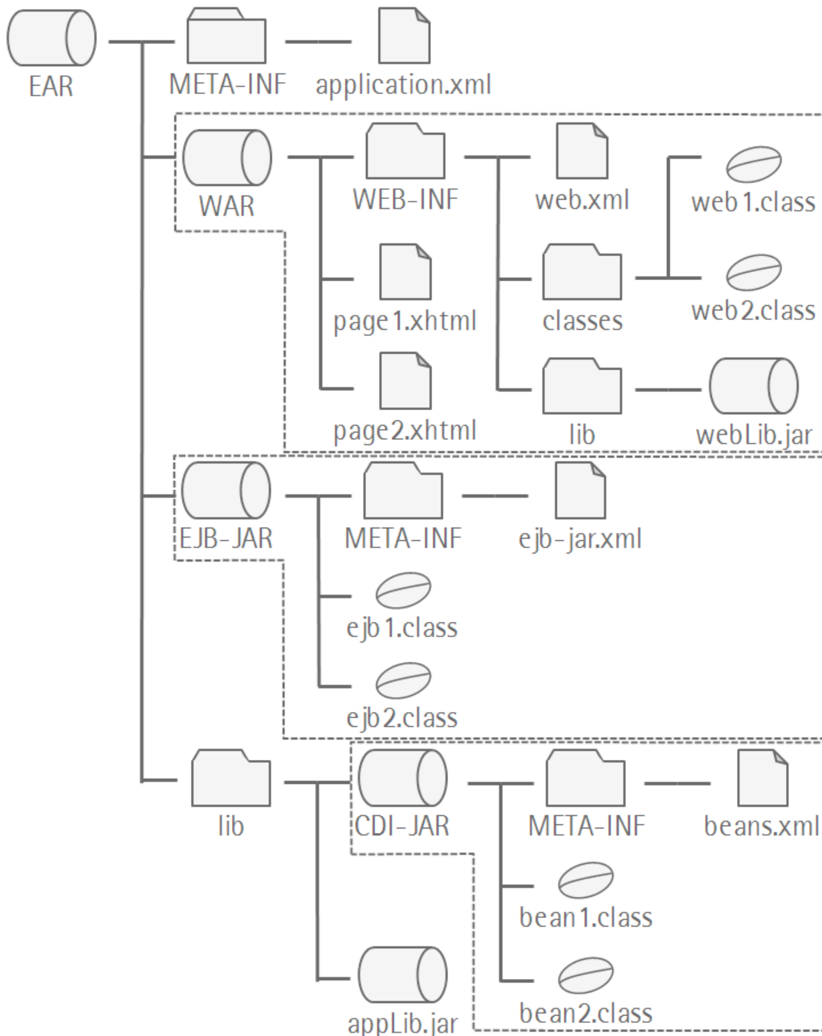


Abbildung 1.2: Aufbau einer Enterprise-Anwendung inklusive der Inhalte der Anwendungsteile

Deployment-Formate

Im Sinne der Spezifikation sind Web- und Enterprise-Anwendungen deploy-fähig. Das eigentliche Deployment-Verfahren ist abhängig vom verwendeten Application Server. Vielfach wird ein Autodeploy-Ordner angeboten, in den die *.war*- bzw. *.ear*-Dateien kopiert werden können. Alternativ oder zusätzlich bieten die Hersteller Administrationswerkzeuge an, mit denen ein Deployment von Anwendungen vorgenommen werden kann.

Einige Server erlauben auch das Deployment von Verzeichnissen. Sie haben die interne Struktur der genannten Archivdateien, sind aber eben nicht gepackt. Die gängigen Ent-

wicklungsumgebungen nutzen dies häufig für ein schnelles und inkrementelles Deployment-Verfahren direkt aus der IDE heraus.

Die Entscheidung für eines der beiden „offiziellen" Deployment-Formate ist weniger technischer, sondern eher organisatorischer Natur:

- Das EAR-Format eignet sich recht gut, wenn die Anwendungsteile klar abgegrenzt sind und sich das auch in der erstellten Dateistruktur widerspiegeln soll. Die Anwendung wird im Betrieb mit mehreren Classloadern geladen, die einander in der Art referenzieren, dass die Webanwendungen die Klassen der gesamten Anwendung nutzen können, umgekehrt aber die Klassen einer Webanwendung für andere Anwendungsteile nicht sichtbar sind. Die Abhängigkeiten der Anwendungskomponenten untereinander spiegeln sich hier also im Classloading passend wider.
- In vielen Fällen benötigt man die zwar klare, aber doch komplexe Struktur des EAR-Deployments nicht. Dann kann ein WAR-Deployment gewählt werden (entsprechend dem oberen umrahmten Teil der Abbildung 1.2). Obwohl es sich nun dem Namen nach um eine Webanwendung handelt, können auch EJBs oder CDI Beans integriert werden[1], sodass das WAR-Format keine Nachteile im Hinblick auf die verwendbaren Teile hat. Die Anwendung wird nun allerdings mit nur einem Classloader geladen, sodass zwischen den Anwendungsklassen keinerlei Isolation gegeben ist.

Die Begleitprojekte zu den Buchkapiteln sind als WAR-Deployments aufgebaut.

1.4 Profile

Beginnend mit der Version 6 der Java EE wurde eine Gruppierung der Teilspezifikationen in sog. Profile eingeführt. Damit wird der Tatsache Rechnung getragen, dass für viele Anwendungstypen nur bestimmte Teile der Java EE benötigt werden. Ein Serverhersteller ist damit nicht mehr gezwungen, für einen Java-EE-Server die gesamte Bandbreite der Spezifikation abzudecken. Er kann vielmehr einen Server anbieten, der ein bestimmtes Profil unterstützt.

Derzeit ist neben dem Gesamtumfang – dem Full Profile – nur das Web Profile definiert, das die Teile der Java EE enthält, die man i. d. R. zum Aufbau von webbasierten Anwendungen benötigt (Abb. 1.3). In der Zukunft werden sicher weitere Profile definiert werden, z. B. zur Unterstützung von Web Services oder Messaging.

1 Die Descriptoren aus *META-INF* werden im WAR-Deployment in *WEB-INF* platziert.

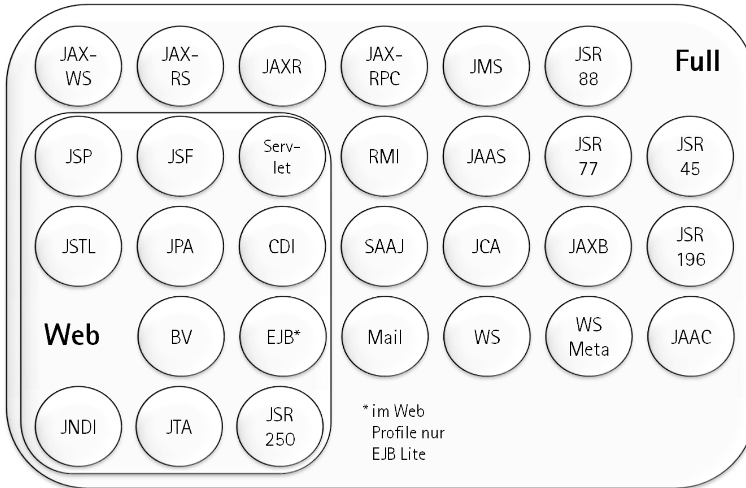


Abbildung 1.3: Web und Full Profile mit den von ihnen umfassten Teilspezifikationen[2] [3]

1.5 Plattformen

Java-EE-Anwendungen können weitgehend unabhängig vom Laufzeitsystem entwickelt werden. Für den späteren Betrieb stehen diverse Server zur Verfügung, teilweise als Open Source, teilweise kommerziell mit entsprechendem Serviceangebot Tabelle 1.1 zeigt einige der derzeit zur Verfügung stehenden Server.

Server	unterstütztes Profil
Oracle GlassFish 3	Full (Referenzimplementierung)
JBoss AS 6 und 7.0	Web
JBoss AS 7.1	Full
Caucho Resin 4	Web
Apache Tomcat + (TomEE, Siwpas)	Web
IBM WebSphere 8	Full
Oracle WebLogic 12	Full

Tabelle 1.1: Einige zur Verfügung stehende Java-EE-6-Server

2 JSR 45: Debugging Support for Other Languages
JSR 77: J2EE Management
JSR 88: Java EE Application Deployment
JSR 196: Java Authentication Service Provider Interface for Containers
JSR 250: Common Annotations for the Java Platform

3 JAXR, JAX-RPC und JSR 88 sind für die nächste Version nur noch optional geplant

2 CDI

2.1 Was ist das?

CDI steht für *Contexts and Dependency Injection for the Java EE Platform* und ist ein Standard innerhalb des Webprofils der Dachspezifikation Java EE 6. Der Arbeitsbereich von CDI ist die Bereitstellung und Verknüpfung von Komponenten und Diensten als Basis für Enterprise-Applikationen. CDI ist allerdings nicht nur im EE-Umfeld nutzbar, sondern kann auch ohne Applikationsserver eingesetzt werden.

CDI wurde im JSR 299 lange Zeit unter den Namen WebBeans entworfen und standardisiert viele Ideen und Konzepte von populären Open-Source-Frameworks wie Seam und Spring(-Core). Die Spezifikation ist für Java-EE-Verhältnisse angenehm kurz: ca. 100 gut lesbare Seiten[1].

Neben der Referenzimplementierung JBoss Weld[2] stehen u. a. Apache OpenWebBeans[3] und Resin CanDI[4] als CDI-Container zur Verfügung.

2.2 Wozu braucht man das?

Professionelle Anwendungen sind nicht monolithisch aufgebaut, sondern bestehen aus Komponenten. Zum einen ergeben sich bei der Entwicklung von Software aus der Analyse der Aufgabenstellung fachliche Bereiche, die durch fachliche Komponenten abgebildet werden können. Innerhalb dieser Komponenten lassen sich wieder Teile abgrenzen, diesmal eher technischer Natur. Die Komponenten benutzen andere Komponenten und die Plattformdienste, sind aber weitgehend abgegrenzt (Abb. 2.1).

---

1 JSR-299: Contexts and Dependency Injection for the Java EE Platform,
JSR-299 Expert Group, 10. Dezember 2009,
http://jcp.org –> Search JSR 299 –> Final Release Download –> web_beans-1_0-fr-eval-spec.pdf

2 *http://seamframework.org/Weld*

3 *http://openwebbeans.apache.org/*

4 *http://www.caucho.com/resin/candi/*

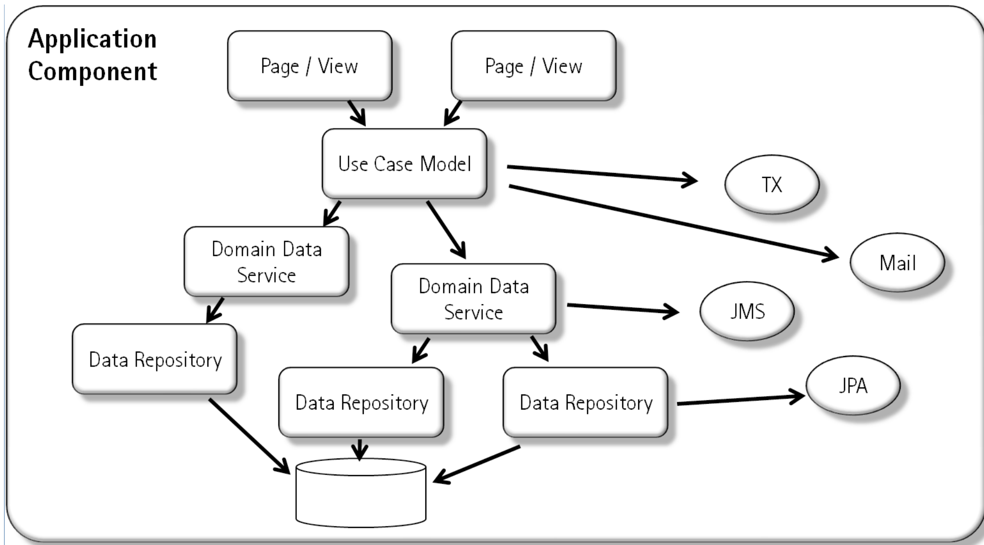


Abbildung 2.1: Anwendungskomponenten

Eine Aufgabe der Softwareentwicklung ist es nun, diese Komponenten untereinander zu verknüpfen, sodass eine saubere Anwendungsarchitektur entsteht. Das kann natürlich mit einfachen Sprachmitteln von Java geschehen. Eine Komponente kann in ihrem Programmcode andere Komponenten instanziieren, indem sie *new* benutzt. Dadurch wird die Kopplung der Komponenten aber sehr stark, denn die aufrufende Komponente muss die benutzte sehr genau kennen, Änderungen sind aufwändig, der Einsatz einer alternativen Komponente unmöglich.

Zudem profitieren solche Objekte kaum von der Umgebung der Anwendung: Der Applikationsserver „kennt“ sie nicht, kann also bspw. kein Monitoring und keine Laufzeitsteuerung dafür durchführen. Flexibler ist es, die benötigten Objekte vom Application Server herstellen zu lassen. In den früheren Versionen der Java EE – damals noch J2EE – hat man dazu weitgehend String-basierte Referenzen benutzt, hat also bspw. die benötigte Komponente per Namen im JNDI-Dienst adressiert. Hier stellt sich aber das Problem der „Zielgenauigkeit“: Ist ein Objekt unter dem verwendeten Namen überhaupt vorhanden und hat es den richtigen Typ (Listing 2.1)?

```
// Unsicher: Ist ein Objekt mit dem Namen konfiguriert?
//           Falls ja, hat es den korrekten Typ?
MyService myService
  = (MyService) jndiContext.lookup("ejb/myService");
```

Listing 2.1: Referenzierung einer Komponente über ihren Namen

Ein weiteres Problem ist die Abhängigkeit der aufrufenden Komponente von ihrer Umgebung: Der Code im Beispiel setzt unumstößlich voraus, dass es einen JNDI-Dienst gibt.

Ein Test des Codes außerhalb des Applikationsservers ist damit unmöglich. Hier setzt die Idee *Inversion of Control* an, die den aktiven Teil der Komponentenverknüpfung aus der Komponente herauslöst und in die Laufzeitumgebung – den Container – verlagert: Nicht die Komponente besorgt sich die von ihr benötigten Serviceobjekte, sondern der Container liefert sie an. Dieses Verfahren firmiert unter dem Namen *Dependency Injection* – Injektion von benötigten Objekten, womit wir auch schon die beiden letzten Drittel des Namens CDI erklärt hätten (Abb. 2.2).

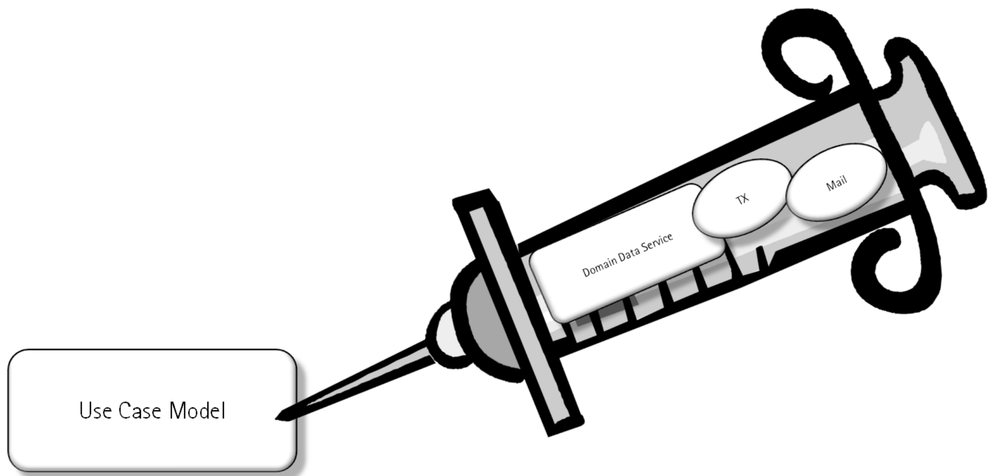


Abbildung 2.2: Dependency Injection

Die Komponente „weiß" jetzt nicht mehr, woher sie die von ihr genutzten Objekte erhält. Damit ist die Kopplung zu ihrer Umgebung so klein geworden, dass ein Austausch leicht möglich wird: Im Produktivsystem werden Komponenten und Ressourcen vom Container bspw. weiterhin im JNDI-Dienst verwaltet, während sie in einer Testumgebung ohne Container von der Testklasse geliefert werden.

Bei der Dependency Injection obliegt es dem Container, wann die benötigten Objekte erzeugt und zerstört werden, er kann also die Komponenten von der kompletten Lifecycle-Steuerung entlasten. Damit sind wir beim ersten Drittel des Namens CDI: Die injizierten Objekte können Kontexten zugeordnet werden, die über ihre Lebensdauer bestimmen. So können die von einem Geschäftsprozess genutzten Services inklusive der darin verwalteten Daten sitzungsorientiert gehalten werden.

Die geschilderten Konzepte sind beileibe nicht neu. Sie haben vielmehr seit vielen Jahren Einzug in die Java-Softwarelandschaft gehalten und dort ihren Nutzen unter Beweis gestellt – stark unterstützt insbesondere durch das Spring-Framework, das damit wesentliche Schwächen der damaligen J2EE adressierte. Neu ist allerdings ein Aspekt von CDI, der die beschriebene lose Kopplung um Typsicherheit ergänzt: Durch weitgehenden Verzicht auf Objektnamen und Verwendung von Java-Typen an ihrer Stelle wird erreicht,

dass sich die Komponentenverdrahtungen schon sehr früh – zur Compile-Zeit, spätestens zur Deployment-Zeit – prüfen lassen und somit Fehler nicht erst zur Anwendungslaufzeit zu Tage treten.

2.3 Bereitstellung und Injektion von Beans

Die durch CDI miteinander verknüpften Klassen werden in der CDI-Spezifikation *Managed Beans* genannt. Im Folgenden wird der Begriff CDI Bean bevorzugt, da Managed Beans auch in anderen Teilen der Java EE auftauchen. CDI Beans können sowohl injizierte Objekte darstellen als auch als Injektionsziel dienen.

CDI Beans

Die Anforderungen an CDI Beans sind denkbar gering: Nahezu jede konkrete Java-Klasse ist dazu geeignet[5]. Benötigt wird nur ein Konstruktor ohne Parameter (wir werden später sehen, dass auch Klassen mit anderen Konstruktoren CDI Beans sein können). Die Klasse *GreetingBean* aus Listing 2.2 ist somit als CDI Bean verwendbar.

```
public class GreetingBean
{
  public String getGreeting()
  {
    int hourOfDay = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
    if (hourOfDay < 10)
      return "Guten Morgen";
    else if (hourOfDay < 18)
      return "Guten Tag";
    else
      return "Guten Abend";
  }
}
```

Listing 2.2: Einfache CDI Bean[6]

CDI beachtet allerdings nicht alle Klassen im Classpath. Zusätzlich zu den Klassen selbst ist eine Datei namens *beans.xml* notwendig, die komplett leer sein darf. Sie muss im Verzeichnis *META-INF* eines Jar-Files oder eines Classpath-Verzeichnisses oder im Verzeichnis *WEB-INF* einer Webanwendung stehen, um die zugehörigen Klassen für CDI sichtbar zu machen. Da viele Entwicklungswerkzeuge über leere XML-Dateien meckern, sollte aber das in Listing 2.3 gezeigte wohlgeformte XML-Dokument statt der leeren Datei verwendet werden. Hierin können dann später auch einfacher Ergänzungen vorgenommen werden.

5 Implementierungen von *javax.enterprise.inject.spi.Extension* ausgenommen
6 Den in diesem Kapitel gezeigten Beispielcode finden Sie im Begleitprojekt *ee-demos-cdi*

```
<?xml version="1.0"?>
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
</beans>
```

Listing 2.3: Effektiv leerer CDI-Deskriptor „beans.xml“

Field Injection

Die Nutzung einer derart bereitgestellten Klasse in einer weiteren CDI Bean kann durch Injektion in ein Feld der Bean geschehen. Dazu wird die betroffene Instanzvariable mit *@Inject*[7] annotiert (Listing 2.4).

```
public class DemoModel
{
  @Inject
  private GreetingBean greetingBean;

  public String getHelloWorld()
  {
    return this.greetingBean.getGreeting() + ", Welt!";
  }
```

Listing 2.4: Injektion in eine Instanzvariable

Bean Type

Es fällt auf, dass zur Injektion kein Name o. ä. verwendet wird, sondern offensichtlich allein der Typ des Injektionsziels für die Zuordnung ausreichend ist. Das ist ein entscheidendes Konzept, das die Injektion nicht passend getypter Objekte verhindert. Der Typ des Injektionsziels muss mit einem Bean Type des injizierten Objekts übereinstimmen. Jede CDI-Bean hat potenziell mehrere Bean Types, nämlich die Klasse selbst, alle Basisklassen und alle direkt oder indirekt implementierten Interfaces. Die Klasse *CocktailMockRepository* in Listing 2.5 hat somit drei Bean Types: *CocktailMockRepository*, *CocktailRepository* und *Object*.

```
public class CocktailMockRepository implements CocktailRepository
{
  public void insert(Cocktail cocktail) { … }
  public Cocktail findById(String id) { … }
  …
}
```

Listing 2.5: CDI-Bean als Implementierung eines Interfaces

7 *javax.inject.Inject*

Als Injektionstyp kann somit auch eine Basisklasse oder ein Interface dienen, womit eine weitere Entkopplung der CDI Beans untereinander stattfindet. Man könnte also später bspw. den konkret injizierten Typ verändern, ohne die Injektionsstelle anpassen zu müssen (Listing 2.6).

```
public class CocktailModel
{
  @Inject
  private CocktailRepository cocktailRepository;
  …
```

Listing 2.6: Nutzung eines Interface als Injektionstyp

Die Injektion muss eindeutig auflösbar sein. Es darf also in der Anwendung nicht zwei CDI Beans geben, die den verlangten Bean Type haben. Konflikte führen spätestens beim Deployment der Anwendung im Applikationsserver zu Fehlermeldungen, werden also nicht erst zur Laufzeit bei Benutzung des entsprechenden Programmcodes erkannt. Einige Entwicklungswerkzeuge erkennen Konflikte bereits zur Entwicklungszeit und geben entsprechende Warnungen aus.

Damit ist der oben skizzierten Flexibilität bei der Wahl der konkreten Implementierung zunächst einmal ein Riegel vorgeschoben: Es kann in der Beispielanwendung nicht mehrere CDI Beans geben, die *CocktailRepository* implementieren. Wir werden später Mechanismen kennen lernen, die dies doch ermöglichen und darüber hinaus eine Auswahl der genutzten Bean erlauben (siehe Abschnitte 2.5 Qualifier und 2.7 Alternatives).

Der Bean Type einer CDI Bean kann ihr explizit zugewiesen werden. Dazu wird die Klasse mit *@Typed*[8] annotiert und als Wert der Annotation werden eine oder mehrere Typen angegeben. Die Bean hat dann nur die derart explizit genannten Bean Types (Listing 2.7).

```
@Typed(CocktailJdbcRepository.class)
public class CocktailJdbcRepository implements CocktailRepository
{


  …
```

Listing 2.7: Explizite Angabe des Bean Type

@Typed kann auch ohne Parameter verwendet werden. Eine so annotierte Klasse hat keinen Bean Type und ist somit sozusagen aus dem Spiel.

8 *javax.enterprise.inject.Typed*

Method Injection

Neben der beschriebenen Möglichkeit der Injektion in Instanzvariablen kann man bei der Erzeugung von CDI-Bean-Objekten auch Methoden aufrufen lassen. Die Spezifikation spricht hier von *Initializer Methods*. Da es aber auch Lifecycle-Methoden gibt, die im Rahmen der Initialisierung von Objekten ablaufen, wird im Folgenden der ebenfalls übliche Begriff der Methodeninjektion verwendet.

Eine CDI Bean kann beliebig viele Methoden enthalten, die mit *@Inject* annotiert sind. Sie dürfen allerdings nicht abstrakt, statisch oder generisch sein. Diese Methoden werden im Zuge der Initialisierung von CDI-Bean-Objekten aufgerufen, wobei die Methodenparameter durch Injektion mit den passenden Werten versorgt werden. Für jeden einzelnen Parameter gelten dabei die gleichen Regeln wie für die Instanzvariablen der Field Injection. Die Injektion geschieht also genau genommen nicht in die Methode, sondern in die Methodenparameter. Statt der Injektion in die Instanzvariable im Beispiel aus Listing 2.4 hätte somit auch eine Methode verwendet werden können (Listing 2.8).

```
public class DemoModel
{
  @Inject
  public void setGreetingBean(GreetingBean greetingBean)
  {
    this.greetingBean = greetingBean;
  }
  …
```

Listing 2.8: Injektion in den Parameter einer Methode

Das Beispiel zeigt den recht üblichen Fall einer *Setter Injection*, d. h. der Injektion in eine Setter-Methode. Darauf ist Method Injection aber nicht eingeschränkt: Die Methode darf einen beliebigen Namen haben und beliebig viele Parameter annehmen. Sie muss nicht *public* sein und darf einen Return-Wert liefern (der im Beispiel aber nicht verwendet wird).

Die hier besprochenen Methoden dürfen natürlich auch direkt vom Programmcode aufgerufen werden. Die Injektionsannotationen haben dann aber keinerlei Bedeutung.

Constructor Injection

Was mit einer Methode geht, kann auch für einen Konstruktor genutzt werden: Eine CDI Bean darf einen Konstruktor besitzen, der mit *@Inject* annotiert ist. Dieser wird dann statt des bisher genutzten Konstruktors ohne Parameter zur Instanziierung von CDI-Objekten verwendet. Für die Parameter des Konstruktors gelten wieder die oben erläuterten Injektionsregeln. Die Construktor Injection ist somit eine weitere Alternative zu den bisherigen Injektionsmöglichkeiten (Listing 2.9).

```
public class DemoModel
{
  private GreetingBean greetingBean;

  @Inject
  public DemoModel(GreetingBean greetingBean)
  {
    this.greetingBean = greetingBean;
  }
  …
```

Listing 2.9: Injektion in den Konstruktorparameter

Es darf immer nur ein Konstruktor mit *@Inject* ausgezeichnet sein. Er darf beliebig viele Parameter haben und muss nicht *public* sein. Gibt es keinen solchen Konstruktor, wird der parameterlose Konstruktor verwendet.

Werden die Konstruktoren direkt aufgerufen (mit *new* o. ä.), so werden die dadurch entstehenden Objekte nicht durch den CDI-Container gemanagt, d. h. es finden in ihnen keine Injektionen statt und die Lebensdauer der Objekte unterliegt nicht der Steuerung durch den Container.

Bean Name

Wir haben gesehen, dass die Zuordnung von CDI-Objekten bei der Injektion allein über ihren Bean Type geschieht, wodurch Typsicherheit garantiert wird. Man kann CDI Beans allerdings auch einen Namen zuordnen, um damit die Referenz aus einer ungetypten Umgebung heraus zu ermöglichen. So bieten bspw. JavaServer Pages und JavaServer Faces eine Expression Language an, mit der aus der textbasierten Definition einer Webseite auf benannte Java-Objekte zugegriffen werden kann.

Die Definition des Bean-Namens geschieht durch Annotation mit *@Named*[9]. Der dabei übergebene String gilt dann als Name der annotierten Bean. Wird die Annotation ohne Parameter verwendet, gilt der einfache Klassenname mit kleinem Anfangsbuchstaben als Bean Name. In Listing 2.10 ist *@Named* also äquivalent zu *@Named("demoModel")*.

```
@Named
public class DemoModel
{
  …
  public String getHelloWorld() { … }
  …
```

Listing 2.10: Definition eines Bean-Namens

Eine derart benannte CDI Bean kann mithilfe der erwähnten Expression Language aus einer Webseite auf Basis von JSP (Listing 2.11) oder JSF (Listing 2.12) referenziert werden.

9 *javax.inject.Named*

```
<%@ page language="java" … %>
<html>
<body>
  ${demoModel.helloWorld}
  …
```

Listing 2.11: Zugriff auf eine benannte CDI-Bean in JSP

```
<html xmlns="http://www.w3.org/1999/xhtml" … >
<h:body>
  <h:outputText value="#{demoModel.helloWorld}" />
  …
```

Listing 2.12: Zugriff auf eine benannte CDI-Bean in JSF

JavaServer Faces werden im gleichnamigen Buchkapitel detailliert behandelt.

2.4 Lifecycle Callbacks

Beim Ablauf des Konstruktors einer CDI Bean ist das entstehende Objekt noch nicht vollständig initialisiert – insbesondere sind die durch Field bzw. Method Injection zu befüllenden Werte noch nicht gesetzt. Insofern ist der Konstruktor für eine Objektinitialisierung nicht geeignet, wenn sie diese Werte benötigt.

Eine CDI Bean darf aber eine parameterlose Methode besitzen, die mit *@PostConstruct*[10] annotiert ist. Sie wird vom Container aufgerufen, nachdem ein Objekt erzeugt wurde und alle Injektionen durchgeführt wurden. Das ist somit der richtige Platz für Initialisierungen (Listing 2.13).

```
public class CocktailModel
{
  @PostConstruct
  public void init()
  {
    … // beliebige Initialisierungen
  }

  @PreDestroy
  public void cleanup()
  {
    … // beliebige Initialisierungen
  }
  …
```

Listing 2.13: Lifecycle-Methoden

10 *javax.annotation.PostConstruct*

Es darf nur eine mit *@PostConstruct* versehene Methode pro Klasse geben. Sie muss den Typ *void* haben und darf keine Checked Exceptions deklarieren. Die Methode muss nicht *public* sein. Sollte eine Basisklasse auch eine solche Methode vorweisen, wird sie ebenfalls vom Container aufgerufen, und zwar vor derjenigen der abgeleiteten Klasse.

Analog zu *@PostConstruct* wirkt *@PreDestroy*[11] am Ende des Lebenszyklus von Objekten: Bevor sie dem Garbage Collector überlassen werden, laufen noch die *PreDestroy*-Methoden, in denen beliebiger Code zum Aufräumen platziert werden kann.

2.5 Qualifier

Wie oben dargestellt wurde, muss die Injektion eines CDI-Objekts eindeutig auflösbar sein, d. h. bislang darf es nur genau eine CDI Bean mit passendem Bean Type geben. Das wäre in der Praxis doch zu einschränkend. CDI bietet aber die Möglichkeit an, Mehrdeutigkeiten in der Bean-Zuordnung mithilfe sog. Qualifier zu lösen. Darunter versteht man Annotationen, die ihrerseits mit *@Qualifier*[12] annotiert sind (Listing 2.14).

```
@Qualifier
@Retention(RUNTIME)
@Target({ METHOD, FIELD, PARAMETER, TYPE })
public @interface Formal
{
}
```

Listing 2.14: Ein einfacher Qualifier

Qualifier können nun bei der Definition einer CDI Bean als Annotationen verwendet werden. Eine Bean ohne selbstdefinierte Qualifier erhält automatisch den vordefinierten Qualifier *@Default*[13] zugeordnet (Listing 2.15).

```
// Implizit: @Default
public class GreetingBean
{
  …
}

@Formal
public class FormalGreetingBean extends GreetingBean
{
  …
}
```

Listing 2.15: Implizite und explizite Angabe eines Qualifiers bei der Bean-Definition

11 *javax.annotation.PreDestroy*
12 *javax.inject.Qualifier*
13 *javax.enterprise.inject.Default*

Zusätzlich bekommt jede Bean den ebenfalls vordefinierten Qualifier *@Any*. Er spielt später bei einem programmgesteuerten Zugriff auf CDI Beans eine Rolle. Hier können wir ihn zunächst einmal ignorieren.

Bei der Injektion von CDI Beans können wiederum Qualifier angegeben werden. Hier gilt *@Default* implizit als angegeben, wenn kein anderer Qualifier verwendet wird. Eine zur Injektionsstelle passende Bean muss mindestens die angegebenen Qualifier besitzen (Listing 2.16).

```
// Injektion eines FormalGreetingBean-Objektes
@Inject @Formal
private GreetingBean greetingBean;

// Injektion eines GreetingBean-Objektes
@Inject
private GreetingBean greetingBean2;

// Deployment-Fehler: Keine Bean mit *beiden* Qualifiern vorhanden
@Inject @Formal @Default
private GreetingBean greetingBean3;
```

Listing 2.16: Qualifier-Nutzung zur Auswahl injizierter Objekte

Bei Injektion in Methoden- oder Konstruktorparameter gehören die Qualifier zu den Parametern (Listing 2.17).

```
@Inject
private void setGreetingBean(@Formal GreetingBean greetingBean)
{
  …
```

Listing 2.17: Qualifier bei der Injektion in Methodenparameter

Qualifier geben uns also die Möglichkeit der Auswahl zwischen mehreren Varianten eines Dienstes o. ä., indem für jede Variante eine entsprechende Annotation bereitgestellt wird. Das kann ein wenig lästig werden, wenn nicht nur einzelne Varianten zur Verfügung gestellt werden sollen, sondern eine größere Anzahl. Für einen solchen Fall können Qualifier mit Parametern versehen werden, die bei der Auswahl der zu injizierenden Bean berücksichtigt werden. Für die Unterscheidung der Varianten bietet sich z.B. ein Aufzählungstyp an (Listing 2.18).

```
@Qualifier
@Retention(RUNTIME)
@Target({ METHOD, FIELD, PARAMETER, TYPE })
public @interface Greeting
{
  GreetingType type();
}
```

```
public enum GreetingType
{
  NORMAL, FORMAL;
}
```

Listing 2.18: Qualifier mit Parameter

An der Injektionsstelle kann dann mit dem Qualifier-Parameter gewählt werden, welche konkrete Variante verwendet werden soll (Listing 2.19).

```
@Greeting(type = GreetingType.FORMAL)
public class FormalGreetingBean extends GreetingBean
{
  …
}

public class DemoModel
{
  @Inject
  @Greeting(type = GreetingType.FORMAL)
  private GreetingBean greetingBean;
  …
}
```

Listing 2.19: Nutzung von Qualifier-Parametern zur Definition und Injektion von Beans

Sollten Sie den Bedarf haben, einem Qualifier zusätzliche Parameter mitgeben zu müssen, die für die Bean-Auswahl nicht verwendet werden sollen, so müssen Sie diese mit *@NonBinding*[14] annotieren.

Die bereits erwähnte Annotation *@Named* stellt einen weiteren vordefinierten Qualifier dar, mit dem, wie beschrieben, Expression-Language-Namen vergeben werden können. Die Verwendung von *@Named* an Injektionsstellen ist analog zu anderen Qualifiern zwar möglich, aber unüblich, da die Verwendung von Namen die beschriebene Sicherheit der Bean-Zuordnung aushebeln würde. Aus dem gleichen Grund sollten Sie übrigens bei einem selbstentwickelten Qualifier keine String-basierten Parameter verwenden.

2.6 Alternatives

Qualifier eignen sich gut zur Auswahl aus mehreren Varianten einer Bean-Implementierung, wobei diese Varianten durchaus gleichzeitig in der Anwendung zur Verfügung stehen. Unterschiedliche Anwendungsteile können somit je nach Einsatzfall die eine oder andere Variante benutzen.

14 *javax.enterprise.util.Nonbinding*

Ein anderes Szenario ist die komplett alternative Nutzung von Implementierungsvarianten. Hier sind zwar auch mehrere Implementierungen eines Dienstes o. ä. in der Anwendung vorhanden, aber nur eine davon ist aktiv. Diese Situation findet man regelmäßig im Testumfeld vor: Ein Teil der Anwendung wird zum Test durch eine andere Implementierung ausgetauscht. Unabhängig von den in diesem Zusammenhang häufig eingesetzten Mock-Frameworks bietet CDI hier *Alternatives* an.

Wird eine CDI Bean mit *@Alternative*[15] annotiert, ist sie ohne weiteres nicht für Injektionen etc. sichtbar. Damit treten auch keine Konflikte mit anderen Beans mit gleichem Bean Type auf (Listing 2.20).

```
public class CocktailJdbcRepository implements CocktailRepository
{
  …
}

@Alternative
public class CocktailMockRepository implements CocktailRepository
{
  …
}

public class CocktailModel
{
  @Inject
  private CocktailRepository cocktailRepository;
  …
```

Listing 2.20: Deklaration einer Alternative

Im Beispiel wird der Instanzvariablen *CocktailModel.cocktailRepository* ein Objekt des Typs *CocktailJdbcRepository* zugewiesen. Die Bean *CocktailMockRepository* ist durch die Annotation inaktiv.

Mithilfe des Deskriptors *beans.xml* können Alternatives nun aktiviert werden, wobei gleichzeitig die entsprechende bislang aktive Bean deaktiviert wird. Dazu muss die zu aktivierende Klasse voll qualifiziert im Element *<alternatives>* eingetragen werden (Listing 2.21).

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <alternatives>
    <class>de.gedoplan. … .CocktailMockRepository</class>
  </alternatives>
</beans>
```

Listing 2.21: Aktivierung einer Alternative durch Eintrag der Klasse in „beans.xml“

15 *javax.enterprise.inject.Alternative*

Im Beispiel würde jetzt die Bean *CocktailMockRepository* anstelle von *CocktailJdbcRepository* benutzt.

2.7 Nutzung der Java-EE-Umgebung

Bei einem Einsatz in einem Application Server können CDI Beans die vom Server bereitgestellte Umgebung nutzen. Das sind einerseits Ressourcen im Sinne der Plattform Java EE, andererseits einige Infrastrukturdienste, die der Server anbietet.

Java EE Resources

Hierunter versteht man Objekte, die im Server oder auch als Anwendungsteile zur Verfügung gestellt und konfiguriert werden, über die auf Teilfunktionalitäten wie bspw. Datenbanken oder Web Services zugegriffen werden kann. Sie können mithilfe der in der Plattform Java EE allgemein definierten Injektionsannotationen verwendet werden (Tabelle 2.1).

Annotation	Bedeutung
@Resource	Injektion von Environment Entries und Resource Manager Factories (s. Text.)
@EJB	Injektion einer EJB-Referenz: im Zusammenhang mit CDI nur für Remote EJBs nötig; lokale EJBs können einfach mit **@Inject** injiziert werden, s. Kapitel „Enterprise JavaBeans“
@PersistenceContext, *@PersistenceUnit*	Injektion, s. Kapitel „Java Persistence“
@WebServiceRef	Injektion einer Web-Service-Referenz

Tabelle 2.1: Annotationen zur Injektion von Java EE Resources

Von den genannten Annotationen verwenden wir hier nur *@Resource*[16]. Die anderen sind entweder nicht Thema dieses Buches (*@WebServiceRef*) oder werden in späteren Kapiteln wieder aufgegriffen.

@Resource kann zum Zugriff auf sog. Resource Manager Factories genutzt werden. Damit sind Objekte gemeint, mit deren Hilfe man mit Subsystemen wie Datenbanken, Messaging, Mail etc. kommunizieren kann. Sie werden entweder global im Server administriert oder innerhalb einer Anwendungskomponente konfiguriert und sind mit einem Namen im Namensdienst des Servers eingetragen. *@Resource* akzeptiert globale Namen als Parameter *lookup*. Im Beispiel (Listing 2.22) wird eine *DataSource*[17], die im Server global unter dem JNDI-Namen *"jdbc/ee_demos"* konfiguriert ist, in eine Instanzvariable injiziert.

16 *javax.annotation.Resource*
17 *javax.sql.DataSource*

```
@Resource(lookup = "jdbc/ee_demos")
private DataSource dataSource;
```

Listing 2.22: Injektion einer Datasource mithilfe von „@Resource“

Bedenken Sie, dass es sich hier um eine andere Art von Injektion handelt als wir sonst in diesem Kapitel betrachten: Der Zugriff auf das gewünschte Objekt geschieht über einen Namen, der zur Entwicklungszeit weder auf Existenz noch auf Typkonformität geprüft werden kann!

@Resource kennt neben *lookup* den Parameter *name*, mit dem der Objektname relativ zur aktuellen Anwendungskomponente angegeben werden kann. Es würde den Rahmen des Kapitels sprengen, darauf näher einzugehen. Bei Interesse finden Sie im Demoprojekt aber eine Anwendung davon (Klasse *DatabaseConnectionProducer*, Deskriptoren *web.xml*, *jboss-web.xml* und *glassfish-web.xml* – suchen Sie nach dem Namen *"jdbc/tempDb"*).

Built-in Beans

Der CDI-Container stellt einige Objekte standardmäßig als CDI Beans bereit, d. h. @Inject kann ohne weitere Vorbereitung für diese Typen genutzt werden (Tabelle 2.2).

Vordefinierter Bean Type	Bedeutung
Principal[18]	Derzeit angemeldeter Benutzer
Validator[19], *ValidationFactory*[20]	Validator für Bean Validation bzw. Factory dazu, s. Kapitel „Bean Validation“
UserTransaction[21]	Transaktionssteuerungsobjekt

Tabelle 2.2: Vordefinierte CDI Beans

2.8 Producer und Disposer

Die Erzeugung von Objekten mittels Konstruktoraufruf ist nicht immer passend, z. B. wenn die Initialisierung komplexer ist oder nicht immer der gleiche Typ geliefert werden soll. Letzteres passiert regelmäßig, wenn der gewünschte Typ gar keine Klasse, sondern ein Interface ist. In solchen Fällen greift man in Java auf Factory-Methoden zurück. Die CDI-Entsprechung dazu sind Producer Methods.

18 *javax.security.Principal*
19 *javax.validation.Validator*
20 *javax.validation.ValidationFactory*
21 *javax.transaction.UserTransaction*

Producer Methods

Eine CDI Bean kann beliebig viele Producer Methods deklarieren. Das sind nichtabstrakte Methoden – ggf. *static* – die mit *@Produces*[22] annotiert sind. Sie liefern CDI-Objekte zur Injektion in andere Objekte. Der Name der Methode ist zweitrangig, sie muss nicht *public* sein (Listing 2.23).

```
public class DatabaseConnectionProducer
{
  @Resource(lookup = "jdbc/ee_demos")
  private DataSource dataSource;

  @Produces
  public Connection createConnection() throws SQLException
  {
    return this.dataSource.getConnection();
  }
  …
```

Listing 2.23: Producer Method

Das von der Producer-Methode gelieferte Objekt kann wie gewohnt in eine CDI Bean injiziert werden, z. B. so: *@Inject private Connection dbConnection;*

Dazu wird vom Container zunächst ein Objekt der Producer-Klasse bereitgestellt, falls die Producer Method nicht *static* ist. Die Producer-Klasse ist eine normale CDI Bean, d. h. alle Mechanismen zur Erzeugung von Beans durch den Container laufen auch hier ab.

Anschließend wird die Producer-Methode aufgerufen. Die Methode darf Exceptions auswerfen. Sollte beim Aufruf durch den Container eine Checked Exception auftreten, wird diese in eine *CreationException*[23] verpackt an den jeweiligen Aufrufer weitergeleitet.

Die Methode kann Qualifier besitzen, die ebenso wie bei anderen CDI Beans bei der Injektion von Werten berücksichtigt werden. Die implizite Zuordnung der Qualifier *@Any* und *@Default* geschieht wie bisher.

Der Methode kann auch mittels *@Named* ein Name zugewiesen werden. Der Default-Name, der bei Verwendung von *@Named* ohne Parameter vergeben wird, ist der Methodenname oder, falls die Methode eine Getter-Methode im Sinne von JavaBeans ist, der zugehörige Property-Name.

Besitzt eine Producer-Methode Parameter, sind diese Injektionsziele, werden vom Container also per Injektion mit Werten versehen. Parameter des Typs *InjectionPoint* haben dabei eine besondere Bedeutung. Dies wird weiter unten im Abschnitt „Introspektion des Injektionsziels“ erläutert.

---

22 *javax.enterprise.inject.Produces*
23 *javax.enterprise.inject.CreationException*

Producer Fields

Eine vereinfachte Variante der Producer Methods sind mit *@Produces* annotierte Klassen- oder Instanzvariablen einer CDI Bean. Sie liefern den Wert der Variablen zur Injektion in andere CDI Beans (Listing 2.24).

```
@Resource(lookup = "jdbc/ee_demos")
@Produces
private DataSource dataSource;
```

Listing 2.24: Producer Field

Disposer Methods

Wird von einer Producer Method ein Objekt geliefert, für das nach seiner Verwendung ein Cleanup nötig ist – bspw. das Schließen einer geöffneten Verbindung –, so kann dazu in der gleichen Klasse eine Disposer Method bereitgestellt werden. Das ist eine Methode mit genau einem Parameter von dem Typ, den die Producer-Methode liefert, und der Annotation *@Disposes*[24] (Listing 2.25).

```
public class DatabaseConnectionProducer
{
  @Produces
  public Connection createConnection() { … }

  public void disposeConnection(@Disposes Connection connection)
  {
    if (!connection.isClosed())
      connection.close();
  }
  …
```

Listing 2.25: Disposer Method (Exception Handling aus Platzgründen nicht dargestellt)

Die Methode muss nicht *public* sein und darf auch einen Return-Wert liefern, der im beschriebenen Zusammenhang aber nicht verwendet wird.

Der sog. *Disposed Parameter* kann Qualifier besitzen. Auch hier wird *@Default* implizit angenommen, wenn kein anderer Qualifier vorhanden ist. Die Qualifier des Parameters müssen zu einer oder mehreren Producer-Methoden passen, d. h. es muss in der Klasse zumindest eine Producer-Methode geben, die mindestens die Qualifier des betroffenen Parameters haben. Mit anderen Worten: Jede Disposer-Methode muss passende Producer-Methoden haben, umgekehrt darf zu einer Producer-Methode maximal eine Disposer-Methode passen – jeweils in der gleichen Klasse (Listing 2.26).

24 *javax.enterprise.inject.Disposes*

```
public class DatabaseConnectionProducer
{
  @Produces
  public Connection createConnection() { … }

  @Produces @TempDb
  public Connection createTempConnection() { … }

  public void disposeConnection(@Disposes @Any Connection conn) { … }
  …
```

Listing 2.26: Disposer Method für mehrere Producer

Wichtig ist im Beispiel die Angabe von *@Any* für den *Disposed*-Parameter, sonst würde implizit *@Default* eingesetzt, womit die Methode nur auf den ersten Producer passen würde.

Eine Disposer-Methode darf weitere Parameter annehmen. Diese sind Injektionsziele, werden also vom Container per Injektion mit Werten versorgt.

Introspektion des Injektionsziels

Besitzt eine Producer Method einen Parameter vom Typ *InjectionPoint*[25], injiziert der Container dort Informationen über die Injektionsstelle. Das kann man bspw. nutzen, um das injizierte Objekt an sein Ziel anzupassen. Die Spezifikation enthält dazu ein Beispiel, das in der Praxis recht nützlich ist: Die Bereitstellung von Logger-Objekten. Listing 2.27 zeigt eine angepasste Version, die Apache-Commons-Logging-Objekte[26] liefert, die mit der Klasse des Injektionsziels parametrisiert sind.

```
public class LoggerProducer
{
  @Produces
  public static Log getLogger(InjectionPoint injectionPoint)
  {
    Class<?> targetClass
      = injectionPoint.getMember().getDeclaringClass();
    return LogFactory.getLog(targetClass);
  }
}

public class DemoModel
{
  @Inject
  private Log logger; // Logger für Kanal "DemoModel"
  …
```

Listing 2.27: Bereitstellung von an das Injektionsziel angepassten Log-Objekten

25 *javax.enterprise.inject.spi.InjectionPoint*
26 *org.apache.commons.logging.Log*

InjectionPoint bietet neben *getMember* weitere Methoden an, mit denen noch mehr Informationen über die Injektionsstelle abgefragt werden können. Für Details sei auf die Dokumentation des Interfaces und die CDI-Spezifikation verwiesen (5.5.7. Injection Point Metadata).

Der Parameter vom Typ *InjectionPoint* ist nur erlaubt, wenn die Producer-Methode den Scope *@Dependent* hat. Scopes werden im nächsten Abschnitt behandelt.

2.9 Kontexte und Scopes

Bislang haben wir uns um die reine „Verdrahtung" der Anwendung gekümmert, d. h. um die Zuordnung von Beans untereinander mittels Dependency Injection. Damit ist der Name CDI aber nur zu zwei Dritteln erklärt. Das C in CDI steht für *Contexts* und deutet an, dass die CDI-Objekte Kontexten zugeordnet werden können. Eng damit verbunden ist der Begriff *Scope*, der die Lebensdauer der Objekte adressiert und auch assoziiert, dass die Objekte gemanagt, also nach gewissen Regeln vom Container erzeugt und wieder zerstört werden.

Scopes sind Ihnen aus dem Bereich der Webanwendungen vermutlich schon ein Begriff. Wir unterscheiden hier Request-, Session- und Application Scope. Dahinter stehen Objekte, die eine entsprechende Lebensdauer haben (Tabelle 2.3).

Scope	**Lebensdauer**	**Java-Typ** **Zugriff auf das Scope-Objekt in Servlets** **Vordefinierte JSP-Variable**
Request	Ein Request	*ServletRequest* oder *HttpServletRequest* *request* (Parameter der Service-Methode) *request*
Session	Vom Beginn einer Sitzung bis zu ihrem Ende	*HttpSession* *request.getSession(true)* *session*
Application	Vom Anwendungsstart an solange die Anwendung läuft	*ServletContext* *request.getServletContext()* *application*

Tabelle 2.3: Scopes von Webanwendungen

Die in der Tabelle angegebenen Informationen über Java-Typen und Zugriffsmöglichkeiten sollen Ihnen zur Orientierung dienen, wenn Sie bereits mit Servlets oder JavaServer Pages gearbeitet haben. Wenn nicht, vergessen Sie es einfach, denn Sie werden sich in den allermeisten Fällen mit diesen Dingen nicht mehr belasten müssen.

Die Objekte erlauben es mit Zugriffsfunktionen ähnlich denen in *Map*, andere Objekte in ihnen zu speichern. Diese haben dann die gleiche Lebensdauer, werden also am Ende des jeweiligen Lifecycle dem Garbage Collector überlassen.

CDI kennt neben den drei angesprochenen Scopes noch einen weiteren und kann mithilfe der Portable Extensions sogar noch beliebig ergänzt werden.

Der Fokus bezüglich der Scopes liegt in diesem Buch auf Webanwendungen. Scopes sind aber auch in anderen Umgebungen nutzbar, bspw. in EJB-Anwendungen.

Request Scope

Der Begriff „Request" stammt im Wesentlichen aus dem Bereich der Webanwendungen, wo der Browser z. B. aufgrund der Betätigung eines Buttons einen Request auslöst, der auf Serverseite bearbeitet wird und mit dem Senden der Response zum Browser endet.

CDI Beans werden durch die Annotation *@RequestScoped*[27] dem Request Scope zugeordnet. Das hat zwei Effekte: Einerseits bleibt ein entsprechendes Objekt wie schon erwähnt bis zum Ende des aktuellen Requests erhalten. Der zweite Effekt wird offenbar, wenn man mehrere Injektionsstellen des betroffenen Typs betrachtet: Sie verweisen innerhalb eines Requests alle auf das gleiche Objekt (Listing 2.28).

```
@RequestScoped
public class RequestInfoBean
{
  …
}

public class ScopeModel
{
  @Inject
  private RequestInfoBean requestInfoBean;
  …
}

public class SomeOtherBean
{
  @Inject // referenziert gleiches Objekt wie DemoModel
  private RequestInfoBean requestInfoBean;
  …
}
```

Listing 2.28: CDI Bean im Request Scope

27 *javax.enterprise.context.RequestScoped* (Achtung: nicht *javax.faces.bean.RequestScoped*!)

entwickler.press

Session Scope

In typischen Anwendungen muss man sich häufig Dinge länger als nur für einen Request merken. Das klassische Beispiel ist der Warenkorb einer Shopanwendung, der seinen Inhalt während des gesamten Einkaufsvorgangs behalten soll. Dazu lässt sich der Session Scope verwenden, dessen Lebensdauer sich vom ersten Zugriff bis zum Ende der Sitzung erstreckt. Die CDI-Annotation dazu ist *@SessionScoped*[28](Listing 2.29).

```
@SessionScoped
public class SessionInfoBean implements Serializable
{
  …
```

Listing 2.29: CDI Bean im Session Scope

Wird eine damit annotierte Bean in eine andere injiziert, ist gewährleistet, dass über die Laufzeit der Session immer dasselbe Objekt referenziert wird.

Na ja, das ist nicht ganz richtig: Sie haben im Beispiel vielleicht die Deklaration des Interfaces *Serializable*[29] bemerkt. Sie ist nötig, damit sich der Container zeitweise sitzungsgebundener Objekten entledigen kann, um Hauptspeicherplatz zu gewinnen. Diese sog. Passivierung wird durch Serialisierung der Objekte in einen Sekundärspeicher durchgeführt. Das genaue Verfahren ist providerabhängig, ebenso der Zeitpunkt der Passivierung. Klar ist, dass der Vorgang für die Anwendung transparent ist, d. h. die Auslagerung geschieht nur, wenn ein Objekt derzeit nicht verwendet wird, und es wird wieder eingelagert – aktiviert –, wenn die Anwendung erneut darauf zugreift. Insofern kann ein referenziertes Objekt über die Lebenszeit einer Session zwar wechseln, sein Inhalt bleibt aber dadurch unverändert.

Wie man eine Sitzung beginnt und beendet, wird im Kapitel „JavaServer Faces“ ein Thema sein. Für jetzt können Sie davon ausgehen, dass eine Session automatisch beginnt und dann endet, wenn Sie den Browser schließen.

Sitzungsgebundene Objekte bergen eine Gefahr: Sie tendieren dazu, groß zu werden. Dieses *Fat Session Problem* hat seine Ursache darin, dass es leicht passieren kann, dass Objekte in der Session liegengelassen werden, obwohl sie nicht mehr benötigt werden. Entfernt man bspw. den besagten Warenkorb nach dem Einkaufsvorgang nicht aus der Session, bleibt der zugehörige Speicher belegt.

Application Scope

Für eine noch längere Lebensdauer kann der Application Scope genutzt werden. Die darin liegenden Objekte sind applikationsglobal, d. h. während der Laufzeit der Applikation

28 *javax.enterprise.context.SessionScoped* (Achtung: nicht *javax.faces.bean.SessionScoped*!)
29 *java.io.Serializable*

werden diese Werte nur jeweils einmal erzeugt und vorrätig gehalten. Die zugehörige Annotation ist *@ApplicationScoped*[30] (Listing 2.30).

```
@ApplicationScoped
public class ApplicationInfoBean
{
  …
```

Listing 2.30: Deklaration einer CDI Bean im Application Scope

Der Application Scope leidet theoretisch auch unter dem oben erwähnten Fat Session Problem. In der Praxis ist die Gefahr aber gering, da im Application Scope in aller Regel keine fachlichen Objekte abgelegt werden, sondern nur einzelne Dienste o. ä., die eher technischer Natur sind und wenig Speicherplatz belegen. Für den Application Scope ist im Standard daher auch kein Passivierungsverfahren vorgesehen.

Conversation Scope

Der Session Scope wird häufig genutzt, um Request-übergreifende Daten zu halten, ist dafür aber meist ‚zu groß', d. h. seine Laufzeit ist zu lang, länger als der jeweilige Geschäftsprozess die Daten benötigt. Hier bietet CDI mit dem Conversation Scope eine sehr interessante Erweiterung an: Eine Konversation ist eine Art Minisession, deren Laufzeit vom Programm in einfacher Weise gesteuert werden kann. CDI Beans werden durch die Annotation *@ConversationScoped*[31] im Conversation Scope platziert. Da der Conversation Scope wie schon der Session Scope eine Passivierung unterstützt, muss die Bean Serialisierung unterstützen (Listing 2.31).

```
@ConversationScoped
public class ScopeModel implements Serializable
{
  …
```

Listing 2.31: Eine Bean im Conversation Scope

Eine Konversation verhält sich zunächst wie ein Request, d. h. sie beginnt und endet mit der Bearbeitung eines Requests. Der Standard nennt eine Konversation in diesem Zustand *transient*. Während der Request-Bearbeitung kann man nun auf ein Objekt des Typs *Conversation*[32] zugreifen und eine andauernde (*long-running*) Konversation beginnen bzw. sie später wieder beenden. Das *Conversation*-Objekt erhält man – Sie ahnen es schon – per Injektion (Listing 2.32).

30 *javax.enterprise.context.ApplicationScoped* (Achtung: nicht *javax.faces.bean.ApplicationScoped*!)
31 *javax.enterprise.context.ConversationScoped*
32 *javax.enterprise.context.Conversation*

```
@Inject
private Conversation conversation;

public void beginConversation()
{
  if (this.conversation.isTransient())
    this.conversation.begin();
}

public void endConversation()
{
  if (!this.conversation.isTransient())
    this.conversation.end();
}
```

Listing 2.32: Konversation beginnen und beenden

Eine transiente Konversation, d. h. insbesondere eine über mehrere Requests genutzte und schließlich beendete Konversation, gibt die in ihr gespeicherten Objekte automatisch frei, sodass dem Fat Session Problem mit Konversationen elegant ausgewichen werden kann. Die Lebensdauer einer Konversation entspricht dazu im Allgemeinen dem zugehörigen Geschäftsprozess.

Bean Proxies

CDI-Objekte der sog. normalen Scopes Request, Session, Application und Conversation werden nicht direkt in andere Objekte injiziert, sondern in Form von dynamischen Proxies. Dadurch ist der Container einerseits in der Lage, die beschriebene Passivierung und Aktivierung von Objekten für die restliche Anwendung transparent durchzuführen. Andererseits ist damit eine beliebige Kombination von Scopes in einem Objektgeflecht möglich. So ist es z. B. vollkommen unproblematisch, in ein *@RequestScoped*-Objekt eine Bean aus dem Application Scope zu injizieren und darin wiederum ein *@SessionScoped*-Objekt zu referenzieren. Trotz dieser Mischung wird beim Zugriff auf das zuletzt genannte Objekt die Zuordnung zur aktuellen Session berücksichtigt, auch wenn diese über die Lebenszeit der Applikation vermutlich schon mehrfach gewechselt hat.

Für die Erstellung der dynamischen Proxies wird in den Beans ein parameterloser Konstruktor benötigt. Es reicht, wenn er *protected* ist.

Dependent Scope

CDI Beans ohne eine Scope-Annotation sind implizit dem Pseudo-Scope *Dependent* zugeordnet. Mithilfe der Annotation *@Dependent*[33] lässt sich das auch explizit ausdrücken (Listing 2.33).

33 *javax.enterprise.context.Dependent*

```
@Dependent
public class DependentInfoBean
{
  …
```

Listing 2.33: Bean im Pseudo-Scope „Dependent“

Objekte aus dem Dependent Scope werden anders als solche aus den normalen Scopes direkt injiziert. Es werden hier also keine Proxies erstellt. Das führt einerseits dazu, dass für jede Injektionsstelle ein neues Objekt erstellt wird, andererseits unterliegen Dependent-Objekte keiner eigenen Lifecycle-Steuerung. Stattdessen fügen sich die Objekte in den Lifecycle der Objekte ein, in die sie injiziert werden.

Dependent-Objekte sollten serialisierbar sein, um eine Injektion in einen passivierenden Scope zu unterstützen.

Die zuvor beschriebene Introspektion der Injektionsstelle ist nur für Dependent-Objekte zulässig, da nur hier eine direkte Zuordnung zwischen Objekt und Ziel erfolgt.

Qualifier @New

Soll eine CDI Bean aus einem der normalen Scopes für eine bestimmte Injektionsstelle so behandelt werden, als wäre sie *@Dependent*, kann dafür der Qualifier *@New*[34] verwendet werden. Dadurch wird effektiv ein neues Objekt injiziert, statt ein Objekt aus dem jeweiligen Kontext zu benutzen. Dieses Feature hat nur wenige Anwendungsfälle. Für weitere Details sei daher auf die CDI-Spezifikation verwiesen (3.12. @New qualified beans).

2.10 Interceptors

Interceptors sind Klassen, die zusätzlichen Programmcode enthalten, der vor und nach einem Aufruf einer Methode einer anderen Klasse ablaufen soll. Sie stellen damit eine Form der aspektorientierten Programmierung im Java-Standard dar. Interceptors dienen häufig der Implementierung eher technischer, klassenübergreifender Aspekte und helfen damit, fachlichen und technischen Programmcode voneinander zu trennen. Häufig genannte Beispiele für technische Aspekte sind Protokollierung oder Transaktionssteuerung.

Die Interceptor-Spezifikation ist ein Anhang zur EJB-Spezifikation[35]. Trotz dieser scheinbaren Verbindung zu EJBs sind Interceptors darin eher übergreifend für die gesamte Plattform beschrieben. Im Folgenden werden sie aber durch die CDI-Brille betrachtet und es

34 *javax.enterprise.inject.New*
35 Interceptors 1.1,
EJB 3.1 Expert Group, 10. Dezember 2009,
http://jcp.org –> Search JSR 318 –> Final Release Download –> interceptors-1_1-fr-spec.pdf

werden nur die Eigenschaften berücksichtigt, die im Rahmen dieses Kapitels relevant sind.

Interceptor Class

Interceptors sind einfache Klassen, die mit *@Interceptor*[36] annotiert sind und einen parameterlosen Konstruktor besitzen. Darüber hinaus haben sie eine Methode mit der Signatur *Object name(InvocationContext) throws Exception*, die mit *@AroundInvoke*[37] annotiert ist. Wird der Interceptor einer Methode einer CDI Bean zugeordnet, so umhüllt die *AroundInvoke*-Methode jeden Aufruf der Bean-Methode (Listing 2.34).

```
@Interceptor
public class TransactionRequiredInterceptor implements Serializable
{
  @Resource
  UserTransaction userTransaction;

  @AroundInvoke
  public Object manageTransaction(InvocationContext invocationContext)
    throws Exception
  {
    // Falls schon eine TX aktiv, Methode direkt aufrufen
    if (isTransactionActive())
      return invocationContext.proceed();


    // TX starten
    this.userTransaction.begin();

    // Bean-Methode aufrufen
    Object result = invocationContext.proceed();

    // TX abschliessen
    this.userTransaction.commit();

    return result;
  }
}
```

Listing 2.34: Grundsätzlicher Aufbau eines Interceptors

Einer Bean-Methode können später durchaus mehrere Interceptors zugeordnet werden, die in der Art einzelner Schichten um die Bean-Methode herum funktionieren. Die Methode *InvocationContext.proceed* leitet den Aufruf weiter an die nächste Schicht, bis schließlich die Bean-Methode aufgerufen wird. Dadurch kann vor und nach dem Ablauf der Bean-Methode beliebiger Programmcode ausgeführt werden.

36 *javax.interceptor.Interceptor*
37 *javax.interceptor.AroundInvoke*

Im Beispiel ist ein Interceptor gezeigt, der zur Ausführung einer Methode eine aktive Transaktion sicherstellt. Der Ausschnitt in Listing 2.34 ist allerdings stark vereinfacht: Er enthält keine Behandlung von Exceptions und schließt die Transaktion stets mit Commit ab, was sicher im wahren Leben nicht immer erwünscht ist. Die vollständige Implementierung finden Sie im Begleitprojekt.

Ein Interceptor wird im gleichen Scope verwaltet wie das Objekt, dem er zugeordnet ist. Damit das auch für die passivierenden Scopes funktioniert, sollte ein Interceptor serialisierbar sein.

Interceptor Binding

Die Verknüpfung eines Interceptors mit einer Bean-Methode geschieht nun in einer Weise, die an Qualifier erinnert, nämlich mithilfe einer Annotation auf beiden Seiten. Diese Annotation muss ein Interceptor Binding darstellen, d. h. sie muss selbst mit *@InterceptorBinding*[38] annotiert sein (Listing 2.35).

```
@InterceptorBinding
@Target({ TYPE, METHOD })
@Retention(RUNTIME)
public @interface TransactionRequired
{
}
```

Listing 2.35: Interceptor Binding passend zu Listing 2.34

Mit einer solchen Annotation müssen dann sowohl der gewünschte Interceptor als auch die zu umhüllende Bean-Methode annotiert werden. Wird die Annotation für eine CDI Bean verwendet, gilt sie automatisch für alle darin enthaltenen Methoden (Listing 2.36).

```
@TransactionRequired
@Interceptor
public class TransactionRequiredInterceptor implements Serializable
{
  …
}

public class CocktailJdbcRepository implements CocktailRepository
{
  @TransactionRequired
  public void insert(Cocktail cocktail)
  {
    …
  }
  …
```

Listing 2.36: Verknüpfung von Interceptor und Methode durch Interceptor Binding

38 *javax.interceptor.InterceptorBinding*

Sind für einen Interceptor mehrere Interceptor Bindings definiert, muss die Methode mit allen annotiert sein, um mit dem Interceptor verknüpft zu werden.

Wie bei einem Qualifier gehen auch bei einem Interceptor Binding eventuelle Parameter in die Zuordnung ein. Sollen einem Interceptor Binding also Parameter mitgegeben werden, die die Interceptor-Zuordnung nicht beeinflussen sollen, müssen diese mit *@Nonbinding*[39] annotiert werden (Listing 2.37).

```
@InterceptorBinding
@Target({ TYPE, METHOD })
@Retention(RUNTIME)
public @interface Transactional
{
  @Nonbinding
  TransactionAttributeType value();
}
```

Listing 2.37: Interceptor Binding mit einem Parameter, der nicht zur Bindung verwendet wird

Aktivierung eines Interceptors

Ein Interceptor ist zunächst inaktiv. Seine Aktivierung geschieht durch einen Eintrag im Deskriptor *beans.xml*, und zwar im Element *<interceptors>*. Dort sind die voll qualifizierten Namen aller aktiven Interceptor-Klassen einzutragen. Die Anordnung der Einträge entscheidet darüber hinaus über die Reihenfolge der Interceptor-Aufrufe bei den Methoden, denen mehrere Interceptors zugeordnet sind.

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <interceptors>
    <class>de.gedoplan. … .TransactionRequiredInterceptor</class>
    <class>de.gedoplan. … .TraceCallInterceptor</class>
    …
```

Listing 2.38: Interceptor-Aktivierung im Deskriptor „beans.xml“

2.11 Decorators

Ein Decorator ähnelt in seiner Funktion einem Interceptor: Er wird auch in den Aufrufweg eingeschleust, kann also die Semantik von Bean-Methoden nahezu beliebig verändern. Anders als ein Interceptor implementiert der Decorator aber ein fachliches Interface, wodurch er sich gut für die Modifikation fachlicher Logik eignet.

39 *javax.enterprise.util.Nonbinding*

Decorator Class

Ein Decorator ist eine Klasse, die mit *@Decorator*[40] annotiert ist und einen Bean Type implementiert. Zudem benötigt der Decorator genau einen Injektionspunkt für sein Delegate, d. h. für das Objekt, das er dekoriert, auf dessen Funktionalität sich sein Code abstützt. Dieser Injektionspunkt muss mit *@Delegate*[41] annotiert sein. Es kann sich dabei um ein Feld, einen Methodenparameter oder einen Konstruktorparameter handeln (Listing 2.39).

```
@Decorator
public abstract class NonAlcoholicCocktailRepository
  implements CocktailRepository
{
  @Inject @Delegate @Any
  private CocktailRepository cocktailRepository;

  public List<Cocktail> findAll()
  {
    List<Cocktail> result = this.cocktailRepository.findAll();

    Iterator<Cocktail> iterator = result.iterator();
    while (iterator.hasNext())
    {
      Cocktail cocktail = iterator.next();
      if (cocktail.isAlcoholic())
        iterator.remove();
    }

    return result;
  }
}
```

Listing 2.39: Decorator

Der Injektionspunkt für das Delegate bestimmt, für welche CDI Beans der Decorator eingesetzt wird, nämlich genau diejenigen, die mindestens die Qualifier des Delegates haben. Im Beispiel steht der Decorator also für alle Beans des Typs CocktailRepository zur Verfügung, unabhängig von ihren restlichen Qualifiern. Würde man *@Any* weglassen, wäre der Decorator nur für *@Default-CocktailRepositories* zuständig.

Der Decorator darf *abstract* sein, muss somit auch nicht alle Methoden des dekorierten Interfaces implementieren. Die restlichen Methoden werden dann durch direkten Aufruf des dekorierten Objekts aufgelöst. Mit dem gezeigten Beispiel wird also nur die Funktion der *findAll*-Methode verändert. Aufrufe der anderen Methoden aus *CocktailRepository* werden direkt zum Delegate durchgeleitet.

40 *javax.decorator.Decorator*
41 *javax.decorator.Delegate*

Die dekorierte Version im Beispiel entfernt übrigens die alkoholischen Drinks aus der Ergebnisliste, sodass Sie das Buch nun erst recht weiterlesen sollten, selbst wenn Sie alkoholische Beispiele für moralisch grenzwertig halten.

Aktivierung eines Decorators

Analog zu einem Interceptor muss auch ein Decorator im Deskriptor *beans.xml* eingetragen werden, wenn er aktiv sein soll. Das zugehörige Element ist *<decorators>* (Listing 2.40).

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
     http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <decorators>
    <class>de.gedoplan. … .NonAlcoholicCocktailRepository</class>
    …
```

Listing 2.40: Decorator-Aktivierung in „beans.xml“

2.12 Stereotypes

In der Praxis werden CDI Beans häufig mit immer wiederkehrenden Kombinationen von Scopes und Interceptor Bindings versehen, die den Elementen der verwendeten Architekturmuster entsprechen. So könnten etwa die Objekte, die aus einer auf Basis von JSF entwickelten Benutzeroberfläche angesprochen werden, immer mit *@Named* und *@RequestScoped* annotiert sein.

CDI bietet zur Modellierung solcher wiederkehrender Bean-Eigenschaften Stereotypes an. Das sind Annotationen, die selbst mit *@Stereotype*[42] und den gewünschten weiteren Annotationen versehen sind. Listing 2.41 zeigt den bereits vordefinierten Stereotype *@Model*[43], der für Beans gedacht ist, die als Model in einer MVC-basierten Anwendung genutzt werden sollen.

```
@Named
@RequestScoped
@Stereotype
@Target( { TYPE, METHOD, FIELD })
@Retention(RUNTIME)
public @interface Model
{
}
```

Listing 2.41: Stereotype

42 *javax.enterprise.inject.Stereotype*
43 *javax.enterprise.inject.Model*

Die erlaubten Elemente eines Stereotyps sind:

- Eine Scope-Annotation
- *@Named*
- *@Alternative*
- Interceptor Bindings
- Andere Stereotypes

Eine CDI Bean kann dann anstelle der einzelnen Annotationen mit einem oder mehreren Stereotypes markiert werden. Ein im Stereotype enthaltener Scope wirkt dabei als Vorgabewert, der mit einer zusätzlich angegebenen Scope-Annotation überschrieben werden kann (Listing 2.42).

```
@Model
@SessionScoped
public class SomeModel
{
  …
```

Listing 2.42: Nutzung eines Stereotyps, aber mit anderem Scope

Das Gleiche gilt für die Annotation *@Named*: Ist sie im Stereotyp enthalten, wird ein Standardname vergeben, wenn die Bean nicht zusätzlich mit *@Named("…")* annotiert ist.

Enthält ein Stereotype die Annotation *@Alternative*, so sind damit annotierte Beans nach den oben beschriebenen Regeln inaktiv, wenn sie nicht in *beans.xml* explizit aktiviert werden (Listing 2.43). Hier gibt es eine Ergänzung: Durch Angabe des Stereotyps im Deskriptor können alle damit markierten Alternativ-Beans gemeinsam aktiviert werden (Listing 2.44).

```
@Alternative
@Stereotype
@Retention(RUNTIME)
@Target({ METHOD, FIELD, TYPE })
public @interface Mock
{
}

@Mock
public class CocktailMockRepository implements CocktailRepository
{
  …
```

Listing 2.43: Nutzung eines Stereotyps zur Markierung von Alternativ-Beans

```
<beans xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/beans_1_0.xsd">
  <alternatives>
    <stereotype>de.gedoplan.buch.eedemos.stereotype.Mock</stereotype>
    …
```

Listing 2.44: Aktivierung alternativer Beans mittels Stereotyp

2.13 Eventverarbeitung

Die Bereitstellung von Objekten durch den Container und ihre Injektion in Beans reduziert die Kopplung der Anwendungsteile untereinander deutlich: Eine benutzende Komponente „weiß" nicht mehr, wie die benutzten Objekte herzustellen sind. Wird mit Interfaces gearbeitet, ist ihr sogar der konkrete Typ der verwendeten Objekte unbekannt.

Man kann Anwendungskomponenten noch weiter voneinander entkoppeln, indem man eine Event-basierte Kommunikation verwendet: Eine Komponente sendet bspw. aufgrund eines Zustandswechsels einen Event aus. Andere Komponenten reagieren darauf mit der Ausführung eines beliebigen Programmcodes. Die Komponenten „kennen" einander nicht – das einzige verbindende Element ist der verarbeitete Event.

Events erzeugen

Events bestehen in CDI aus einem Java-Objekt und einer – ggf. leeren – Menge von Qualifiern. Sie werden mithilfe einer Event Source versendet. Das ist ein Objekt des Typs *Event<T>*[44], das in die aufrufende Bean mit den gewünschten Qualifiern injiziert wird. Der Typparameter gibt den Typ der zu versendenden Java-Objekte – den Eventtyp – an (Listing 2.45).

```
public class CocktailModel
{
  …
  // zum Feuern von Inhouse-Bestell-Events
  @Inject @Inhouse
  private Event<CocktailOrder> cocktailInhouseOrderEvent;

  // zum Feuern von Takeaway-Bestell-Events
  @Inject @Takeaway
  private Event<CocktailOrder> cocktailTakeawayOrderEvent;
  …
```

Listing 2.45: Event Sources für Events des Typs „CocktailOrder" mit verschiedenen Qualifiern

44 *javax.enterprise.event.Event*

Der Versand – das „Feuern“ – von Events geschieht durch Aufruf der Methode *fire* der Event Source (Listing 2.46).

```
public class CocktailModel
{
  …
  public void bestellungAbschliessen(boolean takeaway)
  {
    …
    this.cocktailInhouseOrderEvent.fire(this.cocktailOrder);
    …
```

Listing 2.46: Feuern eines Events

Events verarbeiten

Zur Verarbeitung der gefeuerten Events wird eine Observer Method benötigt. Das ist eine Methode einer beliebigen CDI Bean, die einen mit *@Observes*[45] und den gewünschten Qualifiern annotierten Parameter hat. Der Parametertyp entscheidet zusammen mit den Qualifiern darüber, welche Events von der Methode empfangen werden: Es werden die Events berücksichtigt, deren Event Type dem Parameter zugewiesen werden kann und die mindestens die Qualifier des Parameters haben (Listing 2.47).

```
public class CocktailMixer
{
  public void mixCocktail(@Observes CocktailOrder cocktailOrder)
  {
    …
  }
}

public class TakeawayCounter
{
  public void count(@Observes @Takeaway CocktailOrder cocktailOrder)
  {
    …
  }
}
```

Listing 2.47: Observer Methods

Im Beispiel wird also *CocktailMixer.mixCocktail* für jeden Event des Typs *CocktailOrder* aufgerufen, während *TakeawayCounter.count* nur dann zum Zug kommt, wenn der Event den Qualifier *@Takeaway* trägt.

45 *javax.enterprise.event.Observes*

Die Methoden dürfen auch *static* sein und eine andere Sichtbarkeit als *public* haben. Sind neben dem mit *@Observes* annotierten weitere Parameter vorhanden, sind dies Injektionsziele, die nach den bekannten Regeln mit Werten versorgt werden (Listing 2.48).

```
public class EventLogger
{
  public static void logEvent(@Observes Object event, Log logger)
  {
    …
```

Listing 2.48: Observer mit zusätzlichem Parameter

Das Beispiel zeigt auch, dass der Event-Parameter sehr allgemein angelegt werden kann: *EventLogger.logEvent* wird für Events beliebigen Typs aufgerufen.

CDI-Events werden synchron verarbeitet, d. h. die *fire*-Methode kehrt erst dann zum Aufrufer zurück, wenn alle Observer Methods abgearbeitet wurden. Sollten mehrere Observer Methods zum Event passen, ist deren Ausführungsreihenfolge nicht vorbestimmt. Eine Exception in einer der Methoden bricht die Bearbeitung an dieser Stelle ab.

Dieser Abschnitt stellt nur einen Teil des CDI-Event-Systems dar. Weitere Möglichkeiten sind

- Programmgestütztes Hinzufügen von Qualifiern beim Feuern von Events
- Bedingter Empfang von Events
- Transaktionale Verarbeitung von Events

Details dazu finden Sie in der CDI-Spezifikation in Kapitel 10 (Events).

2.14 Programmgesteuerter Zugriff auf CDI Beans

Die bislang besprochenen Injektionsmöglichkeiten sind zweifellos in den meisten Fällen genau die Mittel der Wahl. Es gibt aber auch Situationen, in den man mehr eigene Kontrolle über die Auswahl und Nutzung von CDI Beans benötigt, um z. B. bestimmte Beans dynamisch auszuwählen, alle Beans eines bestimmten Typs aufzurufen etc. CDI bietet dazu einerseits eine Injektionsmöglichkeit für alle verfügbaren Instanzen einer Bean und andererseits den Zugriff auf den zentralen Bean Manager.

Injektion von Bean-Instanzen

Mithilfe des Interface *Instance<T>*[46] können alle verfügbaren Beans eines Bean Type, die zu einer gegebenen Menge von Qualifiern passen, in eine CDI Bean injiziert werden. Die

46 *javax.enterprise.inject.Instance*

Instanziierung der gefundenen Beans geschieht mithilfe der Methode *Instance.get*, wenn nur eine Bean gefunden wurde. Andernfalls kann über die Beans iteriert werden.

```
public class BeanModel
{
  @Inject @Any
  private Instance<GreetingBean> greetingBeans;

  public List<Class<?>> getGreetingBeanClasses()
  {
    for (GreetingBean bean : this.greetingBeans)
    {
      …
```

Listing 2.49: Injektion ggf. mehrerer Beans

Instance bietet die Methoden *isAmbiguous* und *isUnsatisfied* an, mit denen festgestellt werden kann, ob die Bean-Menge mehrdeutig bzw. leer ist. Darüber hinaus kann mit *select* die gefundene Bean-Menge weiter reduziert werden. *select* erhält dazu die gewünschten Annotationen als Parameter. Leider ist es in Java etwas umständlich, Annotationen im Programm zu instanziieren. Als kleine Hilfe bietet CDI dazu das Interface *AnnotationLiteral<T>*[47] an, das entweder direkt inline zur Instantiierung von Annotation-Objekten (*new AnnotationLiteral<Any>(){}*) oder zur Definition von Annotations-Literal-Klassen genutzt werden kann (Listing 2.50).

```
public class FormalLiteral extends AnnotationLiteral<Formal>
  implements Formal
{
}
```

Listing 2.50: Klasse für Literale der Annotation „@Formal“

Im Beispiel wird eine Literalklasse für die Annotation *@Formal* gezeigt. Die Implementierung von *Formal* ist hier eigentlich überflüssig, da *@Formal* keine Attribute besitzt. Andernfalls müssten die entsprechenden Methoden mit aufgenommen oder die Klasse *abstract* gemacht werden.

Objekte solcher Literalklassen können dann als Parameter für *Instance.select* verwendet werden. Das Ergebnis des Methodenaufrufs ist die auf die angegebenen Qualifier eingeschränkte Bean-Menge (Listing 2.51).

```
public class BeanModel
{
  @Inject @Any
  private Instance<GreetingBean> greetingBeans;
```

47 *javax.enterprise.util.AnnotationLiteral*

entwickler.press

```
  public List<Class<?>> getFormalGreetingBeanClasses()
  {
    Instance<GreetingBean> selectedBeans
      = this.greetingBeans.select(new FormalLiteral());
    for (GreetingBean bean : selectedBeans)
    {
      …
```

Listing 2.51: Einschränkung der gefundenen Bean-Menge anhand eines Qualifiers

Bean Manager

Direkten Zugriff auf den Motor des CDI-Containers ermöglicht das Interface *BeanManager*[48]. Objekte dieses Typs lassen sich wie gewohnt per *@Inject* in eine Bean injizieren. *BeanManager* bietet diverse Methoden zur Interaktion mit dem Container an. Sie dienen hauptsächlich dazu, eigene Erweiterungen des CDI-Systems portabel entwickeln zu können, können aber teilweise auch im normalen Programmcode sinnvoll eingesetzt werden. Es würde den Rahmen dieses Abschnitts sprengen, *BeanManager* komplett beschreiben zu wollen, daher soll hier exemplarisch nur die Methode *getBeans* betrachtet werden, mit der ähnlich wie im vorigen Abschnitt Beans eines bestimmten Typs und mit bestimmten Qualifiern gesucht werden können. Die Methode nimmt dazu den Bean Type und eine beliebig große Menge von Annotationen als Parameter an. Für Letztere lassen sich wieder die Annotations-Literal-Klassen verwenden. Die Ergebnisliste enthält die gefundenen Beans in Form von *Bean*[49]-Objekten, die zur Abfrage der Eigenschaften der Beans (Qualifiers, Scope etc.) genutzt werden können.

```
public class BeanModel
{
  …
  @Inject
  private BeanManager beanManager;

  public List<CharSequence> getAllBeanDescriptions()
  {
    …


    for (Bean<?> bean :
      beanManager.getBeans(Object.class, new AnnotationLiteral<Any>(){}))
    {
      … bean.getQualifiers() …
      … bean.getBeanClass() …
      …
```

Listing 2.52: Zugriff auf alle CDI Beans mithilfe des Bean Managers

48 *javax.enterprise.inject.spi.BeanManager*
49 *javax.enterprise.inject.spi.Bean*

Im gezeigten Beispiel werden alle verfügbaren CDI Beans aufgelistet. Es zeigt dabei auch die Verwendung von AnnotationLiteral zur Inline-Instanziierung eines Annotations-Objekts.

Weitere Details finden Sie in der CDI-Spezifikation im Kapitel 11 (Portable extensions).

2.15 Integration von JPA, EJB und JSF

CDI stellt einen sehr zentralen Teil der Plattform Java EE dar. Insofern ist eine gute Anbindung anderer Plattformbestandteile ein wesentlicher Faktor für die Nutzbarkeit von CDI. Die im Folgenden angesprochenen Spezifikationen werden in entsprechenden Kapiteln dieses Buches noch im Detail dargestellt.

Java Persistence

Die JPA-Anbindung geschieht im Wesentlichen über die Injektion von *EntityManagerFactory* bzw. *EntityManager* mithilfe der Annotationen *@PersistenceUnit* bzw. *@PersistenceContext*. Diese sind aber nicht nur für CDI Beans gültig, sondern plattformweit definiert. Mithilfe einer Producer-Methode lässt sich recht leicht ein im CDI-Sinne injizierbarer Entity Manager zur Verfügung stellen.

Enterprise JavaBeans

EJBs sind auch gemanagte Objekte. Die Integration mit CDI Beans ist in beiden Richtungen problemlos möglich. EJBs veröffentlichen allerdings eine oder mehrere genau definierte Sichten auf sich (Local Interface etc.). Für CDI Beans sind auch nur diese Sichten nutzbar.

CDI und EJB überlappen sich in vielen Bereichen: Beide sind Komponentensysteme, beide unterstützen Dependency Injection, beide kennen unterschiedliche Lebenszyklen. Eine Verschmelzung der beiden Spezifikationen in einer der künftigen Versionen der Plattform ist gut denkbar. Trotz großer Überschneidungen bieten EJBs einige Aspekte, die sich derzeit mit CDI alleine nicht lösen lassen.

JavaServer Faces

JSF nutzt Managed Beans als Schnittstelle von der meist HTML-basierten Oberfläche zur Java-Programmlogik. An dieser Stelle ist die Überschneidung mit CDI 100 %ig: Die Managed Beans aus JSF lassen sich problemlos durch CDI Beans ersetzen, wenn diese mittels *@Named* einen Namen zugewiesen bekommen. Dieser ist dann wie oben dargestellt in Ausdrücken der JSF-EL nutzbar. Vorsicht ist geboten bei Benutzung der Scope-Annotationen: Die gibt es gleichnamig auch in JSF. Für CDI müssen aber die Annotationen aus dem Paket *javax.enterprise.context* verwendet werden, nicht die aus *javax.faces*.

2.16 Plattformen und Ergänzungen

Als Plattformen für CDI-Anwendungen stehen einige Applikationsserver zur Verfügung. Da CDI Teil des Java EE 6 Web Profile ist, sind prinzipiell alle Java-EE-6-Server geeignet.

Applikationsserver	CDI-Implementierung
Oracle GlassFish 3	JBoss Weld
JBoss AS 6 / 7	JBoss Weld
Caucho Resin 4	Caucho CanDI
Apache Tomcat + (TomEE, Siwpas)	Apache OpenWebBeans
IBM WebSphere 8	Apache OpenWebBeans
Oracle WebLogic 12	JBoss Weld

Tabelle 2.4: Einige Server mit CDI-Unterstützung

Die Funktionalität von CDI kann durch Portable Extensions ergänzt werden. Hier wurde darauf geachtet, dass die Ergänzungsbibliotheken portabel sind, also auf einer beliebigen Plattform eingesetzt werden können. Durch eine Extension können u. a. weitere Kontexte, Beans, Interceptors etc. hinzugefügt werden.

Die Bibliotheken nutzen zu ihrer Initialisierung das seit Java 6 verfügbare Servicekonzept der Sprachplattform sowie das CDI-Eventkonzept. Dadurch kann eine Extension allein dadurch für eine Anwendung aktiviert werden, indem sie in den Classpath aufgenommen wird, was üblicherweise durch die Platzierung im *lib*-Ordner der Anwendung geschieht. Eine weitere Konfiguration ist nicht erforderlich.

- Apache MyFaces CODI[50]

Diese Ergänzungsbibliothek besteht aus diversen Modulen, die nahezu unabhängig voneinander eingesetzt werden können. Einen Überblick über die enthaltenen Features gibt Tabelle 2.5. Für Details sei auf die CODI-Dokumentation verwiesen.

50 *http://myfaces.apache.org/extensions/cdi/*

Apache MyFaces CODI …	Inhalt u. a.
… Core	Project Stage Konfiguration des Entwicklungsstands eines Projekts bspw. mittels System Property und davon abhängige Aktivierung von Alternatives (s. Klasse *ProjectStage* und *@ProjectStageActivated*) Logger Injection Producer analog zu Listing 2.27, allerdings für *java.util.logging*-Logger Default Annotation Creator Methode zum bequemen Erzeugen von Annotationsliteralen (s. Klasse *DefaultAnnotation*)
… for JSF	Zusätzliche Scopes *@ConversationScoped*[5]: Konversationen ggf. parallel und gruppiert *@RestScoped*: Konversation für REST *@ViewAccessScoped*: Referenzierte Beans bleiben auch bei Seitenwechsel aktiv *@WindowScoped*: wie *@SessionScoped*, aber pro Browserfenster Annotation Mapping Mapped JSF-Annotationen *@ManagedBean*, *@RequestScoped* etc. auf CDI-Annotationen *@Named*, *@RequestScoped* etc.[6] Producer für diverse JSF-Ressourcen
… for Bean Validation	Injektionsmöglichkeiten für BV Constraints
… for Messages	Fluent API zum Aufbau lokalisierter Meldungen
… for Scripting	Injektionsmöglichkeiten und EL-Erweiterungen für den Aufruf von Skripten nach JSR 233

Tabelle 2.5: Einige Features von Apache MyFaces CODI

- JBoss Seam 3[51]

Der Vorläufer Seam 2 wurde als Framework für Rich Internet Applications entwickelt, wobei der Name Seam auf die Naht zwischen Weboberfläche auf Basis von JSF und Geschäftslogik in Form von EJBs oder POJOs hindeutet, die durch Seam geschlossen wird. Seam hatte einen großen Einfluss auf die Entstehung von CDI; ein großer Teil von Seam 2 ist Teil des Standards geworden. Seam 3 ergänzt nun wie CODI den Standard, u. a. um einige Features, die nicht von Seam 2 übernommen wurden. Einige interessante Elemente zeigt Tabelle 2.6.

51 *http://seamframework.org/Seam3*

Modul	Inhalt u.a.
Solder	Auflösung von Mehrdeutigkeiten durch Typangabe bei der Injektion:
	@Inject @Exact(FormalGreetingBean.class) *private GreetingBean greetingBean;*
	@Named für Packages (in *package-info.java*)
	Voll qualifizierte Klassennamen als Bean-Namen:
	@Named @FullyQualified *public class DemoModel { … }*
	Annotation Literals für Standardqualifier, z. B. *AnyLiteral.INSTANCE* für *@Any*
	Logger Injection
	Producer analog zu Listing 2.27, allerdings für JBoss Logging
	Konfiguration der Injektionen etc. mittels XML statt Annotationen
Faces	Integration des JSF-Scopes *@ViewScoped*
	Beans bleiben erhalten, solange die View nicht wechselt
Persistence	Deklarative Transaktionssteuerung mittels Annotation und Interceptor:
	@TransactionAttribute *public void transactionalMethod() { … }*
	Konversationsgebundener Entitymanager im Seam Managed Persistence Context:
	// Producer für SMPC *public class SeamManagedPersistenceContextProducer* *{* *@PersistenceUnit* *@ExtensionManaged* *@Produces @ConversationScoped* *EntityManagerFactory entityManagerFactory;* *}* *public class CountryJpaRepository implements CountryRepository* *{* *// Injektion eines Conversation Scoped EntityManagers* *@Inject* *private EntityManager entityManager;*

Tabelle 2.6: Einige Features von Seam 3

Viele der hier angesprochenen Ergänzungen werden erst unter Einbeziehung der folgenden Kapitel – insbesondere der über Java Persistence und JavaServer Faces – verständlich. Im letzten Buchkapitel finden Sie ein übergreifendes „Full Stack"-Projekt, das auch die genannten Erweiterungen benutzt.

3 Java Persistence

3.1 Worum geht's?

Vor der Behandlung der Details soll hier kurz abgesteckt werden, in welchem Bereich der Anwendungsentwicklung wir uns in diesem Kapitel befinden. Wie der Name schon sagt, geht es um Persistenz, d. h. das Speichern und Laden von Java-Objekten in bzw. aus einem dauerhaften Speichermedium. Java Persistence schränkt den Blick noch etwas weiter ein, und zwar auf relationale Datenbanken als Speichermedium, was sicher den allergrößten Teil von Unternehmensanwendungen abdeckt.

Die Aufgabenstellung lautet also, Objekte von Java-Klassen wie der in Listing 3.1 auf eine Tabelle einer relationalen Datenbank wie der in Abbildung 3.1 abzubilden, d. h. die grundlegenden Operationen der Datenbank wie Erzeugen und Löschen von Einträgen, Lesen und Verändern von Werten oder Suchanfragen möglichst einfach und effektiv zur Verfügung zu stellen.

```
public class Country
{
  private String isoCode; // Primärschlüssel
  private String name;
  private String phonePrefix;
  private String carCode;

  // Getter und Setter nach Bedarf …
}
```

Listing 3.1: Beispielhafte Java-Klasse für persistente Daten

Abbildung 3.1: Beispieltabelle

3.1.1 Lösungsansätze

Um die beschriebene Aufgabe zu lösen, stehen uns unterschiedliche Wege offen. Zunächst enthält Java mit JDBC eine Abstraktion für Datenbankzugriffe als Core Library. Damit kann man sich die gewünschten Operationen z. B. in einer Helferklasse zur Verfügung stellen (Listing 3.2).

```
public class CountryJdbcHelper
{
  public void save(Country country)
    throws ClassNotFoundException, SQLException
  {
    Connection connection = JdbcUtil.getConnection();
    PreparedStatement statement = null;
    try
    {
      statement = connection.prepareStatement(
       "insert into EE_DEMOS_COUNTRY(ISO_CODE,NAME,PHONE_PREFIX,CAR_CODE)
        values (?,?,?,?)");
      statement.setString(1, country.getIsoCode());
      statement.setString(2, country.getName());
      statement.setString(3, country.getPhonePrefix());
      statement.setString(2, country.getCarCode());
      statement.executeUpdate();
      …
```

Listing 3.2: DB-Zugriff per JDBC[1]

Aus Gründen der Übersichtlichkeit ist im Listing der Auf- und Abbau der Verbindung zur Datenbank weggelassen worden. Ebenso ist die Exception-Verarbeitung nur angedeutet. Man sieht daran schon, dass man sich bei der direkten Nutzung von JDBC auf einer recht niedrigen Ebene im Programm bewegt. Das ist zwar nicht wirklich schwierig, aber doch nicht mal eben gemacht.

Im Bereich der serverbasierten Anwendungen, mit dem sich dieses Buch beschäftigt, gab es schon recht früh Ansätze zur Lösung der Thematik auf einer etwas höheren Ebene. Der EJB-Standard enthält bis zur Version 2.1 Entity Beans, die von der direkten Nutzung von JDBC abstrahieren und bei Nutzung der sog. Container Managed Persistence die „niederen Aufgaben“ dem EJB-Container überlassen. Diese an sich gute Idee wurde allerdings unglücklicherweise mit dem Komponentenkonzept von EJBs vermischt, obwohl die damit verbundenen Vorteile wie z. B. Ansprechbarkeit über Remote- und Local-Interfaces, konfigurierbares Environment etc. für den Persistenzaspekt keine Vorteile bringen, ja sogar hinderlich sind. So gerieten die Entity Beans sehr schnell in den Ruf, schwergewichtige und unflexible Monster zu sein, was sogar auf die gesamte EJB-Spezifikation ausstrahlte und viele Projekte komplett von der Nutzung von EJB abgebracht hat.

1 Den in diesem Kapitel gezeigten Beispielcode finden Sie im Begleitprojekt *ee-demos-jpa*

Parallel zu den EJBs – teilweise sogar schon früher – traten die sog. O/R-Mapper auf den Plan. Die Object/Relational-Mapping-Frameworks verfolgen grob den Ansatz, möglichst wenig Restriktionen bezüglich der speicherbaren Objekte aufzubauen, also einfache Java-Klassen als persistente Klassen zu nutzen und sie so mit Metadaten in Form von XML oder später auch Annotationen anzureichern, dass mithilfe von Frameworkmethoden ein Speichern und Laden der Objekte möglich wird.

Das mündete einerseits in einen Java-Standard: Java Data Objects (JDO). Der Persistenzaspekt wurde bei den JDO-implementierenden Produkten häufig durch ein Bytecode Enhancement nach dem Compile der Klassen in den Bytecode hineingewebt, was zum damaligen Zeitpunkt ein neues und ungewöhnliches Vorgehen war. JDO hat interessanterweise nie die Marktbreite erreicht, die man eigentlich hätte erwarten können, vielleicht weil nahezu zeitgleich ein äußerst populäres Open-Source-O/R-Framework namens Hibernate auftauchte, das mit ganz ähnlichen Konzepten glänzte, statt des Bytecode Enhancements zur Build-Zeit allerdings auf dynamische Proxies und Bytecode-Modifikation zur Laufzeit setzte.

Die Ideen aus JDO und Projekten wie Hibernate sind es nun, die sich im Standard Java Persistence wiederfinden. Mehr noch: Produkte wie Hibernate, Eclipselink (Nachfolger des O/R-Urgesteins Toplink), oder DataNucleus (Referenzimplementierung von JDO) sind heute Implementierungen von JPA.

Ein Wort zu den Begrifflichkeiten: Der Standard heißt eigentlich Java Persistence. Im gängigen Sprachgebrauch findet sich aber häufig die gerade erwähnte Abkürzung JPA für Java Persistence API. Im Rahmen dieses Buches werden die Begriffe synonym verwendet.

3.1.2 Anforderungen an O/R-Mapper

Die Erwartungen an ein Persistenzframework lassen sich teilweise schon an der selbstgestrickten JDBC-Lösung ablesen. Darüber hinaus entwickeln sich rasch weitere Wünsche in Richtung Einfachheit und Komfort, sodass im Endeffekt eine Liste wie diese herauskommt – ohne Gewichtung und Anspruch auf Vollständigkeit:

- Verbindungsverwaltung
- Mapping von Klassen und Attributen auf Tabellen und Spalten
- Formulierung und Ausführung von SQL-Befehlen
- Transaktionssteuerung
- Verwaltung von Relationen mit Navigation darüber
- Abbildung von Vererbungsbeziehungen
- Generierung technischer IDs
- Komfortable, objektorientierte Suchmöglichkeiten
- Caching

3.1.3 Entwicklung des Standards

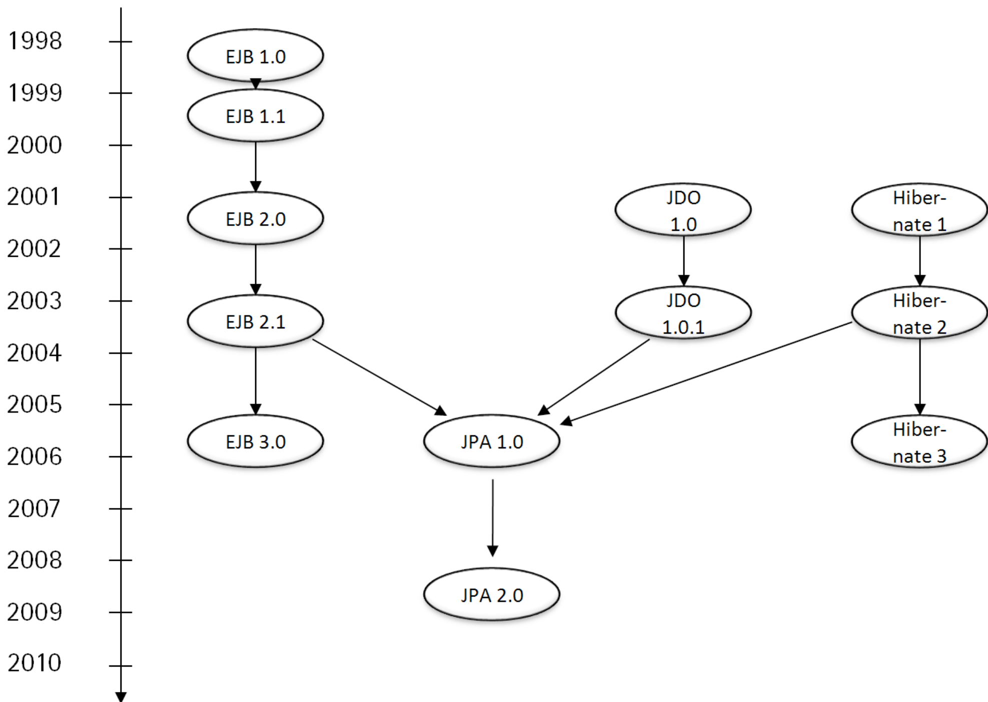


Abbildung 3.2: Historie von Java Persistence

Der Bedarf an Frameworks zur Abbildung von Java-Objekten auf Datenbankeinträge ist immer schon groß gewesen. So hatten sich bis zur Mitte der vergangenen Dekade mehrere Alternativen entwickelt, u. a. die in Abbildung 3.2 gezeigten Spezifikations- bzw. Produktstränge EJB, JDO und Hibernate. War man bis zu dem Zeitpunkt in kontroverse, teilweise schon als religiös zu bezeichnende Diskussionen verstrickt, haben sich die verschiedenen Teams dann doch zusammengerauft und den gemeinsamen Standard JPA 1.0 als Teil der Java EE 5 aus der Taufe gehoben. JPA wurde damals gemeinsam mit EJB 3.0 im JSR 220 entwickelt. Seit JPA 2.0 wird die Weiterentwicklung unabhängig von EJB vorgenommen: JSR 317 für 2.0, JSR 338 für die kommende Version 2.1.

3.1.4 Architektur von Anwendungen auf Basis von JPA

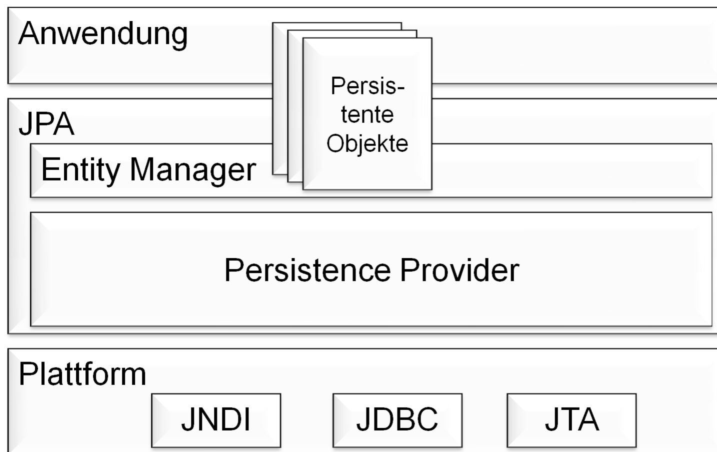


Abbildung 3.3: 10 000-m-Sicht auf JPA-Anwendungen

Nutzt eine Anwendung Java Persistence, so kommuniziert sie mit dem JPA-Framework im Wesentlichen mithilfe des dort vorhandenen Entity Managers (Abb. 3.3). Die zu verarbeitenden Daten werden als persistente Objekte ausgetauscht, die sich – wie später deutlich werden wird – nur wenig von normalen Java-Objekten unterscheiden.

Der Entity Manager ist technisch als Java-Interface ausgebildet. Die konkrete Implementierung ist nicht in JPA selbst enthalten, sondern wird von einem Persistence Provider beigesteuert. Dafür stehen u. a. folgende Produkte zur Verfügung:

- Eclipselink (Referenzimplementierung)[2]
- Hibernate[3]
- OpenJPA[4]

Als Plattform reicht JPA eine Java-SE-Umgebung. Es nutzt dann JDBC und bietet eigene Methoden zur Initialisierung und Transaktionssteuerung an. In einem Applikationsserver nutzt JPA die darin angebotenen Infrastrukturdienste wie Datasources oder globale Transaktionssteuerung. Das Buch fokussiert hauptsächlich auf die serverseitige Nutzung.

Die Java-EE-Applikationsserver haben verpflichtend einen eingebauten JPA-Provider. Im Fall von GlassFish wird Eclipselink verwendet, JBoss nutzt Hibernate, Geronimo wird mit OpenJPA ausgeliefert und WebSphere wie auch WebLogic nutzen eine von OpenJPA abgeleitete Implementierung.

2 *http://www.eclipse.org/eclipselink/*
3 *http://www.hibernate.org/*
4 *http://openjpa.apache.org/*

3.2 Die Basics

Vor der Benutzung von JPA sind nur wenige Hürden zu nehmen, da für die meisten Konfigurationsparameter sinnvolle Vorgabewerte existieren. Das in der Java EE häufig anzutreffende Prinzip *Convention over Configuration* findet auch hier Anwendung: Eine explizite Konfiguration ist nur dann nötig, wenn von den Vorgaben abgewichen werden soll. Lassen Sie uns im Folgenden einen Blick auf die Grundlagen der Nutzung von JPA werfen – zunächst anhand einfacher persistenter Objekte. Relationen, Vererbung und andere Spezialitäten heben wir uns für später auf.

3.2.1 Entity-Klassen

Entities, d. h. persistente Objekte im Sinne von JPA, sind – Sie ahnen es schon – POJOs. Entity-Klassen sind also gemäß dieser im Java-EE-Umfeld allgegenwärtigen Bezeichnung nichts Besonderes, haben insbesondere keine vorgeschriebene Basisklasse und müssen kein bestimmtes Interface implementieren. Einzige Voraussetzung ist die Existenz eines parameterlosen Konstruktors. Mithilfe von Annotationen aus dem Paket *javax.persistence* oder alternativ mittels eines XML-Deskriptors werden den Klassen Metainformationen für die Nutzung durch JPA mitgegeben. Vieles davon ist optional, sodass die Klasse *Country* in Listing 3.3 bereits eine ausreichend ausformulierte persistente Klasse darstellt.

```
@Entity
@Access(AccessType.FIELD)
public class Country
{
  @Id
  private String isoCode;
  private String name;
  private String phonePrefix;
  private String carCode;

  // Getter und Setter nach Bedarf …
}
```

Listing 3.3: Entity-Klasse mit Field Access

Im Vergleich zu einer normalen Java-Klasse fallen nur die drei Annotationen auf: *@Entity*[5] markiert die Klasse als Entity, *@Access* wählt das Attributzugriffsverfahren (s. u.) und *@Id* bestimmt das identifizierende Attribut – gleichbedeutend mit dem Primärschlüssel in der Datenbank.

5 Alle hier verwendeten Klassen stammen aus dem Paket *javax.persistence* und seinen Unterpaketen.

Soll die Klasse parameterbehaftete Konstruktoren enthalten, muss darauf geachtet werden, dass auch ein parameterloser Konstruktor definiert wird. Es reicht, wenn dieser *protected* ist.

Die Annotation *@Id* am Feld *isoCode* bewirkt, dass beim Schreiben eines Eintrags in die Datenbank der Primärschlüsselwert direkt aus der Instanzvariablen herausgelesen wird. Umgekehrt wird ein aus der Datenbank gelesener Wert direkt dort hineingeschrieben. Mehr noch: Dieses Field Access genannteVerfahren gilt automatisch auch für alle weiteren Felder der Klasse. Die Variablen dürfen durchaus *private* sein.

Alternativ kann der JPA-Zugriff auf Entity-Objekte auch über Getter bzw. Setter geschehen. Dazu müssen statt der Instanzvariablen die Getter annotiert werden (Listing 3.4).

```
@Entity
@Access(AccessType.PROPERTY)
public class Country
{
  private String isoCode;
  // …

  @Id
  public String getIsoCode()
  {
    return this.isoCode;
  }
  protected void setIsoCode(String isoCode)
  {
    this.isoCode = isoCode;
  }
  // …
}
```

Listing 3.4: Entity-Klasse mit Property Access

Auch hier gilt dieser Property Access automatisch für alle Getter/Setter-Paare der Klasse. Die Methoden müssen nicht *public* sein, *protected* reicht.

Das Attributzugriffsverfahren wird mit der Annotation *@Access* explizit gewählt. Das kann wie gezeigt auf Klassenebene geschehen wie auch bei einzelnen Attributen. Eine Vermischung von Field Access und Property Access ist damit möglich, aber nicht empfehlenswert. Ohne *@Access* ergibt sich das Verfahren implizit durch die Platzierung der Attributannotationen, wobei dann keine Vermischung innerhalb einer Entity erlaubt ist.

Ob Sie Field Access oder Property Access einsetzen, ist Ihnen vollkommen freigestellt. Mir gefällt Ersteres besser, da die Persistenzannotationen – wenn es denn später mehr als nur *@Id* sind – nicht so verstreut werden und Getter und Setter nur bei fachlichem Bedarf vorgehalten werden müssen. Beim Property Access ist andererseits positiv, dass durch den Methodenzugriff eine Entkopplung von der reinen Datenstruktur stattfindet.

3.2.2 Konfiguration der Persistence Unit

Ganz ohne XML geht es nicht: Ein Deskriptor namens *META-INF/persistence.xml* ist notwendig, um die Verbindung zur Datenbank zu spezifizieren. Listing 3.5 zeigt ein Beispiel dafür.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/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">
  <persistence-unit name="ee_demos">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>jdbc/ee_demos</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>
```

Listing 3.5: Persistenz-Deskriptor

Der Persistenz-Deskriptor deklariert eine oder mehrere Persistence Units, deren Namen – im Beispiel *ee_demos* – frei gewählt werden können. Die Angabe des Persistenzproviders mit dem Element *<provider>* ist optional. Wird sie weggelassen, wird der Standardprovider des Zielservers verwendet, im JBoss 7 also z. B. Hibernate. Die einzige verpflichtende Angabe ist die der zu nutzenden Datasource. Hier muss der JNDI-Name einer im Server passend konfigurierten Datasource angegeben werden.

Mithilfe der Properties können spezielle Eigenschaften des verwendeten Providers konfiguriert werden. Die oben angegebene Property *hibernate.hbm2ddl.auto* kann bspw. genutzt werden, um Hibernate zu erlauben, Schemaänderungen in der Datenbank vorzunehmen. Der Wert *update* wird den Entwicklern unter uns gut gefallen, während die DB-Administratoren ein mulmiges Gefühl beschleicht: Hier folgt das DB-Schema der Klassenstruktur der Entities, beim Start der Anwendung nicht vorhandene Tabellen oder Spalten werden automatisch angelegt. Das ist ein idealer Modus in den Entwicklungsphasen einer Anwendung. Und bevor die DB-Administratoren sich schaudernd abwenden, stellen wir die Property für ein Release der Anwendung auf *verify*. Dann prüft Hibernate beim Start nur noch, ob die Struktur passt. Es stehen noch weitere Modi zur Verfügung, und auch andere Provider haben ein solches Feature. Schauen Sie bei Bedarf in die entsprechende Dokumentation.

An dieser Stelle sei darauf hingewiesen, dass hier zunächst nur der Einsatz von JPA innerhalb einer gemanagten Umgebung beschrieben wird, d. h. innerhalb eines Applikationsservers, der zumindest CDI beherrscht. Am Ende des Kapitels finden Sie Informationen darüber, wie JPA in SE-Anwendungen genutzt werden kann.

Der Deskriptor hat noch eine weitere Bedeutung: Er bestimmt die Klassen, die JPA berücksichtigt, die also Teil der Persistence Unit sind. Das sind alle Klassen mit entsprechender Annotation, die in einem Classpath-Verzeichnis oder einem Jar-File liegen, das in seinem Verzeichnis *META-INF* den Deskriptor *persistence.xml* enthält. In einer Webanwendung muss der Deskriptor also in *WEB-INF/classes/META-INF* platziert werden, damit die in *WEB-INF/classes* abgelegten Klassen für JPA Berücksichtigung finden. In Abbildung 3.4 ist das beispielhaft dargestellt.

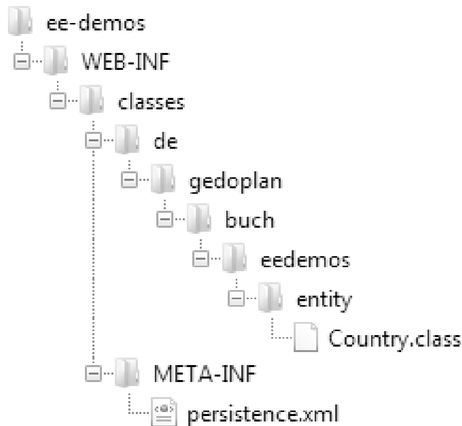


Abbildung 3.4: Persistence Unit in einer Webanwendung

Mithilfe weiterer Elemente lässt sich der Geltungsbereich der *persistence.xml* einschränken oder auch auf weitere Jar-Files ausdehnen. Details dazu findet man in Kapitel 8 (Entity Packaging) der Java-Persistence-Spezifikation.

Zusätzlich zum Persistenz-Deskriptor kann man im Verzeichnis *META-INF* auch die Mapping-Datei *orm.xml* ablegen, wenn man statt mit Annotationen mit XML arbeiten möchte. Damit könnte man z. B. das oben gezeigte Mapping der Klasse *Country* ohne Annotationen definieren, siehe Listing 3.6.

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
  http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
  version="2.0">
  <entity class="de.gedoplan.buch.eedemos.entity.Country" access="FIELD">
    <attributes>
      <id name="isoCode"/>
    </attributes>
  </entity>
</entity-mappings>
```

Listing 3.6: Mapping-Datei

Weitere Mapping-Dateien können mithilfe von *<mapping-file>*-Elementen in *persistence.xml* angemeldet werden. Die Nutzung von XML-Mappings anstelle der entsprechenden Annotationen ist dann nützlich, wenn man den Java-Quellcode der betreffenden Klassen nicht mit Annotationen versehen möchte oder kann, bspw. weil man nicht über die Quellen verfügt. Ansonsten ist die Verwendung von Annotationen natürlich einfacher und auch fehlersicherer.

3.2.3 CRUD

Zur Arbeit mit persistenten Objekten stellt JPA das Interface *EntityManager* zur Verfügung. Objekte dieses Typs übernehmen die wesentliche Arbeit beim Laden, Speichern und Suchen von Datenbankeinträgen. Die konkrete Implementierung wird dabei vom benutzten JPA-Provider beigesteuert.

Eine Entity-Manager-Instanz kann man sich auf einfache Weise per Injektion zur Verfügung stellen lassen. Dazu dient die Annotation *@PersistenceContext*, der der Name der Persistence Unit (aus *persistence.xml*) als Parameter übergeben wird. Der Codeausschnitt in Listing 3.7 referenziert die in Listing 3.5 konfigurierte Persistence Unit *ee_demos*.

```
@PersistenceContext(unitName = "ee_demos")
private EntityManager entityManager;
```

Listing 3.7: Bereitstellung einer Entity-Manager-Instanz per Injektion

Der Parameter kann entfallen, wenn nur genau eine Persistence Unit in *persistence.xml* deklariert wurde.

Nachdem nun alles soweit vorbereitet ist, sind die grundlegenden Operationen zum Erzeugen, Lesen, Ändern und Löschen von persistenten Einträgen trivial: *EntityManager* bietet dazu entsprechenden Methoden an – siehe Listing 3.8.

```
void persist(Object entity)
<T> T find(Class<T> entityClass, Object primaryKey)
void refresh(Object entity)
void remove(Object entity)
```

Listing 3.8: EntityManager-Basisoperationen

Diese Operationen arbeiten allerdings auf einer anderen Ebene als man das z. B. von der direkten Ausführung von SQL-Befehlen mittels JDBC gewohnt ist: Der Zusammenhang zwischen einer Entity-Manager-Methode und der zugehörigen DB-Operation ist eher mittelbar. So führt der Aufruf von *persist* zwar schlussendlich zu einem neuen Eintrag in der Datenbank, die Ausführung des entsprechenden *INSERT*-Befehls kann aber (und wird in der Regel) verzögert geschehen, im Extremfall am Ende der Transaktion.

Womit wir schon beim Thema Transaktionen wären. JPA integriert sich dazu praktischerweise in das Transaktionssystem des Servers: Ein mit dem oben gezeigten Code injizierter Entity Manager ist automatisch transaktionsgebunden. Das bedeutet, dass er ohne weiteres Zutun eine JTA-Transaktion des Servers übernimmt und sich beim Transaktionsende passend verhält: Ein Commit führt (spätestens) zur Ausführung der notwendigen SQL-Befehle, ein Rollback dagegen zum Verwerfen eventueller Änderungen.

Die schreibenden Operationen wie *persist* und *remove* verlangen bei einem transaktionsgebundenen Entity Manager zwingend eine aktive Transaktion, die z. B. mithilfe der im vorigen Kapitel vorgestellten Annotation *@TransactionRequired* gestartet werden kann. Lesende Operationen wie *find* benötigen nicht unbedingt eine aktive Transaktion.

find sucht einen Eintrag anhand seiner ID. Auch hier ist der Zusammenhang zum analogen SQL-*SELECT* recht lose: Der Entity Manager führt einen sog. First Level Cache, in dem sich alle persistenten Objekte befinden, die der Entity Manager während seiner Lebenszeit „gesehen" hat. Man kann sich diesen Cache als *Map* vorstellen, deren Schlüssel die IDs und deren Werte die Entity-Objekte sind. Befindet sich das gesuchte Objekt bereits im Cache, liefert *find* es direkt zurück, ohne erneut die Datenbank zu befragen. Ein explizites Neuladen eines zuvor gelesenen Objektes kann mit *refresh* ausgelöst werden. Bei einem transaktionsgebundenen Entity Manager wird der Cache beim Transaktionsende geleert.

Die Methode *remove* erwartet als Parameter ein bereits gemanagtes Objekt. Das bedeutet, dass das zu löschende Objekt zuvor z. B. mittels *find* eingelesen worden sein muss. Später werden die sog. Bulk Operations beschrieben, mit denen u. a. mehrere Einträge auf einmal ohne vorheriges Einlesen gelöscht werden können.

Die bisher genannten Entity-Manager-Operationen sind zwar nur Einzeiler, dennoch erweist es sich in der Praxis als positiv, sie nicht an den benötigten Stellen der Geschäftslogik jeweils auszuprogrammieren, sondern sie in einer separaten Klasse zu sammeln, die das logische Konzept einer Datenablage – eines Repositories – modelliert. Für die anfangs eingeführte Beispiel-Entity *Country* zeigt Listing 3.9, wie diese Helferklasse aussehen könnte:

```
public class CountryRepository
{
  @PersistenceContext(name = "ee_demos")
  private EntityManager entityManager;

  @TransactionRequired
  public void insert(Country country)
  {
    this.entityManager.persist(country);
  }

  public Country findById(String id)
  {
    return this.entityManager.find(Country.class, id);
  }
```

```
    @TransactionRequired
    public void delete(String id)
    {
      Country country = findById(id);
      if (country != null)
      {
        this.entityManager.remove(country);
      }
    }
  }
```

Listing 3.9: Repository-Klasse

Vermissen Sie etwas? Ja – *Create*, *Read*, *Delete* haben wir gesehen – und was ist mit *Update*? Dafür gibt es interessanterweise keine explizite Entity-Manager-Methode. Vielmehr werden alle Änderungen an Objekten, die der Entity Manager derzeit managt, automatisch spätestens bei einem Commit der Transaktion in die Datenbank gesichert.

Dieses transparente Update lässt sich sogar noch umfassender ausnutzen, wenn man statt eines transaktionsgebundenen Entity Managers einen verwendet, der seine Entity-Objekte auch nach dem Commit weiter managt. Solchen *Extended* oder *Application Managed* Entity Managern wird später in diesem Kapitel ein extra Abschnitt gewidmet.

Wie zuvor erwähnt, werden Änderungen in der Datenbank in aller Regel verzögert durchgeführt, um so mehrere Operationen zusammenzufassen oder sie im Fall eines Rollbacks komplett unterdrücken zu können. Dieser sog. Flush passiert spätestens am Transaktionsende, kann aber auch mithilfe der gleichnamigen Methode explizit ausgelöst werden. Zudem werden die Änderungen normalerweise vor Ausführung einer Query zurückgeschrieben.

3.2.4 Detached Objects

Frühere Persistenzframeworks hatten oftmals die Eigenschaft, dass die persistenten Objekte nur im Kontext des Frameworks lebensfähig waren. So ließen sich bspw. Entity Beans nur innerhalb des EJB-Containers nutzen. Die übliche Antwort der Entwickler auf solche Unzulänglichkeiten ist häufig die Postulierung von Patterns, so auch hier: Das Entwurfsmuster Data Transfer Object (oder auch Value Object) wurde genutzt, um die persistenten Daten auch außerhalb des Persistenzkontextes bereitstellen zu können. Es bedeutete im Wesentlichen die Bereitstellung einer zusätzlichen Klasse mit den gleichen inhaltlichen Eigenschaften wie die der persistenten Klasse, deren Objekte als Transportbehälter genutzt wurden: Im Persistenzkontext gelesene Daten wurden, in entsprechende Data Transfer Objects umkopiert, den Aufrufern zur Verfügung gestellt. Analog mussten die in den Clients veränderten Werte wieder in persistente Objekte kopiert werden, bevor die Ablage in der Datenbank stattfinden konnte.

Dieses zwar nicht komplizierte, aber doch aufwändige Anti-Pattern ist mit JPA obsolet geworden, da JPA-Entities auch ohne Entity Manager lebensfähig sind – es sind ja einfache POJOs. Ihre Anreicherung um Metadaten in Form von Annotationen oder XML ist dabei überhaupt nicht hinderlich. Die JPA-Entities können also direkt im Sinne der alten Data Transfer Objects verwendet werden – eine Kopie ist nicht mehr nötig.

Das Herauslösen persistenter Objekte aus dem Persistenzkontext nennt sich Detachment. Detached Objects unterscheiden sich von Persistent Objects nur dahingehend, dass es für sie keine direkte Verbindung zur Datenbank mehr gibt. Der Entity Manager hat sie sozusagen „vergessen“.

Persistente Objekte lassen sich auf verschiedene Weisen implizit oder explizit detachen: Mithilfe der Entity-Manager-Methode *detach* kann ein einzelnes Objekt aus dem Entity Manager herausgelöst werden. *clear* tut dies für alle vom Entity Manager verwalteten Objekte. Das passiert auch beim Schließen des Entity Managers mit *close* und bei einem Transaktions-Rollback. Ebenfalls *detached* sind Kopien persistenter Objekte, seien sie durch Klonen oder per Serialisierung und Deserialisierung erzeugt. Letzteres ist z. B der Fall, wenn Daten an Remote Clients geliefert werden.

Wichtig ist hier das Verständnis, dass die Detached Objects ganz normale Java-Objekte sind, bei denen es zwar keine besondere Beziehung zur Datenbank mehr gibt, die aber ansonsten beliebig verwendet, verändert oder anderweitig genutzt werden können.

Detached Objects können dem Entity Manager wieder hinzugefügt werden, und zwar mit der Methode *merge*. Dabei ist zu beachten, dass das übergebene Objekt weiterhin *detached* bleibt und als Aufrufergebnis ein inhaltlich gleiches Objekt zurückgegeben wird, das persistent ist.

3.2.5 Entity-Lebenszyklus

JPA-Entities können bezüglich des Entity Managers in verschiedenen Zuständen sein:

- **Transient:** Diese Objekte sind neu oder auf andere Weise ohne Zutun des Entity Managers erzeugt worden. Datenbankeinträge dazu existieren noch nicht.
- **Persistent:** Solche Objekte werden vom Entity Manager gemanagt. Die zugehörigen Datenbankeinträge sind vorhanden und folgen den Statusänderungen der Objekte spätestens zum Transaktionsende.
- **Detached:** Aus dem Persistenzkontext herausgelöste Objekte. Passende Datenbankeinträge sind vorhanden, Objektänderungen werden aber nicht dorthin propagiert.

Durch Benutzung der Entity-Manager-Methoden wechseln Objekte ihren Zustand, wodurch sich der in Abbildung 3.5 dargestellte Lebenszyklus der Entity-Objekte ergibt.

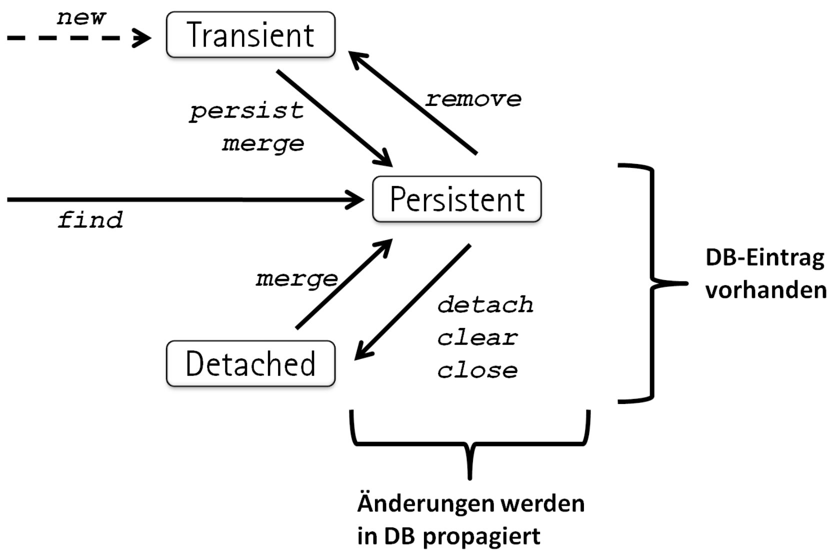


Abbildung 3.5: Entity-Lebenszyklus

Einige Statusübergänge sind aus Übersichtlichkeitsgründen weggelassen worden. So dürfen z. B. *persist* und *merge* auch auf bereits persistente Objekte angewandt werden. Dabei passiert bis auf die später erläuterte Kaskadierung von Operationen nichts.

Eine genaue Beschreibung der Übergänge findet sich in der JPA-Spezifikation im Abschnitt 3.2 (Entity Instance's Life Cycle). Dort wird zudem noch der Zustand *Removed* beschrieben, der hier – ebenfalls der Einfachheit halber – mit *Transient* vermischt wurde.

3.2.6 Mapping-Annotationen für einfache Objekte

Die bisherigen Beispiele haben weitgehend von den Default-Einstellungen Gebrauch gemacht. Lassen Sie uns nun einen Blick auf weitere Möglichkeiten des Persistenz-Mappings einer Java-Klasse werfen. Hier soll allerdings nicht der entsprechende Abschnitt der JPA-Spezifikation dupliziert werden, daher beschränkt sich das Folgende auf die häufig genutzten Parameter. Weitere Details finden Sie in Abschnitt 11 (Metadata for Object/Relational Mapping) der Spezifikation.

- *@Entity*

Die auf Klassenebene angewandte Annotation *@Entity* akzeptiert als Parameter den sog. Entity-Namen. Er wird später benötigt, um Abfragen mithilfe von JPQL formulieren zu können. Wird er nicht angegeben, gilt der einfache Klassenname als Entity-Name.

- *@Table*

Ebenfalls auf Klassenebene kann die Annotation *@Table* verwendet werden. Mithilfe des Parameters *name* (und bei Bedarf *catalog* und *schema*) kann der Name der Tabelle in der

Datenbank bestimmt werden. Die Voreinstellung dafür ist der Entity-Name. Der Parameter *uniqueConstraints* erlaubt – wie der Name schon sagt – die Angabe von Unique Constraints. Er wirkt sich aber nur aus, wenn der Provider die Tabelle anlegt, wenn also bspw. Hibernate wie oben erwähnt mit *hibernate.hbm2ddl.auto=update* konfiguriert ist. Zudem ist hier zu beachten, dass die Contraints auf Basis der Spaltennamen angelegt werden, nicht etwa auf Basis der Java-Attributnamen.

Listing 3.10 zeigt, wie die schon mehrfach bemühte Beispielklasse mit einem Entity-Namen versehen wird – der allerdings dem Vorgabewert entspricht – und wie die zugehörige Tabelle explizit benannt und mit zwei Unique Contraints belegt wird.

```
@Entity(name="Country")
@Access(AccessType.FIELD)
@Table(name = "EEDEMOS_COUNTRY", uniqueConstraints = {
  @UniqueConstraint(columnNames = "PHONEPREFIX"),
  @UniqueConstraint(columnNames = "CARCODE") })
public class Country
{
  …
```

Listing 3.10: Anwendung von „@Entity“ und „@Table“

Ist es sinnvoll, Entity-Namen, Tabellennamen (und später auch Spaltennamen) explizit vorzugeben, oder sollte man auf den Default-Entity-Namen = einfacher Klassenname = Tabellenname vertrauen? Nun, das kommt drauf an. Zum einen sparen die Vorgabewerte natürlich Tipparbeit und man darf sich durchaus eine gewisse sinnvolle Grundfaulheit aneignen, um seine Produktivität zu steigern. Auf der anderen Seite kann man mit den Default-Werten recht leicht in Refactoring-Fallen tappen. Stellen Sie sich z. B. vor, dass der Name der Beispielklasse von *Country* nach *Land* geändert werden soll, weil die allgemeinen Entwicklungsrichtlinien seit Neuestem deutsche Namen für Identifier (pardon: Identifizierer) verlangen. So ein Refactoring ist in den aktuellen IDEs ein Klacks, aber Achtung: Bei Verwendung der Defaults müssen die Query-Texte und der Tabellenname in der Datenbank passend verändert werden! Das würde sich mit expliziten Angaben vermeiden lassen.

Die im Weiteren besprochenen Mapping-Annotationen gelten für die Attributebene – also bei Field Access für die Instanzvariablen und bei Property Access für die Getter.

- *@Column*

Hiermit wird analog zu *@Table* die Angabe von Eigenschaften der Tabellenspalte ermöglicht. Neben dem Parameter *name*, der den Spaltennamen annimmt, akzeptiert *@Column* diverse Parameter zur Beeinflussung der Spaltendeklaration, die sich allerdings wieder nur dann auswirken, wenn der Provider die Tabelle in der Datenbank erzeugt. Details finden Sie in der JPA-Spezifikation.

Der Boole'sche Parameter *unique* erlaubt die Anlage von Unique Contraints zusätzlich zu denen auf Tabellenebene.

Die ebenfalls Boole'schen Parameter *insertable* und *updatable* lassen gewisse Trickserei-en zu: Stehen diese Werte auf *false*, wird das betroffene Attribut nicht in *INSERT*- bzw. *UPDATE*-Statements verwendet. Das lässt sich z. B. nutzen, um mehrere Attribute auf die gleiche Tabellenspalte abzubilden, von denen dann bei Schreibzugriffen nur eines verwendet wird. Anwendungen dazu finden sich bspw. bei den später besprochenen Relationen, um den dabei implizit vorhandenen Foreign Key separat (und nur zum Lesen) zur Verfügung zu stellen.

Schließlich ermöglicht der Parameter *table* die Platzierung der Spalte in eine Secondary Table. Die Beschreibung dazu folgt später.

- *@Enumerated*

Für Aufzählungstypen gibt es zwei Möglichkeiten der Ablage der Werte in der Datenbank. Zum einen kann der numerische Ordnungswert verwendet werden, also *0* für die erste Konstante im Aufzählungstyp, *1* für die zweite usw. Die zweite Möglichkeit legt den Namen der abzuspeichernden Konstanten als Text in der Datenbank ab. Zur Auswahl übergibt man *@Enumerated* eine Konstante aus der Aufzählung *EnumType: ORDINAL* bzw. *STRING*. Voreingestellt ist *ORDINAL*.

Das numerische Verfahren birgt eine große Gefahr, und zwar wieder einmal bei einem Refactoring. Nehmen wir an, wir haben einen Aufzählungstyp *AmpelStatus*, wie in Listing 3.11. Unser Verkehrsleitsystem wird um 3:00 abgeschaltet, wenn alle Ampeln gerade auf *ROT* stehen. Das Softwareupdate, das wir dann aktivieren, enthält u. a. die refaktorierte Klasse *AmpelStatus*, in der die Farbkonstanten aus irgend einem Grund in die Reihenfolge *ROT, GELB, GRUEN* gebracht worden sind. Die ehemals roten Ampeln sind nun plötzlich grün (Ordnungswert *2*). Hoffentlich geht das gut …

```
public enum AmpelStatus
{
  GRUEN,
  GELB,
  ROT
}


@Entity
public class Verkehrsleitsystem
{
  …
  private AmpelStatus ampel1;
  private AmpelStatus ampel2;
  …
```

Listing 3.11: Aufzählungstyp und seine Verwendung in einer Entity

- *@Lob*

Je nach dem unterliegenden Datentyp lassen sich hiermit BLOBs und CLOBs deklarieren. Ist das betroffene Attribut vom Typ *char[], Character[]* oder *String*, wird der Wert in der Datenbank als CLOB behandelt. Andere Typen werden als BLOB verarbeitet.

- *@Transient*

Die Voreinstellung ist so, dass alle Attribute der Entity-Klasse persistent sind. Will man davon abweichen, muss das betroffene Attribut mit *@Transient* markiert werden. Es ist dann nicht Teil der Datenbanktabelle, wird also auch von allen entsprechenden Operationen ignoriert.

- *@Temporal*

Beim Mapping eines Attributs vom Typ *java.util.Date* oder *java.util.Calendar* sind mehrere Abbildungen auf die Datenbank möglich: Ablage des kompletten Werts inkl. Datum und Uhrzeit oder Speicherung von Datum bzw. Uhrzeit alleine. Zur Auswahl akzeptiert *@Temporal* einen Wert aus dem Aufzählungstyp *TemporalType: TIMESTAMP, DATE* bzw. *TIME*.

Wird *@Temporal* nicht verwendet, wird der komplette Wert abgelegt.

Bei der Arbeit mit Zeiten ist zu beachten, dass die Granularität der Werte nicht mit der der Datenbankspalte übereinstimmen muss. Während die Java-Werte bis zur Millisekunde genau sind, speichert die Datenbank ggf. gröber oder feiner ab.

Nachdem wir nun die einfachen Attribute betrachtet haben, können wir uns langsam an etwas komplexere Dinge wagen: Attribute, die aus mehreren Teilelementen zusammengesetzt sind.

- *@Embeddable*

Wir sind es gewohnt, Objekte aus anderen Objekten zu komponieren, um so für eine übersichtliche Struktur zu sorgen und Datentypen ggf. auch wiederverwerten zu können. Listing 3.12 zeigt eine solche Struktur: Der Datentyp *Address*, bestehend aus den üblichen Adressfeldern, wird als Typ eines Attributs in *Employee* verwendet.

Damit das nach Wunsch funktioniert, muss *Address* mit *@Embeddable* annotiert sein. Address muss (wie zuvor die Entities) einen parameterlosen Konstruktor vorweisen und kann mit den bereits besprochenen Annotationen weiter parametrisiert werden. Es ergibt sich daraus ein Mapping der Klasse *Employee* auf eine Tabelle, die für das Attribut *address* drei Spalten enthält (Abb. 3.6).

Die Spaltennamen ergeben sich dabei aus den Vorgaben oder Annotationen der eingebetteten Klasse, im Beispiel sind dies also *zipCode, town* und *street*. Das lässt sich an der einbettenden Stelle mithilfe der Annotationen *@AttributeOverrides* und *@AttributeOverride* erreichen, siehe Listing 3.13 – eine zugegebenermaßen grenzwertige Konstruktion im Hinblick auf die Lesbarkeit des Programmcodes …

```
@Embeddable
@Access(AccessType.FIELD)
public class Address
{
  private String zipCode;
  private String town;
  private String street;
  …
}

@Entity
@Access(AccessType.FIELD)
public class Employee
{
  @Id
  private int     id;
  private String  name;

  private Address address;
  …
```

Listing 3.12: Eingebettetes Objekt

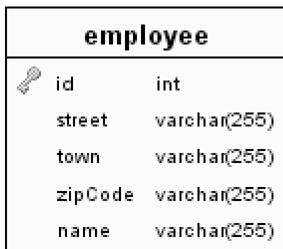


Abbildung 3.6: „Employee"-Tabelle mit eingebetteter „Address"

```
@Entity
@Access(AccessType.FIELD)
public class Employee
{
  …
  @AttributeOverrides({
    @AttributeOverride(name = "zipCode",
                       column = @Column(name = "homeZipCode")),
    @AttributeOverride(name = "town",
                       column = @Column(name = "homeTown")),
    @AttributeOverride(name = "street",
                       column = @Column(name = "homeStreet")) })
  private Address homeAddress;
  …
```

Listing 3.13: Explizite Angabe von Spaltennamen bei einem eingebetteten Attribut

- *@ElementCollection*

Diese Annotation kann einerseits auf Attribute vom Typ *java.util.Collection* oder einem davon abgeleiteten Interface angewendet werden. Der Elementtyp der *Collection* kann ein einfacher Datentyp sein oder ein *@Embeddable*. Die Daten der *Collection* werden in einer separaten Tabelle abgelegt.

Ganz ähnlich funktioniert das auch mit Attributen vom Typ *java.util.Map*. Die Basistypen der *Map* können wieder einfache oder *@Embeddables* sein.

Listing 3.14 zeigt je eine Anwendung von *@ElementCollection*. Die resultierende Tabellenstruktur zeigt Abbildung 3.7.

```
@Entity
@Access(AccessType.FIELD)
public class Employee
{
  …
  @ElementCollection
  private List<String> skills;

  @ElementCollection
  private Map<String, String> phones;
  …
```

Listing 3.14: Collection-Attribute

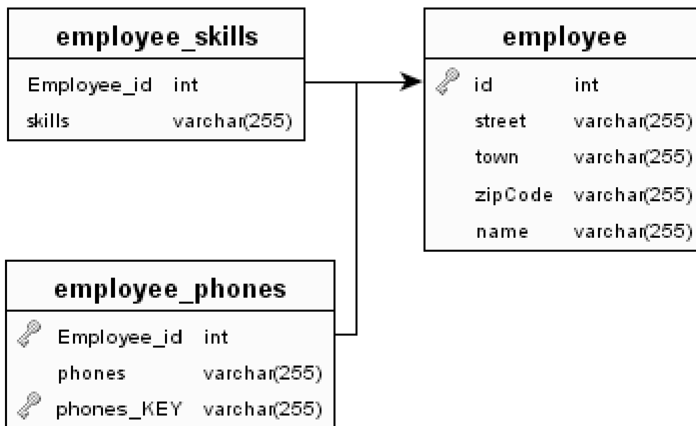


Abbildung 3.7: Ablage von Collection-Attributen in zusätzlichen Tabellen

Der Name der Zusatztabelle ergibt sich standardmäßig als Verkettung des Namens der Haupttabelle, einem "_" und dem Namen des *Collection-* bzw. *Map-*Attributs. Das kann aber mithilfe der Annotation *@CollectionTable* übersteuert werden. Damit können zudem die Namen der Fremdschlüssel der Zusatztabelle bestimmt werden, die per Default aus

dem Zieltabellennamen, "_" und dem Namen des referenzierten Attributs zusammengesetzt werden. Auch die restlichen Spaltennamen der Zusatztabelle können explizit angegeben werden. Dazu dienen die Annotationen *@Column* und *@MapKeyColumn* im Fall einfacher Elementtypen sowie *@AttributOverrides* und *@AttributeOverride* im Fall von *@Embeddable*-Elementen. Die Details dazu finden Sie in der JPA-Spezifikation.

Generell ist zu beachten, dass *@ElementCollection* nur auf Attribute der Interfacetypen aus *java.util* angewandt werden darf. Es darf also statt *List* nicht etwa *ArrayList* benutzt werden. Außerdem empfiehlt sich unbedingt die Nutzung der parametrisierten Typen *java.util.Collection<E>*, da dann der oder die Elementtypen bekannt sind. Bei Verwendung der Raw Types sind zusätzliche Annotationen notwendig, was die Lesbarkeit deutlich verschlechtert.

3.2.7 Generierte IDs

Die bereits mehrfach angeführte Beispielklasse *Country* hat eine fachliche Identität. Das Attribut *isoCode* enthält den international genormten 2-Zeichen-Code, der das entsprechende Land eindeutig identifiziert. Diese Situation finden wir bei Weitem nicht immer vor. Es ist im Gegenteil so, dass aktuelle Softwaredesigns eher technische Identitäten favorisieren. Der Hintergrund ist der, dass man bei fachlichen Identitäten auf längere Sicht nicht sicher sein kann, ob sie wirklich unveränderlich sind. Zudem sind fachliche Attribute oft nur in Kombinationen eindeutig, sodass man sich schnell mit zusammengesetzten IDs konfrontiert sieht.

An eine technische ID haben wir nur den Anspruch, dass sie eindeutig ist. Sie hat keine fachliche Bedeutung, kann also generiert werden. Dies ist mit JPA in einfacher Weise möglich, wenn die Entity ein einzelnes ID-Attribut besitzt. Dieses wird zusätzlich zu *@Id* mit *@GeneratedValue* annotiert. Der JPA-Provider vergibt dann automatisch für jedes neu abgespeicherte Objekt eine neue, eindeutige ID (Listing 3.15).

```
@Entity
@Access(AccessType.FIELD)
public class City
{
  @Id
  @GeneratedValue(strategy=GenerationType.AUTO)
  private Integer id;
  …
```

Listing 3.15: Generierte ID

Zur Generierung der ID stehen drei verschiedene Generatoren zur Auswahl, die mithilfe des Parameters *strategy* ausgewählt werden. Folgende Werte stehen dafür zur Verfügung:

- *GenerationType.IDENTITY*

Bei dieser Strategie wird die Vergabe einer neuen, eindeutigen ID durch die Datenbank vorgenommen. Voraussetzung ist natürlich, dass die eingesetzte DB dies auch unterstützt. So bieten z. B. Derby und MySQL Autoincrement Columns an; MS SQL Server kennt analog Identity Columns. Bei einem *INSERT* weist die Datenbank dem betreffenden Spaltenwert automatisch die nächste freie Nummer zu.

Ein starker Vorteil dieser Strategie ist, dass auch andere Anwendungen, die in die gleiche DB schreiben, ohne weiteres Zutun das gleiche Verfahren nutzen. Nachteilig ist allerdings, dass der generierte Wert erst zum Zeitpunkt der Ausführung des *INSERT*-Statements ermittelt werden kann und er zudem durch einen weiteren DB-Zugriff wieder ausgelesen werden muss, um das Java-Objekt im Speicher entsprechend zu aktualisieren.

- *GenerationType.SEQUENCE*

Hier wird eine Sequence der Datenbank genutzt und damit wiederum ein Konstrukt, dass von der Zieldatenbank angeboten werden muss. Das ist z. B. für Oracle oder PostgreSQL der Fall. Die Sequence liefert bei jedem Zugriff eine neue Zahl, die vom Provider zur Besetzung der ID des neu erzeugten Entity-Objekts verwendet wird.

Vorteilhaft ist hier die Entkopplung vom späteren Einfügen des Satzes in die Datenbank, der ID-Wert steht also schon zum Zeitpunkt des Aufrufs von *persist* zur Verfügung. Außerdem können Datenbanken i. d. R. mit Sequenzen gut optimiert umgehen, benötigen also bspw. auch bei nebenläufigen Zugriffen nicht unbedingt eine Transaktion.

Der Sequence-Generator lässt sich noch weiter parametrisieren, indem man der Annotation *@GeneratedValue* mit dem Parameter *generator* den Namen eines spezifischen Generators mitgibt und diesen mit der zusätzlichen Annotation *@SequenceGenerator* deklariert. Damit wird es möglich, auf die Benennung der Sequence in der Datenbank und ihren Startwert Einfluss zu nehmen (Listing 3.16).

```
@Id
@GeneratedValue(strategy = GenerationType.SEQUENCE,
                generator = "cityGenerator")
@SequenceGenerator(name = "cityGenerator",
                   sequenceName = "CITY_GEN", initialValue = 200000)
private Integer id;
```

Listing 3.16: Konfiguration des Sequence-Generators

- *GenerationType.TABLE*

Der Table-Generator lässt sich mit jeder Datenbank nutzen. Der Provider verwendet eine eigene Tabelle zur Nachbildung einer Sequenz. Im Vergleich zum vorigen Verfahren hat dieses hier den (kleinen) Nachteil, dass für die Verwaltung der Sequenztabelle eine separate Transaktion genutzt werden muss.

Auch der Table-Generator lässt sich mit weiteren Parametern ähnlich zu denen des Sequence-Generators ausstatten, und zwar mithilfe der Annotation *@TableGenerator* (Listing 3.17).

```
@Id
@GeneratedValue(strategy = GenerationType.TABLE,
                generator = "cityGenerator")
@TableGenerator(name = "cityGenerator",
                table = "CITY_GEN",
                pkColumnName = "GENERATOR",
                valueColumnName = "NEXT_ID",
                initialValue = 200000)
private Integer id;
```

Listing 3.17: Konfiguration des Table-Generators

- *GenerationType.AUTO*

Dieser Modus stellt keinen weiteren Generator dar. Vielmehr wird hier dem Provider die Auswahl des vermeintlich günstigsten Generators überlassen. Die meisten Provider lassen die Wahl auf den Table-Generator fallen, was insofern verständlich ist, als dieser Modus mit jeder Datenbank verfügbar ist. Nach meiner Beobachtung ist Hibernate der einzige Provider, der je nach Zieldatenbank unterschiedliche Generatoren nutzt. Der Identity-Generator kommt zum Einsatz, wenn die Datenbank dieses Verfahren unterstützt. Falls das nicht so ist, dafür aber Sequenzen zur Verfügung stehen, wird der Sequence-Generator gewählt. Wenn auch das nicht möglich ist, nutzt Hibernate den Table-Generator. Der Automatik-Modus ist voreingestellt.

Dem Provider kann erlaubt werden, mit einem Generatorzugriff direkt mehrere neue IDs zu besorgen und diese nach und nach zu verbrauchen. Diese sog. Allocation Size kann beim Sequence- und beim Table-Generator konfiguriert werden, und zwar mithilfe des Parameters *allocationSize* der Annotationen *@SequenceGenerator* und *@TableGenerator*. Der Vorgabewert ist lt. Spezifikation *50*, was aber nach meiner Beobachtung nicht von allen Providern eingehalten wird. Eine große Allocation Size wirkt sich günstig auf die Einfügeperformanz aus, wie man der Abbildung 3.8 entnehmen kann. Dort ist die Laufzeit für das Einfügen von 50 000 neuen Einträgen in eine Datenbank für die allocationSize 1, 10, 100 und 1000 qualitativ darstellt. Bei einer Allocation Size größer als 1 bleiben allerdings potenziell generierte Werte ungenutzt.

Für den Identity-Generator gibt es leider keine Möglichkeit, IDs auf Vorrat zu besorgen. Hier müssen die generierten Werte immer sofort zurückgelesen werden.

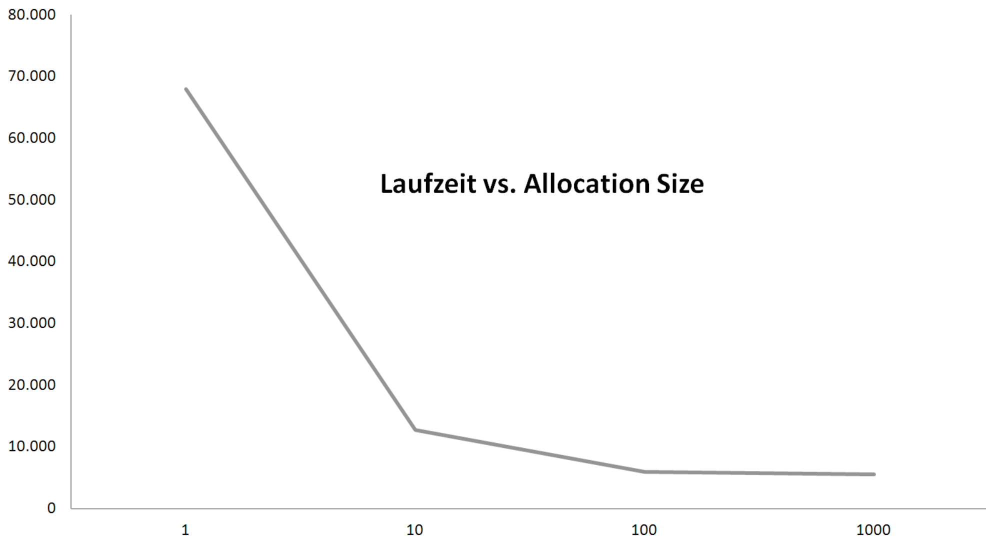


Abbildung 3.8: Einfluss der Allocation Size auf die Laufzeit von Einfügeoperationen

In der aktuellen Version des Standards stehen nur die beschriebenen Generatoren zur Verfügung. Sie benötigen alle ein ganzzahliges ID-Attribut. Die JPA-Spezifikation gibt nicht exakt Auskunft über die verwendbaren Datentypen, enthält aber Beispiele mit *int*, *Integer*, *long* und *Long*. Für die kommende Version ist ein *String*-basierter Generator in der Diskussion, der UUIDs erzeugt.

3.2.8 Objektgleichheit

Die Gleichheit von Java-Objekten wird mithilfe der Methode *equals* definiert. Eng damit verbunden ist *hashCode*: Ist *a.equals(b) true*, muss *a.hashCode()* den gleichen Wert liefern wie *b.hashCode()*.

Es ist eine häufig anzutreffende und gute Programmierkonvention, dass *equals* und *hashCode* in jeder Klasse definiert werden müssen – von einigen technischen Klassen oder Hilfsklassen vielleicht einmal abgesehen.

Wie sollten diese Methoden nun für Entity-Klassen programmiert werden? Java Persistence macht da prinzipiell keine Vorgabe, dennoch lassen sich aus dem Ablagekonzept einer Datenbank Rückschlüsse auf die Objektgleichheit auf der Java-Seite ziehen: In der Datenbank definiert der Primärschlüssel eine Art Gleichheit innerhalb der Einträge einer Tabelle. Vor diesem Hintergrund macht es Sinn, *equals* auf Basis des ID-Attributs zu definieren, also beim Vergleich zweier Entity-Objekte nicht alle Attribute paarweise zu vergleichen, sondern sich auf das ID-Attribut zu beschränken (bzw. die ID-Attribute, wenn es mehrere sein sollten). Für die Country-Klasse könnten *equals* und *hashCode* also wie in

Listing 3.18 definiert werden. Der Aufwand zur Erstellung dieser Methoden ist gering. Bei Nutzung einer aktuellen IDE sind dazu nur wenige Mausklicks nötig.

```
@Entity
@Access(AccessType.FIELD)
public class Country
{
  @Id
  private String isoCode;
  …
  public boolean equals(Object obj)
  {
    if (this == obj)
    {
      return true;
    }
    if (obj == null)
    {
      return false;
    }
    if (getClass() != obj.getClass())
    {
      return false;
    }
    final Country other = (Country) obj;
    return this.isoCode.equals(other.isoCode);
  }

  public int hashCode()
  {
    return this.isoCode.hashCode();
  }
  …
```

Listing 3.18: „equals“ und „hashCode“

Dieses Vorgehen lässt sich natürlich auch bei Klassen mit technischen ID-Attributen anwenden, allerdings gibt es da eine Schwierigkeit, wenn die Attributwerte generiert werden: Die ID-Werte sind frühestens nach dem Aufruf von *persist* gesetzt, evtl. auch erst später, wenn der *INSERT* in die Datenbank erfolgt. Ein Vergleich zweier Entity-Objekte wird also potenziell unterschiedlich ausfallen, wenn er vor oder nach dem Einfügen in die DB erfolgt. Schlimmer noch: Alle noch nicht persistenten Objekte einer Klasse haben ihre ID-Attribute noch nicht gesetzt, d. h. diese sind z. B. alle *null*. Im Sinne einer „normal“ programmierten *equals*-Methode wären diese Objekte dann alle gleich – eine Situation, die vermutlich nicht so ganz im Sinne des Erfinders wäre …

Abhilfe lässt sich auf verschiedene Weise schaffen, wobei es allerdings keinen Königsweg gibt. In jedem Fall wird die Information benötigt, ob die ID des betreffenden Objekts bereits gesetzt worden ist. Im Allgemeinen lässt sich das mithilfe der Methode *getIdentifier*

aus dem Interface *PersistenceUnitUtil* herausfinden. Sie liefert die ID für das übergebene Entity-Objekt zurück. Ist diese noch nicht gesetzt, ist der Returnwert *null*. Ein Objekt vom Typ *PersistenceUnitUtil* kann über zwei Ecken aus dem Entity Manager heraus geliefert werden. Man könnte sich also eine Helfermethode wie die in Listing 3.19 bereitstellen, um den ID-Zustand eines Objekts erfragen zu können. Noch einfacher geht es aber, wenn man für die ID-Attribute keine primitiven Datentypen wählt, sondern objektorientierte, statt *int* also bspw. *Integer*. In dem Fall ist ein noch nicht gesetztes Attribut am Wert *null* zu erkennen. Das hat zudem den Vorteil, dass für die Entscheidung kein Entity Manager benötigt wird, auf den man innerhalb einer Entity-Methode auf einfache Weise keinen Zugriff hat.

```
public boolean isIdSet(Object entity)
{
  PersistenceUnitUtil persistenceUnitUtil
    = entityManager.getEntityManagerFactory().getPersistenceUnitUtil();
  Object id = persistenceUnitUtil.getIdentifier(entity);
  return id != null;
}
```

Listing 3.19: Helfermethode zur Ermittlung, ob die ID eines Objekts gesetzt ist

Nach diesen Vorüberlegungen lässt sich *equals* nun so formulieren, dass für noch nicht mit einer ID versehene Objekte *false* geliefert wird. Der Hashcode solcher Objekte kann z. B. mit *0* angenommen werden (Listing 3.20). Es sei an dieser Stelle darauf hingewiesen, dass dieses Verfahren nicht absolut wasserdicht ist, denn *equals* und *hashCode* werden vor und nach dem Persistieren der Objekte unterschiedliche Ergebnisse liefern. Das ist nur dann tolerierbar, wenn gewährleistet ist, dass frisch persistierte Objekte im weiteren Verlauf des Programms nicht mehr unmittelbar benötigt werden, vor ihrer weiteren Verwendung also neu geladen werden. Das wird bei den allermeisten Geschäftsprozessen so sein.

```
@Entity
@Access(AccessType.FIELD)
public class City
{
  @Id
  @GeneratedValue
  private Integer id;
  …
  public boolean equals(Object obj)
  {
    if (this == obj)
    {
      return true;
    }
    if (obj == null)
    {
      return false;
    }
```

```
    if (getClass() != obj.getClass())
    {
      return false;
    }
    final City other = (City) obj;
    return this.id != null && this.id.equals(other.id);
  }

  public int hashCode()
  {
    return this.id != null ? this.id.hashCode() : 0;
  }
  …
```

Listing 3.20: „equals“ und „hashCode“ bei generierter ID

Auf der sicheren Seite wäre man, wenn man *equals* und *hashCode* im Fall einer noch nicht besetzten ID durch Auswurf einer passenden Exception (*IllegalArgumentException* o. ä.) abbräche. Leider ist das kein wirklich gangbarer Weg, da angrenzende Frameworks (z. B. Bean Validation – siehe gleichnamiges Kapitel) für einige Operationen darauf angewiesen sind, *equals* und *hashCode* schon für transiente Objekte aufrufen zu können.

In der Literatur und in Forenbeiträgen findet man häufig auch die Meinung, man solle Entity-Objekte zusätzlich zur technischen Identität mit einer fachlichen versehen, die man solange für Vergleiche benutzt, bis die technische ID gesetzt ist. Es ist schwer, sich dem anzuschließen, da damit der Sinn einer technischen Identität weitgehend ad absurdum geführt wird. Zudem wäre der Entwicklungsaufwand unvertretbar höher.

3.3 Objektrelationen

Die bisher dargestellten Dinge betrafen einfache Entity-Klassen, die keinerlei Beziehungen zu anderen Entities haben. Im Folgenden wenden wir uns den Relationen zwischen Entities zu. Eine Beziehung drückt sich in der Datenbank durch einen Foreign Key aus, d. h. eine Spalte (oder auch mehrere), deren Wert auf einen Eintrag einer anderen Tabelle referenziert. Auf der Ebene der Java-Klassen werden Beziehungen durch Entity-Attribute ausgedrückt, die ein oder mehrere Objekte der Gegenseite enthalten. Damit wird ein konzeptioneller Unterschied dieser beiden Sichten deutlich: Während es in der Datenbank eine Stelle gibt, die die Beziehung definiert, sind es potenziell zwei Java-Klassen, die Beziehungsattribute enthalten können. Vor der Betrachtung der Details sind drei grundlegende Begriffe in Bezug auf Relationen zu klären.

- Ownership

Eine der beiden an einer Relation beteiligten Entity-Klassen hat bezüglich der Beziehung „den Hut auf“. Das Relationsattribut auf ihrer Seite ist für den Wert des Foreign Keys in der Datenbank maßgeblich. Diese Entity ist der Eigentümer der Relation.

- Kardinalität

Diese Eigenschaft ist Ihnen von der Datenbankseite her bekannt: Es können verschieden viele Einträge der einen Seite mit einer unterschiedlichen Anzahl von Sätzen der anderen Seite verknüpft sein. In diesem Sinne kennen wir 1:1-, 1:n und n:m-Beziehungen. Auf der Java-Seite bestimmt die Kardinalität, ob das Relationsattribut den Typ der Gegenseite hat oder eine Collection darüber ist.

- Direktionalität

Ist in einer Entity-Klasse ein Relationsattribut vorhanden, enthält es das bzw. die durch die Relation referenzierte(n) Element(e). Man kann also von einem Objekt der Klasse durch Benutzung des Attributs sozusagen zur Gegenseite hinübernavigieren. Die Existenz der Relationsattribute auf beiden Seiten der Beziehung ist Ausdruck der Direktionalität: Bei unidirektionalen Relationen ist nur auf der Owner-Seite ein Relationsattribut vorhanden, bei bidirektionalen auf beiden Seiten.

Kombiniert man diese drei grundlegenden Eigenschaften von Relationen, erhält man aufgrund der Symmetrie einiger Kombinationen nicht 12, sondern nur 7 Fälle, die im Folgenden betrachtet werden.

3.3.1 Unidirektionale n:1-Relationen

Hier ist einem Objekt der Owner-Seite der Relation jeweils ein Objekt der Gegenseite zugeordnet. Umgekehrt können es mehrere sein. Da es eine unidirektionale Verbindung ist, enthält nur die Ownerseite ein Relationsattribut. Es hat den Typ der Gegenseite und wird mit *@ManyToOne* annotiert. In Listing 3.21 ist als Beispiel die Verbindung zwischen Flugzeugen und Flügen dargestellt. Ein Flugzeug absolviert mehrere Flüge, wobei hier angenommen wurde, dass im Flug das Flugzeug eingetragen wird, umgekehrt das Flugzeug aber nichts von seinen Flügen „weiß“.

```
@Entity
@Access(AccessType.FIELD)
public class AirCraft
{
  @Id @GeneratedValue
  private Integer id;
  private String  maker;
  …
  }

@Entity
@Access(AccessType.FIELD)
public class Flight
{
  @Id @GeneratedValue
  private Integer  id;
  private int      flightNo;
```

```
  @ManyToOne
  private AirCraft airCraft;
  …
}
```

Listing 3.21: Listing 3.21: Unidirektionale n:1-Beziehung

Die zugeordnete Struktur in der Datenbank zeigt Abbildung 3.9. Die Relation wird durch den Foreign Key *airCraft_id* in der Tabelle *flight* repräsentiert. Der Name der Spalte ergibt sich aus dem Namen des Relationsattributs, einem '_' und dem Namen des referenzierten Primärschlüsselattributs. Mithilfe der Annotation *@JoinColumn* kann ein anderer Name vorgegeben werden (Listing 3.22).

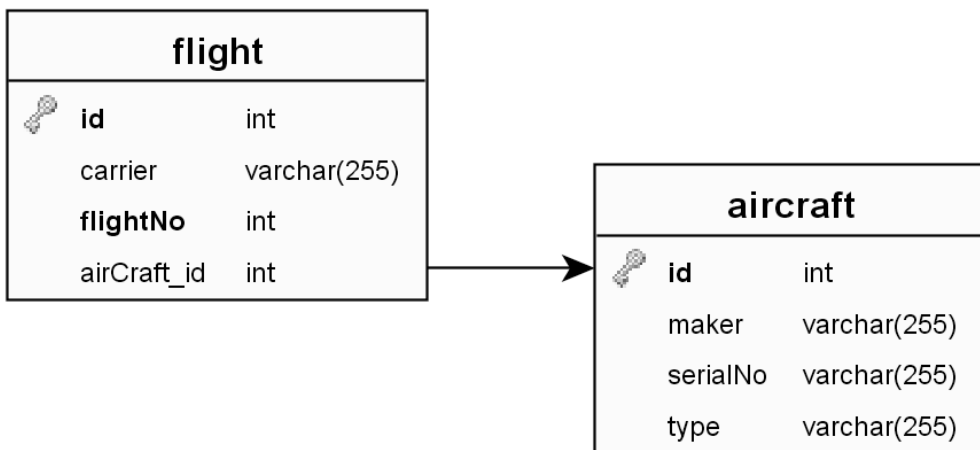


Abbildung 3.9: Tabellenstruktur zu Listing 3.21

```
@ManyToOne
@JoinColumn(name = "PLANE_ID")
private AirCraft airCraft;
```

Listing 3.22: Explizite Benennung des Fremdschlüssels

Die gezeigte Tabellenstruktur ist sicher das, was man regelmäßig für solche Relationen erwartet und was man „auf der grünen Wiese" auch so anlegen würde. Übernimmt man dagegen eine bestehende Datenbankstruktur, ist man vielleicht mit einem Layout wie in Abbildung 3.10 konfrontiert. Hier ist die Beziehung über eine Verknüpfungstabelle abgebildet worden. So etwas lässt sich mit der Annotation *@JoinTable* erreichen (Listing 3.23). Die Parameter der Annotation sind optional. Wenn nicht angegeben, ergeben sich die Namen der Verknüpfungstabelle aus den Namen der verknüpften Tabellen mit einem '_' dazwischen. Die Fremdschlüssel werden als Vorgabe wie oben beschrieben benannt.

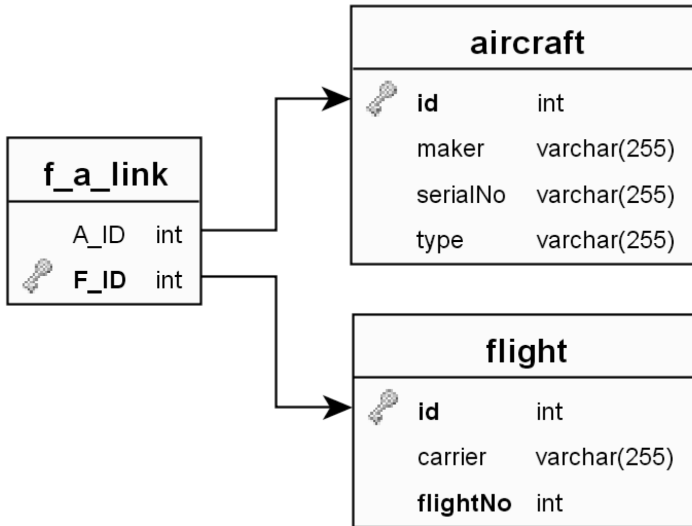


Abbildung 3.10: Alternatives Tabellenlayout mit Verknüpfungstabelle

```
@ManyToOne
@JoinTable(name = "F_A_LINK",
           joinColumns = { @JoinColumn(name = "F_ID") },
           inverseJoinColumns = { @JoinColumn(name = "A_ID") })
private AirCraft airCraft;
```

Listing 3.23: Mapping-Annotationen zu Abbildung 3.10

Die Benutzung der Objektstruktur zur Laufzeit ist trivial: Nach einem Einlesen eines *Flight*-Objekts, z. B. per *find*, enthält das darin befindliche Attribut *airCraft* direkt das zugeordnete *AirCraft*-Objekt. Umgekehrt führt eine Änderung des Attributs – bspw. die Zuweisung eines anderen *AirCraft*s – schließlich zum Eintrag des dazu passenden Werts in das Fremdschlüsselattribut bzw. in die Verknüpfungstabelle (Listing 3.24). Ein derart zugewiesenes Relationsobjekt muss bereits persistent sein oder separat persistent gemacht werden. Später werden Kaskadierungsoptionen vorgestellt, die diese Einschränkung aufheben.

Bei der Arbeit mit Relationsattributen ist es wichtig, in Objekten zu denken – die Fremdschlüsselbeziehung ist auf der Java-Ebene vollständig verdeckt. Man weist dem Relationsattribut ein komplettes Objekt zu, nicht etwa nur seine ID.

```
Flight flight = entityManager.find(Flight.class, id);
System.out.println("AirCraft: " + flight.getAirCraft());

AirCraft otherPlane = entityManager.find(AirCraft.class, somePlaneId);
flight.setAirCraft(otherPlane); // Wird am TX-Ende gespeichert
```

Listing 3.24: Nutzung der unidirektionalen n:1-Beziehung zur Laufzeit

3.3.2 Unidirektionale 1:n-Relationen

Andersherum geht es auch. Die Vorgehensweise ist ähnlich, nur enthält das Relationsattribut nun mehrere Objekte der Gegenseite. Es muss dementsprechend eine *Collection* sein – nutzbar sind die Typen *java.util.Collection*, *java.util.List* und *java.util.Set*[6]. Die Relations-Annotation heißt nun umgekehrt wie im vorigen Fall: *@OneToMany* (Listing 3.25).

```
@Entity
@Access(AccessType.FIELD)
public class Person
{
  @Id @GeneratedValue
  private Integer          id;
  private String           name;

  @OneToMany
  private List<MailAddress> mailAddresses;
  …
}

@Entity
@Access(AccessType.FIELD)
public class MailAddress
{
  @Id @GeneratedValue
  private Integer id;
  …
}
```

Listing 3.25: Unidirektionale 1:n-Beziehung

Für die Deklaration des Relationsattributs sollten unbedingt die generischen Typen genutzt werden, da dann der Typ der Gegenseite daraus geschlossen werden kann. Bei Nutzung eines Raw Types müsste der referenzierte Typ als Parameter *targetEntity* der Annotation *@OneToMany* übergeben werden.

Bemerkenswerterweise wird im Standard-Mapping einer solchen unidirektionalen 1:n-Relation eine Verknüpfungstabelle verwendet. Soll stattdessen ein Fremdschlüssel auf der Many-Seite eingesetzt werden, muss dies explizit mit der Annotation *@JoinColumn* angefordert werden (Abb. 3.11).

6 *java.util.Map* ist auch erlaubt, aber für Relationsattribute kaum sinnvoll

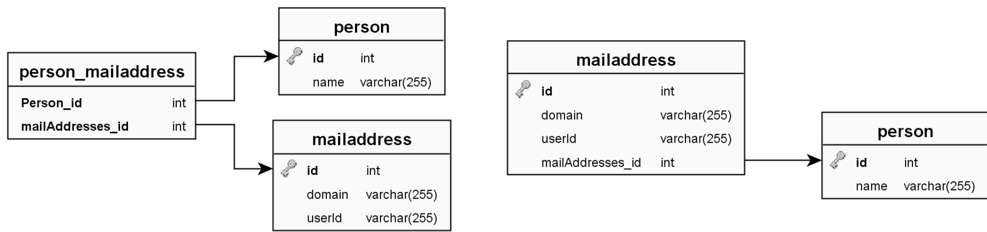


Abbildung 3.11: Tabellenlayout zu Listing 3.25 links ohne und rechts mit „@JoinColumn"

Gibt man *@JoinColumn* keinen Spaltennamen als Parameter mit, wird der Fremdschlüssel in der Tabelle wiederum aus dem Namens des Relationsattributs, einem '_' und dem Namen des referenzierten Primärschlüsselattributs zusammengesetzt. Da das Attribut im Java-Code auf der anderen Seite der Relation definiert ist, ergibt sich i. d. R. ein missverständlicher Spaltenname. So ist im gezeigten Beispiel *mailAddresses_id* ein eher unglücklich gewählter Name. Besser wäre hier eine explizite Angabe des Fremdschlüsselnamens, z. B. mit *@JoinColumn(name = "person_id")*.

Die Arbeit mit den so verknüpften Objekten zur Laufzeit des Programms ist genauso unspektakulär wie im entgegengesetzt gerichteten Fall: Beim Lesen eines Objekts der Owner-Seite (im Beispiel *Person*) ist die *Collection* der referenzierten Objekte (im Beispiel *MailAddress*) ohne Weiteres nutzbar. Ob diese direkt mitgelesen wurde oder ob dazu zusätzlich DB-Zugriffe genutzt werden, hängt davon ab, ob Eager oder Lazy Loading verwendet wird. Dazu folgt später ein separater Abschnitt. Eine Änderung der *Collection* – Modifikation ihres Inhalts oder Zuweisung einer kompletten neuen *Collection* – führt analog zu oben am Transaktionsende zur passenden Änderung der Verweise in der Datenbank.

Bei der Wahl des *Collection*-Typs hat man die Qual der Wahl: Sollte man *List* verwenden oder lieber *Set*? Nun, das kommt drauf an … *Set* entspricht eher der Semantik einer DB-Tabelle, in der ebenso nicht zwei gleiche Objekte liegen können. *List*-Elemente können dagegen angeordnet werden, was sicher auch nützlich sein kann.

Statt eines der einfachen Collection-Typs kann für mengenwertige Relationsattribute auch *java.util.Map* genutzt werden. Anders als bei Element Collections finden sich dafür aber im Bereich der Relationen kaum sinnvolle Anwendungen.

3.3.3 Bidirektionale 1:n-Relationen

Nach der Betrachtung der beiden unidirektionalen Varianten stellt eine bidirektionale Beziehung nun keine große Hürde mehr dar. Hier sind auf beiden Seiten Relationsattribute aufzunehmen, und zwar auf der einen Seite eine *Collection*, auf der anderen Seite ein einzelnes Attribut vom Typ der Gegenseite. Diese Attribute stellen allerdings nicht zwei unabhängige Relationen dar, sondern sind nur zwei Sichten auf eine Beziehung. Um das

auszudrücken, muss der Annotation *@OneToMany* als Parameter *mappedBy* der Name des Relationsattributs der Gegenseite mitgegeben werden (Listing 3.26).

```
@Entity
@Access(AccessType.FIELD)
public class Publisher
{
  @Id @GeneratedValue
  private Integer    id;
  private String     name;

  @OneToMany(mappedBy = "publisher")
  private List<Book> books;
  …
}

@Entity
@Access(AccessType.FIELD)
public class Book
{
  @Id @GeneratedValue
  private Integer   id;
  private String    name;
  private String    isbn;

  @ManyToOne
  private Publisher publisher;
  …
}
```

Listing 3.26: Bidirektionale 1:n-Beziehung

Im Beispiel definiert also das Attribut *publisher* der Klasse *Book* die eine Sicht der Relation, auf die sich die umgekehrte Sicht – das Attribut *books* in *Publisher* – mithilfe des *mappedBy*-Parameters bezieht.

Die Platzierung des *mappedBy*-Parameters bestimmt zudem, welche Seite der Owner der Relation ist, nämlich die Seite ohne diesen Parameter. In der aktuellen JPA-Version muss dies für 1:n-Relationen stets die Many-Seite sein. Diese Beschränkung wird vermutlich in einer der nächsten Versionen des Standards aufgehoben.

Das Mapping zur Datenbank entspricht dem von unidirektionalen n:1-Relationen, wobei mithilfe der Annotationen *@JoinColumn* bzw. *@JoinTable* auf der Owner-Seite die Namen von Fremdschlüsseln und der ggf. vorhandenen Verknüpfungstabelle wie zuvor gezeigt bestimmt werden können.

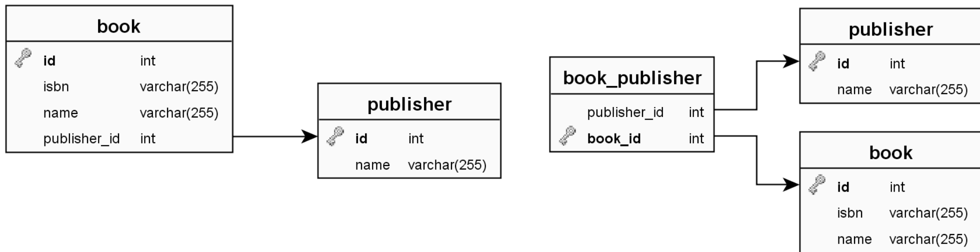


Abbildung 3.12: Tabellen zu Listing 3.26 links ohne und rechts mit „@JoinTable"

Bei der Benutzung der bidirektional verbundenen Entities zur Programmlaufzeit können nun wiederum die Relationsattribute zur Navigation durch die Beziehung genutzt werden: Ein per *find* gelesenes Objekt vom Typ *Publisher* enthält in seinem Attribut *books* alle zugeordneten *Book*s, und umgekehrt enthält das Attribut *publisher* eines eingelesenen *Book*-Objekts den zugehörigen *Publisher*, wobei das Füllen dieser Attribute durchaus bis zur ersten Verwendung verzögert sein kann.

Beim Verknüpfen von Objekten zur Laufzeit ist etwas Vorsicht geboten: Nur das Relationsattribut auf der Owner-Seite ist schlussendlich für den DB-Eintrag maßgeblich. Das Attribut der Gegenseite – die umgekehrte Sicht der bidirektionalen Relation – wird damit aber nicht automatisch mitgepflegt.

```
Publisher publisher = entityManager.find(Publisher.class, 1);
Book book = entityManager.find(Book.class, 4711);
book.setPublisher(publisher);   // (a) Publisher in Book setzen
publisher.getBooks().add(book); // (b) Book zu Publisher hinzufügen
```

Listing 3.27: Objektzuordnung in einer bidirektionalen 1:n-Relation

In Listing 3.27 wird einem *Publisher* ein neues *Book* zugeordnet. Es wird hier angenommen, dass der gezeigte Codeausschnitt innerhalb einer Transaktion ausgeführt wird. Dann wird die neue Zuordnung beim Commit der Transaktion in die Datenbank geschrieben, d. h. der entsprechende Fremdschlüsselwert wird eingetragen. Dafür ist aber nur das mit (a) markierte Statement zuständig, die darauf folgende Anweisung (b) hat dagegen keinerlei Konsequenz im Hinblick auf den DB-Inhalt. Sie führt nur zu einer konsistenten Änderung des *Publisher*-Objekts im Hauptspeicher.

Es stellt natürlich eine gewisse Gefahr dar, dass die Bedienung beider Relationsattribute vom Benutzer der Klassen konsistent vorgenommen werden muss. Eine mögliche Alternative ist die Verkapselung der Zugriffe in der entsprechenden Methode bspw. der Owner-Seite. So könnte man im Beispiel der *Publisher-Book*-Relation den schreibenden Zugriff auf *Publisher.books* unterdrücken, indem man die entsprechenden Methoden nicht *public* macht. Stattdessen würde dann die Methode *Book.setPublisher* die Relationsattribu-

te der Gegenseite mit pflegen. Um den Quereinstieg über den Getter zu verhindern, sollte dieser eine unveränderliche *Collection* liefern (Listing 3.28).

```
@Entity
@Access(AccessType.FIELD)
public class Publisher
{
  …
  public List<Book> getBooks()
  {
    return Collections.unmodifiableList(this.books);
  }

  void addBook(Book book)
  {
    this.books.add(book);
  }

  void removeBook(Book book)
  {
    this.books.remove(book);
  }
  …
}

@Entity
@Access(AccessType.FIELD)
public class Book
{
  …
  public void setPublisher(Publisher publisher)
  {
    if (this.publisher != null)
    {
      this.publisher.removeBook(this);
    }

    this.publisher = publisher;

    if (this.publisher != null)
    {
      this.publisher.addBook(this);
    }
  }
  …
}
```

Listing 3.28: Pflege des Attributs der Gegenseite im Setter einer Entity

Kritische Gemüter werden hier einwenden, dass einerseits die konsistente Pflege des Relationsattributs auf der Nicht-Owner-Seite nun immer durchgeführt wird und damit auch

den entsprechenden Laufzeitaufwand auslöst, selbst wenn die Daten anschließend gar nicht mehr benötigt werden, und dass andererseits *Book.setPublisher* nun deutlich mehr tut, als man von einem Setter gemeinhin erwartet. Und ja, sie haben damit durchaus recht. Sie müssen für sich entscheiden, welche Variante für Sie das geringere Übel darstellt.

3.3.4 Uni- und bidirektionale 1:1-Relationen

Mit Blick auf die Ausführungen zu 1:n-Relationen sind die 1:1-Beziehungen leicht erklärt: Hier hat man auf der Owner-Seite ein Relationsattribut vom Typ der Gegenseite. Bei bidirektionalen Relationen ist dort ein Attribut der Owner-Seite zu finden. Beide sind mit *@OneToOne* annotiert, wobei auf der Nicht-Owner-Seite mithilfe des Parameters *mappedBy* wiederum Bezug auf die andere Seite genommen wird. Listing 3.29 zeigt das Beispiel der 1:1-Beziehung zwischen einer Firma und dem zugehörigen Handelsregistereintrag. Das Mapping auf DB-Tabellen entspricht dem der n:1-Beziehungen, kann aber genau wie dort mittels *@JoinColumn* bzw. *@JoinTable* beeinflusst werden.

```
@Entity
@Access(AccessType.FIELD)
public class Company
{
  @Id @GeneratedValue
  private Integer     id;
  private String      name;

  @OneToOne
  private CommRegEntry commRegEntry;
  …
}

@Entity
@Access(AccessType.FIELD)
public class CommRegEntry
{
  @Id
  public Integer id;

  private String legalForm;
  …
}
```

Listing 3.29: Beispiel für eine unidirektionale 1:1-Relation

Die Relation wird in der Datenbank normalerweise mithilfe eines Foreign Keys abgebildet. So enthält im Beispiel die Tabelle *company* die Spalte *commRegEntry_id* als Referenz auf die Tabelle *commRegEntry* (Abb. 3.13 links). Diesen Fremdschlüssel kann man einsparen, wenn man die beiden verbundenen Tabellen mit gemeinsamen Primärschlüsselwer-

ten versorgt (Abb. 3.13 rechts). Dazu muss zusätzlich zu *@OneToOne* auf der Owner-Seite die Annotation *@PrimaryKeyJoinColumn* verwendet werden (Listing 3.30).

```
@OneToOne
@PrimaryKeyJoinColumn
private CommRegEntry commRegEntry;
```

Listing 3.30: Verknüpfung über den gemeinsamen Primärschlüssel

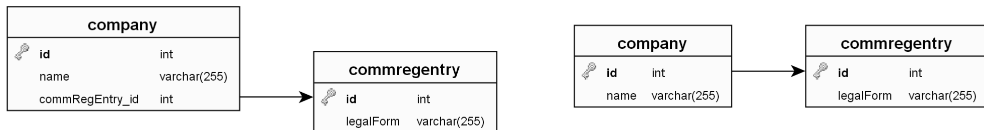


Abbildung 3.13: Tabellen zu Listing 3.29 (links) und Listing 3.30 (rechts)

Die Nutzung eines gemeinsamen Primärschlüssels scheint elegant zu sein, erzeugt aber ein Mismatch mit der Deklaration der Java-Klassen. Aus deren Deklaration könnte man entnehmen, dass die Zuordnung zweier Objekte innerhalb der 1:1-Relation durch Zuweisung des einen Objekts an das Relationsattribut des Owner-Objekts geschieht. So ist das ja im Allgemeinen bei JPA-Relationen – nur nicht bei 1:1-Relationen mit gemeinsamem Primärschlüssel. Hier müssen die IDs der zu verbindenden Objekte explizit auf den gleichen Wert gesetzt werden. Das Relationsattribut wird zwar beim Lesen korrekt besetzt, sein Wert spielt aber beim Schreiben in die DB keine Rolle. Zudem müssen die Einfügeoperationen in der richtigen Reihenfolge geschehen, um die Foreign Key Constraints der DB nicht zu verletzen. Um also z. B. eine neue *Company* mit dem zugeordneten *CommRegEntry* in die DB einzutragen, muss man wie in Listing 3.31 gezeigt vorgehen.

```
Company company = new Company();
company.setId(5812);
company.setName("Gierschlund & Raffke OHG");

CommRegEntry commRegEntry = new CommRegEntry();
commRegEntry.setId(company.getId());   // Gleiche ID erzeugt Zuordnung
commRegEntry.setLegalForm("OHG");

company.setCommRegEntry(commRegEntry); // Unwichtig für DB-Eintrag

entityManager.persist(commRegEntry);   // Reihenfolge!
entityManager.persist(company);
```

Listing 3.31: Objektzuordnung bei gemeinsamem Primärschlüssel

Damit finden wir bei 1:1-Relationen die Situation, dass eine Änderung des Mappings durch Hinzufügen von *@PrimaryKeyJoinColumn* eine andere Behandlung der Objekte zur Laufzeit nach sich zieht. Die Kopplung der Java-Klassen an das Tabellenlayout ist hier

also offensichtlich stärker als bei anderen Mapping-Optionen. Viele machen aus diesem Grund einen Bogen um *@PrimaryKeyJoinColumn*, wenn es möglich ist.

3.3.5 Uni- und bidirektionale n:m-Relationen

N:m-Relationen werden analog zu denen der bislang dargestellten Kardinalitäten definiert. Die Relationsfelder sind hier allerdings auf beiden Seiten – so vorhanden – *Collection*s, und als Relationsannotation wird *@ManyToMany* verwendet. In Listing 3.32 ist eine bidirektionale n:m-Relation zwischen Anwendungen und Benutzern – z. B. als Basis eines Rechtesystems – dargestellt.

```
@Entity
@Access(AccessType.FIELD)
public class Application
{
  @Id @GeneratedValue
  private Integer   id;
  private String    name;

  @ManyToMany(mappedBy = "usableApplications")
  private List<User> authorizedUsers;

  // ...
}

@Entity
@Access(AccessType.FIELD)
@Table(name = "USERS") // USER ist für viele DB ein reserviertes Wort
public class User
{
  @Id
  private String          id;
  private String          name;

  @ManyToMany
  private List<Application> usableApplications;

  // ...
}
```

Listing 3.32: Beispiel für eine bidirektionale n:m-Relation

In der Datenbank lässt sich eine n:m-Relation naturgemäß nur mithilfe einer Verknüpfungstabelle realisieren (Abb. 3.14). Auf deren Gestaltung kann man wiederum mithilfe von *@JoinTable* auf der Owner-Seite Einfluss nehmen.

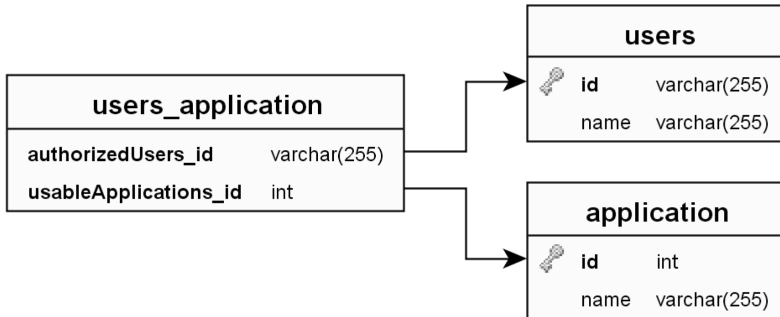


Abbildung 3.14: Tabellen zu Listing 3.32

3.3.6 Eager und Lazy Loading

Nach dem Lesen einen Objekts aus der Datenbank ist gewährleistet, dass wir dessen Attribute inklusive der Relationsattribute direkt benutzen können. Das umfasst auch die per Relation zugeordneten Objekte. Das heißt aber nicht, dass der gesamte Datenumfang bereits beim ersten Zugriff zur Datenbank geladen wird. Die referenzierten Objekte können vielmehr verzögert nachgeladen werden, wenn sie benutzt werden.

Zur Steuerung dieses Ladeverhaltens akzeptieren die Relationsannotationen *@OneToOne*, *@OneToMany*, *@ManyToOne* und *@ManyToMany* den Parameter *fetch*, dem ein Wert der Aufzählung *FetchType* übergeben wird:

- *FetchType.EAGER*

Die referenzierten Objekte werden direkt gelesen, wenn das führende Objekt eingelesen wird (Eager Loading).

- *FetchType.LAZY*

Die referenzierten Objekte werden zunächst nicht gelesen, sondern nachgeladen, wenn sie zum ersten Mal verwendet werden (Lazy Loading).

Wird kein *FetchType* angegeben, gilt für die *Collection*-basierten Attribute *LAZY* als Voreinstellung, während alle anderen Attribute *EAGER* geladen werden.

Die Angabe von *LAZY* ist nur ein Hinweis an den Persistenzprovider, dass Lazy Loading genutzt werden darf. Die Spezifikation verlangt nicht, dass dieses Verfahren für alle Attributtypen verfügbar ist, allerdings unterstützen in einer Serverumgebung alle mir bekannten Provider (Eclipselink, Hibernate, OpenJPA) Lazy Loading durchgängig. Beim Einsatz in einer SE-Umgebung muss ggf. ein Bytecode Enhancement durchgeführt werden, d. h. eine Manipulation des Bytecodes der betroffenen Klassen zur Build-Zeit oder beim Laden der Klassen. Details dazu finden Sie in der Dokumentation Ihres Providers.

Im weiter oben besprochenen Beispiel der 1:n-Relation zwischen *Publisher* und *Book* gilt als Vorgabe Lazy Loading für das Attribut *Publisher.books*. Diese *Collection* wird also beim Lesen eines *Publisher*-Objekts noch nicht mit Werten gefüllt. Das geschieht mit einem erneuten Zugriff auf die Datenbank, wenn die Daten benutzt werden. Die Spezifikation beschreibt nicht exakt, wann der Nachladevorgang ausgelöst wird. In aller Regel reicht der Aufruf einer der *Collection*-Methoden – und sei es nur *size()*.

Wäre das Relationsattribut dagegen mit *EAGER* parametrisiert worden (Listing 3.33), würden mit jedem gelesenen *Publisher* direkt die zugeordneten Books eingelesen.

```
@OneToMany(mappedBy = "publisher", fetch=FetchType.EAGER)
private List<Book> books;
```

Listing 3.33: Konfiguration von Eager Loading bei einer 1:n-Relation

Achtung: Mit Eager Loading ist die Gefahr verbunden, dass man unbewusst (viel) mehr Daten einliest, als man für den aktuellen Geschäftsprozess benötigt. Denken Sie daran, dass die Menge der referenzierten Objekte groß sein kann und dass Eager Loading transitiv funktioniert. Sind also in den direkt geladenen Objekten wiederum *EAGER*-Attribute, werden auch diese geladen usw. Bei ungeschickter Konfiguration der Relationen einer Anwendung hat man so vielleicht mit einem *find* direkt die ganze Datenbank im Speicher …

Gehen Sie also vorsichtig mit Eager Loading um. Die Voreinstellung, dass nur Einzelobjekte *EAGER* geladen werden, und zunächst nur für *Collection*-basierte Werte Lazy Loading gilt, ist in vielen Fällen sehr gut. Nur wenn Sie wissen, dass Ihre Anwendung in allen(!) Fällen die referenzierten Objekte benötigt, und dass weiterhin die Menge dieser Objekte handhabbar ist – nur dann sollten Sie für *Collection*s Eager Loading einsetzen.

Ein Problem kann Lazy Loading im Zusammenhang mit Detached Objects erzeugen: Das Nachladen von LAZY-Attributen ist nur für persistente Objekte möglich. Verlassen diese den Bereich des Entity Managers, können bis dahin nicht geladene Werte nicht mehr gefüllt werden. Greift die Anwendung dann darauf zu, wird eine (providerspezifische) Lazy Load Exception ausgeworfen.

Wollen Sie also mit Detached Objects arbeiten, müssen Sie dafür sorgen, dass alle von der Anwendung genutzten Attribute geladen wurden, bevor die Objekte vom Entity Manager abgekoppelt werden – entweder durch Eager Loading oder durch einen expliziten Zugriff. Bei der Ausführung einer Query besteht die Möglichkeit, durch einen sog. Fetch Join Relationsattribute direkt zu laden, auch wenn sie mit *LAZY* parametrisiert wurden. Queries und Fetch Joins werden später im Detail erläutert.

Lazy Loading kann auch für andere als Relationsattribute angegeben werden: *@ElementCollection* akzeptiert den Parameter *fetch* ebenso wie *@OneToMany*. Auch hier ist der voreingestellte Wert *LAZY*.

Für einfache Attribute steht die Annotation *@Basic* zur Verfügung. Auch hier kann *fetch* verwendet werden, allerdings mit dem Vorgabewert *EAGER*. Sinnvoll ist die Verwendung von Lazy Loading für solche Attribute aber allenfalls für sehr große Objekte wie z. B. LOBs (Listing 3.34).

```
@Lob
@Basic(fetch = FetchType.LAZY)
private byte[] picture;
```

Listing 3.34: Lazy Loading für ein großes Einzelattribut

3.3.7 Kaskadieren

Bislang war eine Voraussetzung für die Arbeit mit den durch eine Relation referenzierten Objekten, dass sie jeweils unabhängig vom referenzierenden Objekt mit dem Entity Manager bearbeitet werden müssen. Um also z. B. einen neuen *Publisher* mit seinen *Book*s in die Datenbank einzufügen, müssen zunächst die *Book*s an *EntityManager.persist* übergeben werden, bevor auch das neue *Publisher*-Objekt persistiert werden kann.

Mithilfe des Parameters *cascade*, den alle Relationsannotationen annehmen, kann dies bequemer gestaltet werden. Enthält der Parameter z. B. den Wert *CascadeType.PERSIST*, so wird ein auf das Entity-Objekt angewandtes *persist* automatisch auch auf die davon abhängigen Objekte angewendet. Sind darin wiederum Relationen mit *CascadeType.PERSIST* vorhanden, setzt sich das Verfahren entsprechend fort. *CascadeType* ist ein Aufzählungstyp, der die Konstanten *PERSIST*, *MERGE*, *REMOVE*, *REFRESH* und *DETACH* enthält. Sie beziehen sich in der beschriebenen Weise auf die Methoden *persist*, *merge* etc. des Entity Managers. Der Parameter *cascade* der Annotationen akzeptiert ein Array von *CascadeType*-Werten, sodass hier eine beliebige Kombination der Entity-Manager-Grundoperationen kaskadierend gestaltet werden kann. Sollen alle genannten Methoden einbezogen werden, kann dafür der Wert *CascadeType.ALL* verwendet werden,der alle restlichen umfasst. Listing 3.35 zeigt als Beispiel eine klassische Master-Detail-Kombination: Einem *Order*-Objekt sind mehrere *OrderLines* zugeordnet. Abbildung 3.15 zeigt die zugehörige Tabellenstruktur. Logisch betrachten wir die Kombination als einen Auftrag, der zusammen mit seinen Auftragspositionen gespeichert werden soll. Daher ist in *@OneToMany* für *cascade* u. a. *PERSIST* eingetragen. Umgekehrt sollen beim Löschen eines Auftrags auch die zugeordneten *OrderLine*-Objekte aus der DB entfernt werden, daher ist auch *REMOVE* eingesetzt.

```
@Entity
// Achtung: ORDER ist ein reserviertes Wort in SQL
@Table(name = "ORDERS")
@Access(AccessType.FIELD)
public class Order
{
```

```
  @Id @GeneratedValue
  private Integer       id;

  @OneToMany(cascade = { CascadeType.PERSIST, CascadeType.REMOVE })
  @JoinColumn(name = "ORDER_ID")
  private List<OrderLine> orderLines;
  …
}

@Entity
@Access(AccessType.FIELD)
public class OrderLine
{
  @Id @GeneratedValue
  private Integer id;
  …
}
```

Listing 3.35: Kaskadieren von „persist“ und „remove“

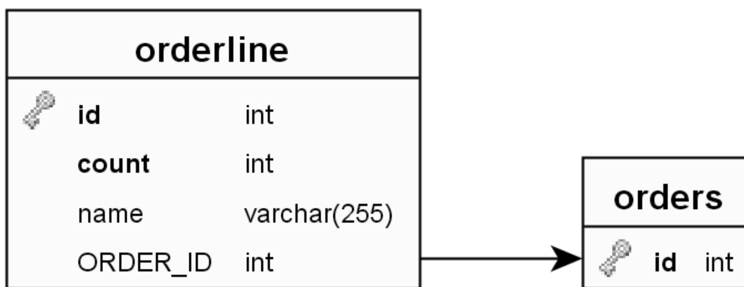


Abbildung 3.15: Tabellen zu Listing 3.35

Im Beispiel sind die restlichen *CascadeType*s weggelassen worden. Die wären hier aber auch sinnvoll, denn die *OrderLine*-Objekte sind komplett abhängige Objekte. Sie sollten alle Operationen des jeweiligen Order-Objekts mitmachen, also auch *merge*, *refresh* und *detach*. In einem solchen Fall kann natürlich statt einer Aufzählung aller Werte *CascadeType.ALL* verwendet werden.

Kaskadierende Operationen sind recht bequem und erlauben eine elegante Programmierung von Zugriffen auf komplexere Objekte. Trotzdem sollten Sie beim Einsatz von *cascade* Vorsicht walten lassen. *REMOVE* entspricht einem Cascading Delete in der DB, was i. A. nur für abhängige Objekte sinnvoll ist, d. h. für Objekte, die ohne das sie referenzierende Objekte irrelevant sind. Die anderen Operationen sind zwar etwas harmloser, machen aber bei ganz genauer Betrachtung auch nur bei abhängigen Objekten Sinn.

3.3.8 Orphan Removal

Von Orphans, verwaisten Einträgen, sprechen wir, wenn sie zwar noch vorhanden sind, aber nicht mehr genutzt werden können. Diese Situation entsteht, wenn die Referenz auf einen abhängigen Eintrag entfernt wird, der Eintrag selbst aber bestehen bleibt. Genau das würde passieren, wenn im Beispiel oben ein Auftrag, bestehend aus einem *Order*-Objekt und einigen *OrderLine*-Objekten, so verändert wird, dass die Anzahl von *OrderLine*s verkleinert wird. Beim Abspeichern des Auftrags würden die betroffenen Referenzen in der DB entfernt, d. h. die Fremdschlüssel auf *null* gesetzt, die Einträge in der Tabelle aber erhalten bleiben.

Angenommen, wir hätten zunächst einen Auftrag mit vier Positionen. Nach Entfernen der letzten Position ergibt sich damit die in Abbildung 3.16 dargestellte Situation. Der betroffene Eintrag existiert weiterhin in der Datenbank, allerdings nun ohne Verbindung zu einem *Order*-Eintrag. Fachlich ist dieser Eintrag nun unsichtbar.

id	name	count	ORDER_ID
14	Apfel	10	13
15	Pflaume	50	13
16	Limette	5	13
17	Birne	30	13

⇨

id	name	count	ORDER_ID
14	Apfel	10	13
15	Pflaume	50	13
16	Limette	5	13
17	Birne	30	(null)

Abbildung 3.16: „OrderLine“-Tabelle vor und nach dem Entfernen einer Position aus der „Order 13“ ohne Orphan Removal …

Man könnte zur Bereinigung dieses Zustands z. B. regelmäßig alle Einträge aus der Tabelle löschen, deren *ORDER_ID null* ist. Das geht aber im Fall von 1:1- oder 1:n-Relationen eleganter mithilfe des Parameters *orphanRemoval*, den die entsprechenden Annotationen *@OneToOne* bzw. *@OneToMany* akzeptieren. Hat er den Wert *true*, werden verwaiste Einträge automatisch gelöscht (Abb. 3.17). Die Voreinstellung für den Parameter ist *false*.

id	name	count	ORDER_ID
14	Apfel	10	13
15	Pflaume	50	13
16	Limette	5	13
17	Birne	30	13

⇨

id	name	count	ORDER_ID
14	Apfel	10	13
15	Pflaume	50	13
16	Limette	5	13

Abbildung 3.17: … und mit Orphan Removal

Orphan Removal ist also auch eine Art kaskadierende Löschoperation, diesmal aber ausgelöst durch die Änderung eines Relationsattributs. Man findet *orphanRemoval=true* daher häufig, wenn *cascade* u. a. *REMOVE* enthält.

3.3.9 Anordnung von Relationselementen

Die Elemente von *Collection*-basierten Relationsattributen werden beim Einlesen in einer nicht vorbestimmten Reihenfolge in die *Collection* eingetragen – üblicherweise durch Eigenschaften der Datenbank bestimmt. Man kann die Elemente allerdings in eine fachliche Reihenfolge bringen lassen, indem man das Relationsattribut mit der Annotation *@OrderBy* versieht. Als Parameter gibt man dabei den Namen des Attributs der *Collection*-Elemente an, nach dem sortiert werden soll. Mit dem Zusatz *ASC* oder *DESC* kann eine aufsteigende oder absteigende Sortierung erreicht werden; *ASC* ist voreingestellt. Mehrere Sortierangaben können durch Komma voneinander getrennt gemacht werden. Das Beispiel in Listing 3.36 zeigt, wie die *OrderLine*s eines *Order*-Objekts anhand ihres Attributs *name* absteigend sortiert werden können (was hier vermutlich nicht sonderlich sinnvoll ist).

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@JoinColumn(name = "ORDER_ID")
@OrderBy("name DESC")
private List<OrderLine> orderLines;
```

Listing 3.36: Anordnung von Relationselementen mit „@OrderBy“

Die geforderte Anordnung der Elemente wird durch einen entsprechenden Zusatz im zum Lesen verwendeten Datenbankbefehl erzeugt. Demzufolge ist danach natürlich nur die Anwendung weiter für die Einhaltung der Reihenfolge verantwortlich. Soll also im Beispiel ein neues *OrderLine*-Objekt in die *Collection* eingefügt werden, ist es die Aufgabe der Anwendung, dies an der in Bezug auf die Sortierung richtigen Stelle zu tun.

Wird *@OrderBy* ohne Parameter verwendet, ergibt sich die Anordnung durch die Sortierung der ID-Werte.

@OrderBy kann auch in Kombination mit *@ElementCollection* verwendet werden. Wird hier keine Sortierangabe gemacht, werden die Elemente entsprechend ihrer natürlichen Ordnung angeordnet. Das ist aber nur bei *Collection*s über einfachen Typen erlaubt. Für *Embeddable*s ist die Sortierangabe obligatorisch.

Eine Alternative zu *@OrderBy* ist die sog. Persistente Ordnung, bei der die Anordnung der *Collection*-Elemente selbst persistent gemacht wird, d. h. die Reihenfolge wird in der Datenbank mit abgespeichert und beim späteren Lesen wieder genauso hergestellt. Dazu muss das Relationsattribut mit der Annotation *@OrderColumn* versehen werden. Das führt zur Verwendung einer zusätzlichen Spalte in der Tabelle der referenzierten Entity bzw. in der Verknüpfungstabelle. Beim Speichern wird diese Spalte zum Durchnummerieren der Einträge verwendet (aufsteigend, ohne Lücken, beginnend mit *0*). Beim späteren Lesen wird anhand dieser Zahlen die alte Ordnung wieder hergestellt. Der Name der Nummerierungsspalte ergibt sich aus dem Namen des Relationsattributs, ergänzt um *'_ORDER'*. Er kann aber auch explizit als Parameter *name* der Annotation *@OrderColumn* mitgegeben werden (Listing 3.37, Abb. 3.18, Abb. 3.19).

```
@OneToMany(cascade = CascadeType.ALL, orphanRemoval = true)
@JoinColumn(name = "ORDER_ID")
@OrderColumn(name = "ORDERLINES_ORDER")
private List<OrderLine> orderLines;
```

Listing 3.37: Persistente Anordnung der Relationselemente

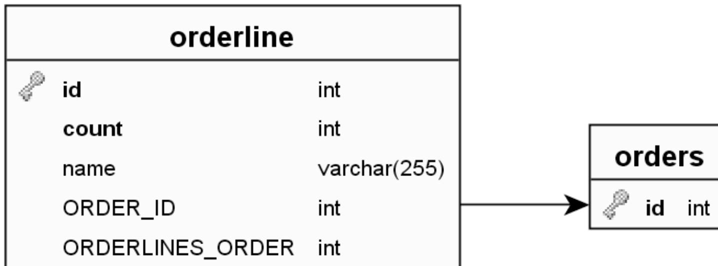


Abbildung 3.18: Tabellenlayout zu Listing 3.37

id	count	name	ORDER_ID	ORDERLINES_ORDER
14	10	Apfel	13	0
15	50	Pflaume	13	1
16	5	Limette	13	2
17	30	Birne	13	3

Abbildung 3.19: Nummerierungsspalte in der „OrderLine"-Tabelle

Auch dieses Anordnungsverfahren kann für Element-Collections verwendet werden.

3.4 Queries

Neben den bisher betrachteten Einzeleintragsoperationen spielen bei der Arbeit mit Datenbankinhalten Suchanfragen eine große Rolle. Java Persistence bietet dafür gleich drei Möglichkeiten an: Eine objektorientierte Abfragesprache, direktes SQL und eine Programmschnittstelle zum Aufbau von Queries.

3.4.1 JPQL

Die Java Persistence Query Language ist eine SQL-ähnliche Abfragesprache, die allerdings nicht mit Tabellen und Spalten arbeitet, sondern mit Klassen und Objekten.

Queries werden zur Laufzeit durch Objekte des Typs *TypedQuery<T>* repräsentiert, wobei der Typparameter angibt, welchen Ergebnistyp man bei Ausführung der Query erwarten

kann. Zur Erzeugung eines Query-Objekts stellt *EntityManager* die Methode *createQuery* zur Verfügung. Die Query-Methode *getResultList* führt die Abfrage durch und liefert das (evtl. leere) Ergebnis in einer passend getypten Liste (Listing 3.38).

```
EntityManager em = …
TypedQuery<Country> query
  = em.createQuery("select c from Country c where c.name like 'D%'",
                   Country.class);
List<Country> countries = query.getResultList();
```

Listing 3.38: Ausführen einer Query mit Ergebnisliste

Die grundsätzliche Form des Abfragetexts wird Ihnen vermutlich von SQL her bekannt sein: *select*, *from* etc. sind übernommen worden und haben die gleiche Bedeutung wie in SQL. Auch die Operatoren wie *like* sind über weite Strecken deckungsgleich. Unterschiede bestehen u. a. darin, dass im Beispiel komplette Objekte selektiert werden anstelle der in SQL üblichen Ergebnis-Tupel. Zudem sind *Country* und *name* Java-Bezeichner und nicht SQL-Namen.

Wird von einer Query nur ein einzelnes Objekt als Ergebnis erwartet, kann man statt *getResultList* die Methode *getSingleResult* verwenden(Listing 3.39). Sie liefert ein passend getyptes Objekt als Rückgabewert und wirft eine Exception aus, wenn nicht genau ein Ergebnis gefunden wird:

- *NoResultException*, falls kein Eintrag gefunden wurde
- *NonUniqueResultException*, falls mehr als ein Eintrag gefunden wurde

```
EntityManager em = …
TypedQuery<Country> query
  = em.createQuery("select c from Country c where c.carCode='D'",
                   Country.class);
Country country = query.getSingleResult();
```

Listing 3.39: Ausführen einer Query mit Einzelergebnis

Die Methode *EntityManager.createQuery* steht auch in einer älteren Variante ohne den Typparameter zur Verfügung. Sie liefert dann statt eines *TypedQuery<T>*-Objekts eines vom Typ *Query*. Es ist grundsätzlich gleichwertig, liefert aber bspw. bei der Query-Ausführung nur eine ungetypte *List* bzw. ein *Object* als Ergebnis. Wo immer es möglich ist, sollten Sie daher die neue, getypte Methodenvariante bevorzugen.

Der grundsätzliche Aufbau eines JPQL-Ausdrucks ist der folgende, wobei die in Klammern gesetzten Teile optional sind:

Select-Klausel
From-Klausel
[Where-Klausel]

[Groupby-Klausel]
[Having-Klausel]
[Orderby-Klausel]

Select- und From-Klausel

Mit diesem obligatorischen Anteil eines JPQL-Ausdrucks wird das Query-Ergebnis strukturell beschrieben. In der einfachsten Form werden im Select-Teil ein oder mehrere Werte ausgewählt, deren Ursprung im From-Teil erklärt wird. Die Verbindung beider Teile geschieht über Identifikationsvariablen. Die selektierten Werte sind komplette Objekte oder Teile davon, die, wie in Java üblich, mithilfe des Punktoperators angegeben werden (Listing 3.40).

```
select c from Country c
select c.population from Country c
select c.name, c.population from Country c
```

Listing 3.40: Selektion kompletter Objekte oder Einzelwerte

Die hinter dem Punkt stehenden Attributnamen sind bei Field Access die Namen der Instanzvariablen der Entity, bei Property Access die Namen der Properties, d. h. die Namen der Getter ohne das führende *get* mit kleinem Anfangsbuchstaben. Steckt hinter dem Attribut ein eingebettetes Objekt oder ein 1:1- oder n:1-Relationsattribut, so kann mit einem weiteren Punkt auf dessen Teilattribute zugegriffen werden.

Das Ergebnis einer Abfrage mit nur einem Selektionswert ist passend zu diesem Wert getypt. Die erste Abfrage im Beispiel würde also eine Liste von *Country*-Objekten liefern, die zweite eine Liste von Long-Zahlen.

Werden mehrere Werte selektiert, ist das Ergebnis der Query ein Object-Array bzw. eine Liste darüber. Die letzte Abfrage im Beispiel würde also eine Liste von *Object[2]*-Objekten liefern, die jeweils mit einem *String* und einem *Long* gefüllt sind. Eine Alternative zum *Object[]*-Ergebnis stellen die weiter unten beschriebenen Constructor Expressions dar.

Die Werte der Select-Klausel beziehen sich immer auf Identifikationsvariablen, die in der From-Klausel definiert werden, und zwar wiederum wie in Java üblich durch Angabe des Variablentyps und eines freigewählten, für die Abfrage eindeutigen Namens. Der Variablentyp ist dabei der Entity-Name, der mit der *@Entity*-Annotation bestimmt wird und als Voreinstellung dem einfachen Klassennamen der Entity entspricht. Mehrere Identifikationsvariablen können durch Kommas getrennt deklariert werden, was aber erst später bei der Formulierung von Joins sinnvoll ist.

Ein bisschen „Syntactic Sugar“ ist auch erlaubt: Statt *select c* darf auch *select object(c)* geschrieben werden und *from Country c* darf auch als *from Country as c* formuliert werden.

Wie oben bereits angeführt, kann mithilfe des Punktoperators durch 1:1- und n:1-Relationen „navigiert" werden. Die Join-Bedingung ist bereits durch die Konfiguration der Relation definiert, muss daher in der Query nicht mehr angegeben werden (Listing 3.41).

```
// Bücher zu Verlagen, deren Name mit O beginnt (n:1-Relation)
select b from Book b where b.publisher.name like 'O%'
```

Listing 3.41: Punktoperator zum Zugriff auf Elemente einer 1:1- oder n:1-Relation

Für 1:n- und n:m-Relationen kann der Punktoperator leider nicht in der gleichen Weise angewendet werden. Vielmehr muss zum Zugriff auf mengenwertige Attribute mithilfe des Operators *join* eine zusätzliche Query-Variable deklariert werden. Die Join-Bedingung wird analog zu oben nicht explizit angegeben, sondern aus der Konfiguration der Entities entnommen (Listing 3.42).

```
// Verlage mit Büchern zum Thema JPA (1:n-Relation)
select distinct p from Publisher p join p.books b
  where b.name like '%JPA%'
```

Listing 3.42: Deklaration einer Join-Variablen zum Zugriff auf 1:n- und n:m-Relationen

Im Beispiel wird neben der Haupt-Query-Variablen *p* die zusätzliche Variable *b* als Repräsentant der Elemente aus *p.books* deklariert. *b* kann dann, wie schon gesehen, in den restlichen Klauseln des JPQL-Ausdrucks verwendet werden. Bei der gezeigten Verwendung von *join* handelt es sich um einen sog. Inner Join, bei dem das Kreuzprodukt der beteiligten Objektmengen zur weiteren Verarbeitung benutzt wird. Bei der Selektion können daher Doppelergebnisse auftreten, was man mit *distinct* unterdrücken kann.

Ein Inner Join berücksichtigt nur Daten, die bzgl. der Join-Bedingung eine Zuordnung haben. Mithilfe eines Outer Joins kann die Ergebnismenge auch auf solche Elemente ausgedehnt werden, die keine zugeordneten Elemente in der betroffenen Relation haben. Java Persistence kennt nur Left Outer Joins, wo das Gesagte für die linke Seite des Joins gilt. Der Operator dazu heißt *left join* (Listing 3.43).

```
// Verlags- und Buchnamen, nur von Verlagen, die Bücher haben
select p.name, b.name from Publisher p join p.books b

// ebenso, nun allerdings für alle Verlage (b.name ist dann ggf. null)
select p.name, b.name from Publisher p left join p.books b
```

Listing 3.43: Left Outer Join

Auch hier gibt es wieder ein bisschen Syntactic Sugar: Statt *join* dürfen Sie *inner join* schreiben, statt *left join* auch *left outer join* und die Join-Variablen können analog zu oben mit *as* abgetrennt werden.

Für den Inner Join gibt es eine weitere Notation, die man aus der EJB-QL, der EJB Query Language, einem Vorläufer von JPQL, übernommen hat: *in(a.n) b* entspricht *join a.n b* (Listing 3.44).

```
// Verlags- und Buchnamen, nur von Verlagen, die Bücher haben
select p.name, b.name from Publisher p, in(p.books) b
```

Listing 3.44: Alternative Angabe eines Inner Join (vgl. Listing 3.43 oben)

Where-Klausel

Mit where kann die Menge der selektierten Daten eingeschränkt werden. Die darin angegebene Bedingung kann aus Attributen, Konstanten, Operatoren und Parametern zusammengesetzt werden.

Attribute werden, wie schon in der Select-Klausel, mithilfe des Punktoperators auf eine der Identifikationsvariablen der Query bezogen.

Konstanten sind, wie in SQL üblich, Texte in Hochkommas, Zahlen in üblicher Notation, die Boole'schen Konstanten *TRUE* und *FALSE* sowie Aufzählungswerte, die allerdings mit voll qualifiziertem Klassennamen angegeben werden müssen (Listing 3.45).

```
// Länder mit mehr als 50 Mio Einwohnern
select c from Country c where c.population>50000000

// Länder in Europa
select c from Country c
  where c.continent=de.gedoplan.buch.eedemos.entity.Continent.EUROPE

// Mitarbeiter aus Bielefeld (address ist ein @Embeddable)
select e from Employee e where e.address.town='Bielefeld'
```

Listing 3.45: Einschränkung der Ergebnismenge mit „where"

Sollten in einem Text Hochkommas vorkommen, müssen sie verdoppelt werden.

Operatoren

Als Operatoren stehen größtenteils die von SQL bekannten zur Verfügung:

- *+, -, *, /:* Grundrechenarten für numerische Werte
- *=, >, >=, <, <=, <>:* Vergleiche (Achtung*:* <> für ‚ungleich', nicht *!=*)
- *[not] between … and …:* Bereichsabfrage (Grenzen sind inklusive)
- *[not] like '…':* Pattern-Test
- *[not] in (…):* Elementtest (Menge kann Liste, Collection oder Subquery sein)
- *is [not] null:* Test auf *null* bzw. *not null*

- *is [not] empty:* Test ob ein Collection-Attribut (nicht) leer ist
- *[not] member of …:* Test, ob ein Element Teil einer Collection ist
- *[not] exists (…):* Test, ob eine Subquery mindestens ein bzw. kein Ergebnis liefert
- *not, and, or:* Logische Negation bzw. Verknüpfung von Bedingungen

```
// Länder mit 50 bis 100 Mio Einwohnern
select c from Country c where c.population between 50000000 and 100000000

// Länder, deren Name mit G beginnt
select c from Country c where c.name like 'G%'

// Länder mit Vorwahl 1 oder 86
select c from Country c where c.phonePrefix in ('1','86')

// Länder mit gesetztem Autokennzeichen
select c from Country c where c.carCode is not null

// Mitarbeiter mit eingetragenen Skills
select e from Employee e where e.skills is not empty

// Mitarbeiter mit Kenntnissen in JPA
select e from Employee e where 'JPA' member of e.skills

// Verlage mit min. einem Buch über JPA
select p from Publisher p where exists
  (select b from Book b where b.publisher=p and b.name like '%JPA%')
```

Listing 3.46: Weitere Selektionsbeispiele

Das Muster des *like*-Operators kann die Zeichen '_' und '%' enthalten als Platzhalter für ein bzw. mehrere (auch 0) beliebige Zeichen. Sollen diese Zeichen ohne ihre Sonderbedeutung verwendet werden, muss ihnen ein frei wählbares Escape-Zeichen vorangestellt werden: *select … where c.name like '_xyz' excape '\'* sucht nach Einträgen mit Namen *'_xyz'*.

Die Operatoren *all* und *any* lassen sich nutzen, um einen Wert mit allen von einer Subquery selektierten Werten zu vergleichen. *all* liefert dabei ein positives Ergebnis, wenn der Vergleich für alle Sub-Ergebnisse gilt, *any* benötigt dafür nur einen positiven Vergleich. *some* kann synonym für *any* verwendet werden (Listing 3.47).

```
// Verlage mit nur dünnen Büchern (bis 200 Seiten)
select p from Publisher p
  where 200 >= all ( select b.pages from Book b where b.publisher=p)

// Verlage mit sehr dicken Büchern (mehr als 800 Seiten)
select p from Publisher p
  where 800 < any ( select b.pages from Book b where b.publisher=p)
```

Listing 3.47: Vergleich mit den Ergebnissen einer Subquery

Parameter

JPQL kann nummerierte oder benannte Parameter in der Where-Klausel enthalten. Nummerierte oder Positionsparameter haben die Form *?n*, wobei n eine ganze Zahl beginnend mit *1* ist. Benannte Parameter werden als *:name* notiert, worin *name* ein für die Query eindeutiger Identifier ist. Die Parameter dürfen in der Query mehrfach in beliebiger Reihenfolge auftauchen. Zur Versorgung der Query-Parameter mit konkreten Werten dient die Methode *Query.setParameter* (Listing 3.48).

```
TypedQuery<Country> query = em.createQuery(
  "select c from Country c where c.carCode=?1 or c.phonePrefix=:pp",
  Country.class);
query.setParameter(1, "D");
query.setParameter("pp", "39");
```

Listing 3.48: Query-Parameter

Da die JPQL-Ausdrücke dynamisch sind, d. h. erst zur Laufzeit erstellt werden, könnte man auf die Idee kommen, den Query-Text mittels String-Konkatenation zusammenzubauen, statt Parameter einzusetzen (Listing 3.49). Das ist aber aus zweierlei Gründen eine schlechte Vorgehensweise. Zum einen wird der Programmcode eher unleserlicher, zumal man sich mit den Besonderheiten der Datentypen explizit herumschlagen muss (z. B. Texte in Hochkommas, Hochkommas als Textinhalt verdoppeln etc.). Zum anderen verhindert man damit eine Optimierung in der Datenbank, die bei parametrisierten Ausdrücken möglich ist, nämlich die einmalige Analyse des Query-Befehls und Aufstellung eines entsprechenden Ausführungsplans. Also: Liebe Programmierer, machen Sie das hier nicht nach!

```
String cc = …;
String pp = …;
TypedQuery<Country> query = em.createQuery(
  "select c from Country c where c.carCode='" + cc + "'
                          or c.phonePrefix='" + pp + "'",
  Country.class);
```

Listing 3.49: String-Konkatenation als schlechter Ersatz für Query-Parameter

Aggregationsfunktionen

In der Select-Klausel können die Aggregationsfunktionen *avg*, *count*, *max*, *min*, und *sum* für Durchschnitt, Anzahl, Maximum, Minimum und Summe verwendet werden. Das Selektionsergebnis reduziert sich dann auf einen Eintrag, d. h. die Ergebnismenge wird entsprechend den verwendeten Funktionen verdichtet (Listing 3.50).

```
// Maximale Seitenzahl aller Bücher
select max(b.pages) from Book b
```

```
// Anzahl Bücher eines Verlages
select count(b) from Book b where b.publisher.name='O''Melly Publishing'
```

Listing 3.50: Aggregationsfunktionen

Im Zusammenhang mit der weiter unten eingeführten Gruppierung können auch mehrzeilige Ergebnisse geliefert werden.

Funktionen

JPQL kennt weitere Funktionen, die in Select-, Where- und Having-Klausel verwendet werden können. Die folgenden Funktionen liefern einen Text:

- *concat(text, text …):* Textverkettung
- *substring(text, start [,länge]):* Teiltext
- *trim(text):* Text ohne führende und folgende Leerzeichen
- *lower(text):* Text in Kleinbuchstaben
- *upper(text):* Text in Großbuchstaben

Bei *trim* kann sogar noch angegeben werden, ob nur führende oder folgende Zeichen entfernt werden sollen, und welches Zeichen entfernt werden soll: *trim(*[*leading* | *trailing* | *both*] [*zeichen*] *from text)*.

Als numerische Funktionen stehen zur Verfügung:

- *abs(zahl):* Absolutwert
- *index(entity):* Position eines Entity-Objekts in einer geordneten Liste
- *length(text):* Textlänge
- *locate(text, teiltext [,start]):* Teiltextposition
- *mod(zahl, teiler):* Ganzzahliger Rest
- *size(collection):* Anzahl Elemente
- *sqrt(zahl):* Quadratwurzel

Zur Ermittlung von aktuellem Datum bzw. Zeit:

- *current_date:* Aktuelles Datum
- *current_time:* Aktuelle Uhrzeit
- *current_timestamp:* Aktueller Timestamp

Schließlich ermöglichen Case-Ausdrücke eine Art eingebettete Fallunterscheidung ähnlich dem Java-Operator *?:*:

- *case when b then w1 else w2 end:* Wenn Bedingung *b* gilt, dann *w1*, sonst *w2*

Für weitere Details sei auf die Spezifikation verwiesen (Abschnitt 4.6.17, Scalar Expressions).

Groupby- und Having-Klauseln

Die Selektionsergebnisse können gruppenweise verdichtet werden. Dazu gibt man nach *group by* ein oder mehrere (kommagetrennte) Variablen oder Attribute an. Im Ergebnis gibt es dann für jede vorkommende Kombination der Gruppierungsattribute einen Eintrag. Die dorthin selektierten Werte müssen für jede Gruppe konstant sein oder mithilfe einer Aggregationsfunktion ermittelt werden (Listing 3.51).

```
// Kontinente mit dem Durchschnitt der Landeseinwohnerzahlen
select c.continent, avg(c.population) from Country c group by c.continent
```

Listing 3.51: Gruppierung des Selektionsergebnisses

having ist das where der Gruppierung. Die Bedingungen in der Having-Klausel filtern also die zuvor gebildeten Gruppen (Listing 3.52).

```
// Kontinente mit Einwohnerdurchschnitt, aber nur für mehr als 10 Mio
select c.continent, avg(c.population) from Country c
  group by c.continent having avg(c.population)>100000000
```

Listing 3.52: Filterung der gebildeten Gruppen

Orderby-Klausel

Das Selektionsergebnis kann schließlich noch sortiert werden, indem nach *order by* ein oder mehrere (kommagetrennte) Attribute angegeben werden, nach denen die Ergebnisliste angeordnet werden soll. Die Sortierattribute können mit dem Zusatz *asc* oder *desc* für aufsteigendes oder absteigendes Sortieren versehen werden, voreingestellt ist *asc* (Listing 3.53).

```
// Länder absteigend nach Einwohnerzahl sortiert
select c from Country c order by c.population desc
```

Listing 3.53: Anordung des Selektionsergebnisses

Constructor Expressions

Werden mehrere Werte in der Select-Klausel angegeben, wird also ein Tupel von Werten selektiert, so wird dieses Tupel wie oben dargestellt in ein *Object[]*-Objekt verpackt. Damit geht ein Teil Übersichtlichkeit und Fehlersicherheit im Programmcode verloren, da man in der weiteren Verarbeitung des *Object*-Arrays „wissen" muss, unter welchem Index man welches fachliche Datum mit welchem konkreten Typ abgreifen kann (Listing 3.54).

```
EntityManager em = …
TypedQuery<Object[]> query = em.createQuery(
  "select c.continent, avg(c.population)"
  + " from Country c group by c.continent", Object[].class);
for (Object[] continentDescription : query.getResultList())
{
  Continent continent = (Continent) continentDescription[0];
  Number averagePopulation = (Number) continentDescription[1];
  …
```

Listing 3.54: Selektion von Tupeln als Object-Arrays

Mithilfe einer sog. Constructor Expression in der Select-Klausel können die Selektion und Weiterverarbeitung typsicherer und „sprechender" gestaltet werden. Dazu ist eine Hilfsklasse nötig, die einen zu den selektierten Werten passenden Konstruktor anbietet (Listing 3.55).

```
public class ContinentDescription
{
  public ContinentDescription(Continent continent, Number avgPopulation)
  {
  …
```

Listing 3.55: Hilfsklasse für die Selektion einer Constructor Expression

Eine Constructor Expression ist nun die Nutzung des beschriebenen Konstruktors in der Select-Klausel. Die Syntax entspricht einem normalen Kostruktoraufruf im Java-Code inklusive des Operators *new*, allerdings muss hier der Klassenname voll qualifiziert angegeben werden. Die Query liefert dann statt *Object[]* Objekte des Hilfstyps (Listing 3.56).

```
EntityManager em = …
TypedQuery<Object[]> query = em.createQuery(
  "select new de.….ContinentDescription(c.continent, avg(c.population))"
  + " from Country c group by c.continent", ContinentDescription.class);
for (ContinentDescription continentDescription : query.getResultList())
{
  Continent continent = continentDescription.getContinent();
  Number averagePopulation = continentDescription.getAveragePopulation();
…
```

Listing 3.56: Selektion von Tupeln mithilfe einer Constructor Expression

Paging

Eine Query, die ein Listenergebnis liefert, kann auf einen bestimmten Teil des Ergebnisses eingeschränkt werden, z. B. auf die Anzahl Einträge, die sich auf einer Dialogseite o. ä. anzeigen lassen. Dazu bietet *Query* die Methoden *setFirstResult* und *setMaxResults* an, mit

denen die Startposition in der Liste – beginnend mit *0* – und die Anzahl der gelieferten Einträge angegeben werden können (Listing 3.57).

```
// Länder nach Namen sortiert, Einträge 20 bis 29
EntityManager em = …
TypedQuery<Country> query
  = em.createQuery("select c from Country c order by c.name",
                   Country.class);
query.setFirstResult(20);
query.setMaxResults(10);
List<Country> countries = query.getResultList();
```

Listing 3.57: Einschränken der Treffermenge einer Query

Query Hints

Java Persistence hat natürlich den Anspruch, die Entwicklung von DB-Zugriffen möglichst unabhängig von Provider und Datenbank zu gestalten. Dennoch hat man sich an einigen Stellen ein Hintertürchen offen gelassen, so auch bei Queries über die sog. Hints. Diese können mithilfe der Methode *setHint* als Schlüssel/Wert-Paare der Query hinzugefügt werden. Der einzige derzeit in der Spezifikation definierte Hint hat den Schlüssel *javax.persistence.query.timeout* und setzt eine Maximalzeit für die Durchführung von Queries in Millisekunden (Listing 3.58).

```
// Query-Ausführung auf 1 s einschränken
TypedQuery<Country> query = …
query.setHint("javax.persistence.query.timeout", 1000);
List<Country> countries = query.getResultList();
```

Listing 3.58: Einschränkung der Query-Laufzeit mit einem Hint

Hints müssen vom Provider nicht beachtet werden. Darüber hinaus dürfen die Hersteller weitere Hints definieren, solange der Schlüssel nicht mit *javax.persistence* beginnt. Portable Anwendungen sollten nicht von der Wirksamkeit der Hints abhängig sein.

Flush-Modus

Wenn eine Query ausgeführt wird, werden normalerweise zunächst alle im Entity Manager befindlichen Objektänderungen der aktuellen Transaktion in die Datenbank geschrieben, damit die Query die veränderten Daten auch „sieht“. Dieses Verhalten kann mit der Methode *setFlushMode* beeinflusst werden, und zwar in *Query* für eine einzelne Query oder in *EntityManager* als Default für alle zukünftigen Queries. Die Methoden akzeptieren einen Parameter vom Typ *FlushModeType* – einem Aufzählungstyp mit zwei Konstanten:

- *FlushModeType.AUTO:* Flush der Änderungen vor Queries oder bei Commit
- *i* Flush der Änderungen nur bei Commit

Wird eine Query außerhalb von Transaktionen ausgeführt, werden keine Änderungen in die Datenbank geschrieben.

Fetch Joins

Im Abschnitt 3.3.6 wurde beschrieben, dass Relationsattibute ggf. *LAZY* geladen werden, d. h. sie werden nicht direkt mit dem sie enthaltenden Objekt geladen, sondern erst beim ersten Zugriff. Dieses Verhalten lässt sich für eine Query mit einem Fetch Join übersteuern. Die Syntax dazu ähnelt einem Outer Join, allerdings mit dem Zusatz *fetch* und ohne Angabe einer Query-Variablen.

```
// Normale Query; books werden nicht geladen (1:n-Relation, LAZY)
select p from Publisher p

// Query mit direktem Laden der books
select p from Publisher p left join fetch p.books
```

Listing 3.59: Fetch Join

Der Fetch Join ist auch als Inner Join möglich – ohne das Schlüsselwort *left* –, was aber meist weniger sinnvoll ist, da es hier nicht um eine Einschränkung der Ergebnismenge geht, sondern um das Übersteuern des Lazy Loadings.

Named Queries

Neben der bislang gezeigten Möglichkeit, JPQL-Ausdrücke dynamisch zu erstellen, d. h. als String-Parameter an *EntityManager.createQuery* zu übergeben, bietet Java Persistence auch an, Queries unter einem frei gewählten Namen vorzudefinieren und sich bei der Ausführung dann nur auf den Namen zu beziehen. Dazu dient die Annotation *@NamedQuery*, die einer Entity-Klasse (oder einer Mapped Superclass – von der später noch die Rede sein wird) mitgegeben wird. Sie verknüpft einen Namen mit einem JPQL-Ausdruck. Achtung: Der Name der Named Query muss für die Persistence Unit eindeutig sein, er ist also nicht etwa der Entity-Klasse zugeordnet, die annotiert wird (Listing 3.60).

```
@Entity
@Access(AccessType.FIELD)
@NamedQuery(name = "Country_findByPhonePrefix",
            query = "select c from Country c where c.phonePrefix=?1")
public class Country
{
  …
}
```

Listing 3.60: Definition einer benannten Query

Für die Ausführung einer Named Query verwendet man *EntityManager.createNamedQuery* in der gleichen Weise wie zuvor *EntityManager.createQuery* (Listing 3.61).

```
EntityManager em = …
TypedQuery<Country> query
  = em.createNamedQuery("Country_findByPhonePrefix", Country.class);
query.setParameter(1, phonePrefix);
Country country = query.getSingleResult();
```

Listing 3.61: Ausführung einer Named Query

Sollen mehrere Named Queries einer Klasse hinzugefügt werden, müssen die dazu nötigen *@NamedQuery*-Annotationen in eine *@NamedQueries*-Annotation eingepackt werden, da der Java-Sprachstandard ja nicht mehrere gleichnamige Annotationen an einem Platz erlaubt.

Die *@NamedQuery*-Annotation akzeptiert die optionalen Parameter *lockMode* und hints, mit denen der Query ein Lock-Modus und Query-Hints beigefügt werden können. Locking wird in einem späteren Abschnitt beschrieben.

Die Verwendung von Named Queries anstelle der dynamischen Queries hat verschiedene Vorteile. Zum einen kann man Named Queries an zentraler Stelle – z. B. bei der Entity-Klasse, auf die sie sich im Wesentlichen beziehen – sammeln und dadurch für mehr Überblick sorgen. Zum anderen können Named Queries ggf. vorweg analysiert und mit einem Ausführungsplan versehen werden, was aber abhängig vom genutzten Provider und der Datenbank ist.

3.4.2 Native Queries

Für die Arbeit mit persistenten Daten sollten die Möglichkeiten von JPQL eigentlich ausreichen. Nun ist „eigentlich“ in einem Satz – insbesondere in der Informationsverarbeitung – ein Signalwort, das auf Ausnahmen und höhere Aufwände hindeutet. Sollten Sie also mit JPQL nicht zum Ziel gelangen, können Sie SQL zum Aufbau von sog. Native Queries verwenden.

Die grundsätzliche Vorgehensweise entspricht dem zuvor gesagten: Mithilfe der Methode *EntityManager.createNativeQuery* wird ein Query-Objekt erstellt und mit *getResultList* oder *getSingleResult* ausgeführt (Listing 3.62).

```
EntityManager em = …
Query query = em.createNativeQuery(
  "SELECT NAME, POPULATION/AREA FROM CITY");
List<Object[]> resultList = query.getResultList();
for (Object[] entry : resultList)
{
  String name = (String) entry[0];
  Number populationDensity = (Number) entry[1];
  …
```

Listing 3.62: Aufbau und Ausführung einer SQL-Query

Der Query-Text darf auch die von JDBC bekannten Positionsparameter *'?'* enthalten. Sie können mit der Methode *Query.setParameter* mit konkreten Werten besetzt werden. Namensparameter werden nicht unterstützt.

Leider liefert *createNativeQuery* ‚nur' *Query* als Ergebnis, d. h. es stehen nicht wie bei *TypedQuery<T>* noch Informationen über den selektierten Typ zur Verfügung. Die Übernahme des Ausführungsergebnisses erzeugt also im Fall von *getResultList* eine Unchecked Warning, die man ggf. mit *@SuppressWarnings("unchecked")* unterdrückt, im Fall von *getSingleResult* ist sogar eine explizite Typwandlung nötig.

Native Queries bieten die Möglichkeit, beliebige SQL-Befehle auszuführen, also bspw. auch DB-spezifische Funktionen oder Stored Procedures zu nutzen. Diese Flexibilität kommt zu dem Preis, dass die im SQL genutzten Namen nun die DB-Namen sind, man also das Mapping zur Datenbank in jeder Query erneut berücksichtigen muss. Nutzt man darüber hinaus spezielle Eigenschaften der Zieldatenbank aus, wird ein DB-Wechsel aufwändig oder vielleicht sogar unmöglich.

Wie schon JPQL-Queries, lassen sich auch Native Queries vorformulieren. Dazu dient die Annotation *@NamedNativeQuery*, mit der auf Ebene einer Entity-Klasse (oder einer Mapped Superclass – siehe Abschnitt 3.5.4) SQL-Befehle mit einem Namen versehen werden können, der im Aufruf von *createNativeQuery* wiederum benutzt werden kann. *@NamedNativeQuery* akzeptiert den Parameter *resultSetMapping*, mit dem auf eine weitere Annotation, *@SqlResultSetMapping*, referenziert wird. Damit wird deklariert, welche Entities und Werte von der SQL-Query geliefert werden. Ein einfaches Beispiel zeigt Listing 3.63.

```
@Entity
@Access(AccessType.FIELD)
@NamedNativeQuery(
  name = "City_populationDensity",
  query = "SELECT NAME, POPULATION/AREA AS DENSITY FROM CITY",
  resultSetMapping = "City_populationDensity_Mapping")
@SqlResultSetMapping(
  name = "City_populationDensity_Mapping",
  columns = { @ColumnResult(name = "NAME"),
              @ColumnResult(name = "DENSITY") })
public class City
{
  …
```

Listing 3.63: Vorwegdefinition einer SQL-Query mit Mapping

Werden mehrere *@NamedNativeQuery*-Annotationen benötigt, müssen sie in die „Plural-Annotation" *@NamedNativeQueries* verpackt werden.

Die Ausführung einer Named Native Query geschieht analog zu der einer Named (JPQL) Query (Listing 3.64).

```
EntityManager em = …
Query query = em.createNamedQuery("City_populationDensity");
List<Object[]> resultList = query.getResultList();
…
```

Listing 3.64: Ausführung einer Named Native Query

Die Aussage der *@SqlResultSetMapping*-Annotation im gezeigten Beispiel ist, dass die von der SQL-Abfrage gelieferten Spaltenwerte *NAME* und *DENSITY* als Einzelwerte im Ergebnis zu finden sein werden. Weitere Möglichkeiten wären die Zuordnung der Spaltenwerte zu Entity-Objekten oder eine Kombination von Entity-Objekten und Einzelwerten. Ein solches komplexes Mapping wäre auch ohne Named Query mit einer Variante von *EntityManager.createNativeQuery* möglich, die als zweiten Parameter den Namen eines Result Set Mappings annimmt. Komplexe Mappings werden in der Praxis selten benötigt. Für Details dazu sei daher hier nur auf die Java-Persistence-Spezifikation, Abschnitt 3.8.15 (SQL Queries) verwiesen.

3.4.3 Criteria Queries

Java Persistence bietet mit JPQL eine leistungsfähige Abfragesprache an, deren Befehle aus Sicht des Java-Programms allerdings einfache Strings sind. Dadurch können sich leicht Fehler einschleichen, die in aller Regel erst zur Laufzeit erkannt werden. Neben so trivialen Fehlern wie vergessenen oder falsch geschriebenen Befehlswörtern sowie Objekt- und Attributnamen (Case-sensitiv!) kommen auch subtilere Fehlersituationen vor. So ist z. B. *select c from Country* fehlerhaft, da in JPQL verpflichtend eine Query-Variable genutzt werden muss, anders als in SQL, wo dies nur optional ist. Auch ist die Typsicherheit nicht hundertprozentig, denn kein Compiler kann prüfen, ob der in *createQuery* angegebene JPQL-Text zum Typparameter passt (Listing 3.65).

```
TypedQuery<City> query
  = em.createQuery("select c from Country c", City.class);
```

Listing 3.65: Mismatch zwischen JPQL und Typparameter

An dieser Stelle setzt das Criteria Query API an, eine Schnittstelle zum objektorientierten Aufbau von Queries. *EntityManager* liefert durch die Methode *getCriteriaBuilder* ein Objekt vom Typ *CriteriaBuilder*, das den Zugang zum Criteria Query API darstellt. Ein damit erzeugtes Query-Objekt wird anstelle des bisherigen JPQL-Texts verwendet. Die Ausführung der Query geschieht auf gleiche Weise wie zuvor (Listing 3.66).

```
EntityManager em = …
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
// Füllen des Query-Objektes
…
```

```
TypedQuery<Cocktail> query = em.createQuery(cQuery);
List<Cocktail> cocktails = query.getResultList();
```

Listing 3.66: Grundsätzliche Vorgehensweise zur Nutzung von Criteria Queries

Eine Anmerkung des Autors: Die Beispiele dieses Abschnitts verwenden die Entity-Klasse *Cocktail*. Ich habe mich natürlich gefragt, ob es politisch korrekt ist, indirekt Alkohol in einem Fachbuch zu verwenden. Die Beispieldaten enthalten aber auch einen nichtalkoholischen Cocktail!

Query Roots und Selektion

Für eine Abfrage muss zunächst bestimmt werden, auf welche Entities sie sich bezieht. Diese sog. Query Roots entsprechen des Query-Variablen der From-Klausel in JPQL. Sie werden der *CriteriaQuery* mithilfe der Methode *from* hinzugefügt.

Sollen die durch ein Query Root repräsentierten Objekte Ergebnis der Query sein, wird das Root-Objekt der Query mittels *select* als Selektion übergeben. Eine mit diesen beiden Mitteln aufgebaute Query, die alle Einträge der Entity als Ergebnis liefert, zeigt Listing 3.67.

```
// Criteria Query äquivalent zu "select c from Cocktail c"
EntityManager em = …
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.select(c);
TypedQuery<Cocktail> query = em.createQuery(cQuery);
…
```

Listing 3.67: Criteria Query zur Selektion aller Objekte einer Entity-Klasse

An dem Beispiel wird deutlich, dass der Programmaufwand im Vergleich zu einer in JPQL formulierten Query zwar signifikant höher ist, dafür aber auch mehr Schutz vor Fehlern geboten wird. So muss bspw. der Parameter von *select* zum Typ der Query passen.

Der Programmaufwand kann übrigens etwas verringert werden, da die Methoden in den allermeisten Fällen einen Return-Wert liefern, mit dem direkt die nächste Methode aufgerufen werden kann – eine Ausprägung eines sog. Fluent API (Listing 3.68).

```
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
TypedQuery<Cocktail> query
  = em.createQuery(cQuery.select(cQuery.from(Cocktail.class)));
…
```

Listing 3.68: Alternatives „Fluent API“

Welche Schreibweise Sie übersichtlicher finden, ist natürlich Ihnen überlassen. Im Folgenden wird nicht vom Fluent API Gebrauch gemacht, da Einzelanweisungen im Buch etwas „satzfreundlicher“ sind.

Neben einzelnen kompletten Objekten können mithilfe von Criteria Queries auch mehrere Werte – Entities, Attribute, berechnete Werte – selektiert werden. Details dazu folgen später.

Attributzugriffe

Auf die Teile eines Entity-Objekts kann ausgehend vom Query Root mithilfe der Methode *get* zugegriffen werden. Sie liefert ein Objekt des Typs *Path* als Repräsentant für das referenzierte Attribut. *Path*-Objekte können dann bspw. zur Selektion verwendet werden (Listing 3.69).

```
CriteriaQuery<String> cQuery = cBuilder.createQuery(String.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
Path<String> cName = c.get("name");
cQuery.select(cName);
TypedQuery<String> query = em.createQuery(cQuery);
…
```

Listing 3.69: Zugriff auf Attribute über ihren Namen

Auf ein *Path*-Objekt lässt sich wieder *get* anwenden, sodass man damit auf Teile von eingebetteten Objekten oder 1:1- bzw. n:1-Relationsattributen zugreifen kann.

Achtung: Die im Beispiel genutzte Version von *get* erhält den gewünschten Attributnamen als *String*-Parameter. Damit ist zur Compile-Zeit wieder keine Prüfung möglich, ob das Attribut überhaupt existiert!

Statisches Metamodell

Das beschriebene Problem lässt sich lösen, wenn schon zur Übersetzungszeit statische Informationen zum dynamischen Aufbau der Entity-Objekte vorliegen. Anders als z. B. C++ bietet Java diese statischen Metadaten nicht von Haus aus an, sodass sie mit einem entsprechenden Werkzeug generiert werden müssen. Zu jeder persistenten Klasse *E* wird dabei eine Metadatenklasse *E_* erzeugt, die für jedes persistente Attribut von *E* eine statische Variable gleichen Namens enthält, deren Typ Auskunft über die Art des persistenten Attributs gibt (Listing 3.70).

```
@Entity
@Access(AccessType.FIELD)
public class Cocktail
{
  @Id @GeneratedValue
  private Integer    id;
```

```
  private String     name;

  @ManyToMany
  private Set<CocktailZutat> zutaten = new HashSet<CocktailZutat>();
  …
}

@StaticMetamodel(Cocktail.class)
public abstract class Cocktail_
{
  public static volatile SingularAttribute<Cocktail, Integer> id;
  public static volatile SingularAttribute<Cocktail, String> name;
  public static volatile SetAttribute<Cocktail, CocktailZutat> zutaten;
}
```

Listing 3.70: Metamodellklasse zu einer Entity-Klasse

Der Name *E_* entspricht nur einer Namenskonvention. Die Verbindung zwischen den beiden Klassen wird über die Annotation *@StaticMetamodel* hergestellt, was für den hier gezeigten Einsatz der Metamodellklasse aber unerheblich ist.

Die Metadaten enthalten offensichtlich Informationen über die betreffende Klasse und den Typ der Attribute. Die weiteren Details werden hier übergangen, da sie für das weitere Verständnis nicht wichtig sind. Bei Interesse schauen Sie in den Abschnitt 5 (Metamodel API) der Java-Persistence-Spezifikation.

Die Metamodellklassen lassen sich mit einem Annotation Processor generieren, z. B. dem Hibernate-Modellgenerator. Seine Benutzung ist natürlich abhängig vom genutzten Build-System und daher hier nicht allgemeingültig darstellbar. Das Beispielprojekt enthält dazu entsprechende Maven-Plug-in-Aufrufe (Listing 3.71). In Eclipse kann der Annotation Processor in den Projekteigenschaften im Abschnitt Java Compiler | Annotation Processing eingetragen werden (Abb. 3.20).

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <inherited>true</inherited>
      <configuration>
        <generatedSourcesDirectory>
          target/generated-sources/annotations
        </generatedSourcesDirectory>
      </configuration>
    </plugin>

    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>build-helper-maven-plugin</artifactId>
```

```
      <inherited>true</inherited>
      <executions>
        <execution>
          <id>add-source</id>
          <phase>generate-sources</phase>
          <goals>
            <goal>add-source</goal>
          </goals>
          <configuration>
            <sources>
              <source>target/generated-sources/annotations</source>
            </sources>
          </configuration>
        </execution>
      </executions>
    </plugin>
```

Listing 3.71: Ausschnitt aus der Maven-Projektkonfiguration zur Erzeugung des Metamodells

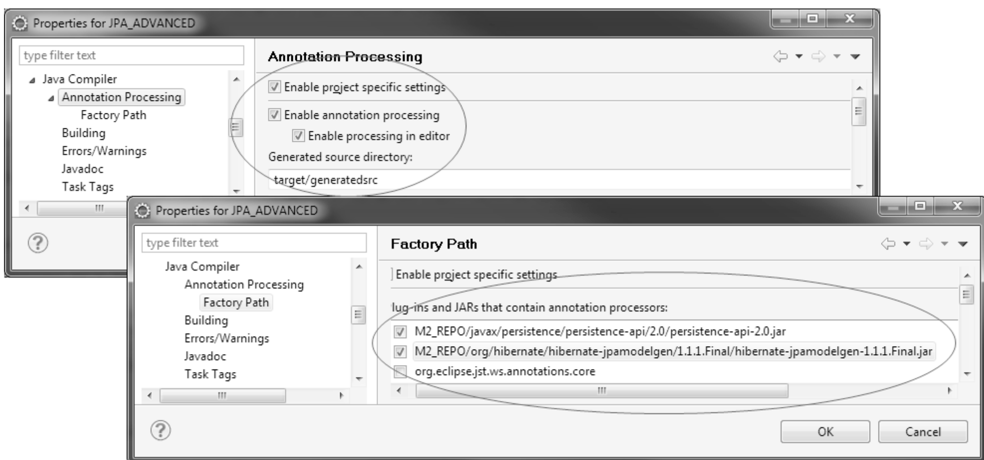


Abbildung 3.20: Eclipse-Projektkonfiguration zur Generierung von Metamodellklassen

Mithilfe des statischen Metamodells lässt sich die oben gezeigte Anweisung zum Zugriff auf ein Attribut nun sicherer gestalten (Listing 3.72).

```
Path<String> cName = c.get(Cocktail_.name);
```

Listing 3.72: Attributzugriff mittels Metamodellattribut

Bedingungen

CriteraBuilder bietet diverse Methoden an, mit denen sich Attribute – in Form von *Path*-Objekten – und andere Werte zu Prädikaten, d. h. Bedingungsobjekten, kombinieren las-

sen. So prüft ein mit *CriteriaBuilder.equal* erstelltes Prädikat seine beiden Parameter auf Gleichheit. Analog funktionieren *greaterThan*, *isFalse*, *isNotNull* etc.

Ein oder mehrere Prädikate (implizit Und-verknüpft) lassen sich der Query als Where-Bedingung mithilfe der Methode *where* hinzufügen (Listing 3.73).

```
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
Path<String> cName = c.get(Cocktail_.name);
Predicate ipanema = builder.equal(cName, "Ipanema");
cQuery.where(ipanema);
…
```

Listing 3.73: Nutzung eines Prädikats als Where-Bedingung

Joins

Für die Navigation in Relationen werden *Join*-Objekte benötigt. Sie werden ausgehend von einem Query Root mithilfe der Methode *join* erstellt – am besten wieder unter Verwendung statischer Metamodelldaten – und können in der Folge wiederum wie Query Roots genutzt werden, z. B. um darauf Prädikate zu definieren (Listing 3.74).

```
// Criteria Query äquivalent zu
// select distinct c from Cocktail c JOIN c.zutaten z where z.volProz<>0"
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
Join<Cocktail, Zutat> z = c.join(Cocktail_.zutaten);
Path<Double> zVolProz = z.get(Zutat_.volProz);
Predicate enthaeltAlkohol = cBuilder.equal(zVolProz, 0);
cQuery.where(enthaeltAlkohol);
cQuery.select(c);
cQuery.distinct(true);
TypedQuery<Cocktail> query = em.createQuery(cQuery);
```

Listing 3.74: Nutzung eines Joins in einer Criteria Query

Das Beispiel zeigt auch die Anwendung von *distinct* zur Vermeidung von Dubletten im Ergebnis.

Die Methode *join* akzeptiert als zweiten Parameter einen Wert aus der Aufzählung *JoinType*:

- *JoinType.INNER:* Normaler (innerer) Join
- *JoinType.LEFT:* Left Outer Join
- *JoinType.RIGHT:* Right Outer Join

Die Variante mit nur einem Parameter nutzt implizit *JoinType.INNER*. Right Outer Joins müssen in JPA 2.0 nicht unterstützt werden.

Parameter

CriteriaBuilder bietet mit der Methode *parameter* die Möglichkeit an, Query-Parameter in die Query einzubauen, z. B. als Teil eines Vergleichs. Diese werden vor der Ausführung der Query wie gewohnt mit aktuellen Werten besetzt (Listing 3.75).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.where(cBuilder.equal(c.get(Cocktail_.name),
            cBuilder.parameter(String.class, "name")));
cQuery.select(c);
TypedQuery<Cocktail> q = em.createQuery(cQuery);
q.setParameter("name", name);
```

Listing 3.75: Criteria Query mit Parameter

Tupel-Selektion

Zur Selektion mehrerer Werte bietet das Criteria Query API wie schon JPQL zwei Varianten an. Zum einen kann man sich des Query-Typs *Tuple* bedienen. Ein *Tuple*-Objekt ist ein Container für mehrere Teilobjekte, die daraus per Index oder Aliasnamen geliefert werden können. Für die Selektion wird dann statt *select* die Methode *multiSelect* verwendet, die eine unbrenzte Menge von Einzelselektionswerten als Parameter annimmt. Dabei kann man den einzelnen Selektionswerten mithilfe der Methode *alias* einen Aliasnamen mitgeben, den man später bei der Verarbeitung der Ergebnismenge nutzt (Listing 3.76).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cQuery = cBuilder.createQuery(Tuple.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.multiselect(c.get(Cocktail_.name).alias("cocktailName"),
                 c.get(Cocktail_.basisZutat).get(CocktailZutat_.name));
TypedQuery<Tuple> query = em.createQuery(cQuery);
for (Tuple entry : query.getResultList())
{
  String name = entry.get("cocktailName", String.class);
  String basisZutat = entry.get(1, String.class);
  …
```

Listing 3.76: Tuple-Selektion

Die Alternative ist wie bei JPQL die Nutzung einer Constructor Expression. *CriteriaBuilder* bietet die Methode *construct* an, mit der die Constructor Expression erstellt werden kann (Listing 3.77).

entwickler.press

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<NameAndBasisZutat> cQuery
  = cBuilder.createQuery(NameAndBasisZutat.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.select(cBuilder.construct(NameAndBasisZutat.class,
  c.get(Cocktail_.name),
  c.get(Cocktail_.basisZutat).get(CocktailZutat_.name)));
TypedQuery<NameAndBasisZutat> query = em.createQuery(cQuery);
for (NameAndBasisZutat entry : query.getResultList())
{
  String name = entry.getName();
  String basisZutat = entry.getBasisZutat();
  …
```

Listing 3.77: Criteria Query mit Constructor Expression

Funktionen

Für die im Abschnitt über JPQL beschriebenen Funktionen – inklusive der Aggregationsfunktionen – bietet CriteriaBuilder gleichnamige Methoden an, mit denen Selektions- oder Vergleichsausdrücke zusammengestellt werden können. Ein Beispiel folgt im nächsten Abschnitt. Dort wird die Aggregationsfunktion *avg* eingesetzt.

Gruppierung

Mithilfe der Methode *groupBy* können einer *CriteriaQuery* ein oder mehrere Gruppierungsparameter hinzugefügt werden. Wie bei JPQL beschrieben, wird die Ergebnismenge dann so verdichtet, dass für jede Kombination der Gruppierungsparameterwerte nur eine Ergebniszeile erscheint. Die Selektion muss dementsprechend aus für die jeweiligen Gruppen konstanten Werten oder Aggregationswerten bestehen. Sollen die Gruppenergebnisse weiter eingeschränkt werden, kann dies mit der Methode *having* geschehen, die so wirkt wie *where* in Bezug auf die nicht gruppierten Werte (Listing 3.78).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Tuple> cQuery = cBuilder.createTupleQuery();
Root<Cocktail> c = cQuery.from(Cocktail.class);
Join<Cocktail, Zutat> z = c.join(Cocktail_.zutaten);
Path<String> cName = c.get(Cocktail_.name);
Path<Double> zVolProz = z.get(Zutat_.volProz);
cQuery.multiselect(cName, cBuilder.avg(zVolProz));
cQuery.groupBy(cName);
TypedQuery<Tuple> q = em.createQuery(cQuery);
…
```

Listing 3.78: Criteria Query mit Gruppierung

Sortierung

Die Anordnung des Query-Ergebnisses kann mithilfe der Methode *orderBy* festgelegt werden. Sie erhält ein oder mehrere Sortierangaben als Parameter, die wiederum durch die Methoden *CriteriaBuilder.asc* oder *CriteriaBuilder.desc* für auf- bzw. absteigende Sortierung erstellt werden (Listing 3.79).

```
CriteriaBuilder cBuilder = em.getCriteriaBuilder();
CriteriaQuery<Cocktail> cQuery = cBuilder.createQuery(Cocktail.class);
Root<Cocktail> c = cQuery.from(Cocktail.class);
cQuery.select(c);
cQuery.orderBy(cBuilder.asc(c.get(Cocktail_.name)));
TypedQuery<Cocktail> query = em.createQuery(cQuery);
…
```

Listing 3.79: Sortierung des Selektionsergebnisses einer Criteria Query

Fetch Joins

Wie bei JPQL, gibt es auch bei Criteria Queries eine Möglichkeit, Relationsattribute direkt laden zu lassen, auch wenn sie für Lazy Loading konfiguriert sind. Dazu bieten die Query Roots und Join-Objekte die Methode *fetch* an. Der erste Parameter bestimmt das *EAGER* zu ladende Attribut, der zweite gibt wie bei *join* den Join-Typ an (Listing 3.80).

```
Root<Publisher> p = cQuery.from(Publisher.class);
p.fetch(Publisher.books, JoinType.LEFT);
```

Listing 3.80: Fetch Join in einer Criteria Query

Wie schon bei JPQL-Fetch-Joins ausgeführt, ist der Join-Typ *LEFT* hier am sinnvollsten. Auf das Ergebnis der Methode *fetch* kann erneut *fetch* angewendet werden, sodass hier sogar mehrstufige Fetch-Joins möglich sind.

3.5 Vererbungsbeziehungen

Vererbung ist ein wichtiges Prinzip der Objektorientierung, mit dem man „Ist ein“-Beziehungen ausdrücken kann: Ein abgeleitetes Objekt ist ein Objekt der Basisklasse, d. h. es hat alle Eigenschaften und Funktionalitäten der Basisklasse, ergänzt um weitere. Technisch drückt sich die Vererbung auf der Attributebene dadurch aus, dass ein abgeleitetes Objekt alle Attribute seiner Basisklasse enthält.

Relationale Datenbanken haben ein solches Feature im Allgemeinen nicht. Diese konzeptionelle Lücke gilt es nun beim Mapping der Daten zu überbrücken. Dazu bietet Java Persistence drei verschiedene Strategien an, die im Folgenden anhand der bewusst einfach gehaltenen Klassen *Vehicle*, *Car* und *Ship* betrachtet werden sollen (Listing 3.81).

```
@Entity
@Access(AccessType.FIELD)
public abstract class Vehicle
{
  @Id @GeneratedValue
  private Integer id;
  private String  name;
  …
}

@Entity
@Access(AccessType.FIELD)
public class Car extends Vehicle
{
  private int noOfDoors;
  …
}

@Entity
@Access(AccessType.FIELD)
public class Ship extends Vehicle
{
  private double tonnage;
  …
}
```

Listing 3.81: Basisklasse „Vehicle“ mit davon abgeleiteten Klassen „Car“ und „Ship“

3.5.1 Mapping-Strategie „SINGLE_TABLE“

Diese Strategie ist vorgegeben, kann aber auch durch Annotation der Basisklasse mit *@Inheritance(strategy = InheritanceType.SINGLE_TABLE)* explizit gewählt werden (Listing 3.82). Die Daten der gesamten Ableitungshierarchie werden in einer einzelnen Tabelle abgespeichert. Eine zusätzliche Spalte – der Diskriminator – speichert für jeden Eintrag seinen Typ in Form des einfachen Klassennamens. Da die Tabelle Spalten für alle Attribute der beteiligten Klassen enthält, werden für einen Eintrag in aller Regel nicht alle benötigt. Die überzähligen Werte bleiben *null*.

```
@Entity
@Access(AccessType.FIELD)
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public abstract class Vehicle
{
  …
```

Listing 3.82: Explizite Wahl der Strategie zur Abbildung der Vererbung

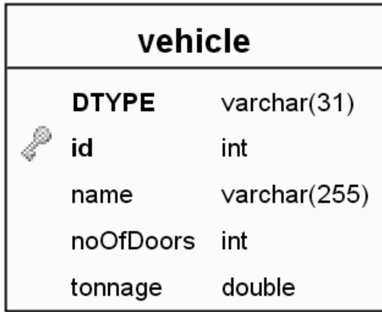
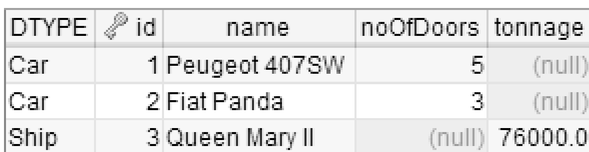


Abbildung 3.21: Tabelle bei Strategie „SINGLE_TABLE“



DTYPE	id	name	noOfDoors	tonnage
Car	1	Peugeot 407SW	5	(null)
Car	2	Fiat Panda	3	(null)
Ship	3	Queen Mary II	(null)	76000.0

Abbildung 3.22: Beispielinhalt bei Strategie „SINGLE_TABLE“

Die Diskriminatorwerte können auch explizit gewählt werden, und zwar mithilfe der Annotation *@DiscriminatorValue*.

Der Name und der Typ der Diskriminatorspalte können mithilfe der Annotation *@DiscriminatorColumn* bestimmt werden. Als Parameter *discriminatorType* wird dabei eine Konstante aus dem Aufzählungstyp *DiscriminatorType* übergeben: Neben dem Vorgabewert *STRING* sind hier auch *CHAR* und *INTEGER* möglich, um statt Strings einzelne Zeichen oder ganze Zahlen als Diskriminatoren zu verwenden. In diesen Fällen muss für die beteiligten Klassen jeweils ein passender Diskriminatorwert mittels *@DiscriminatorValue* angegeben werden (Listing 3.83).

```
@Entity
@Access(AccessType.FIELD)
@DiscriminatorColumn(name = "T",
                     discriminatorType = DiscriminatorType.INTEGER)
@DiscriminatorValue("1234")
public abstract class Vehicle
{
  …
}

@Entity
@Access(AccessType.FIELD)
@DiscriminatorValue("2345")
```

```
public class Car extends Vehicle
{
  …
}
```

Listing 3.83: Einsatz von „@DiscriminatorColumn“ und „@DiscriminatorValue“

Die Strategie *SINGLE_TABLE* hat den Vorteil des einfachen Mappings zur Datenbank: Es wird nur eine Tabelle für die gesamte Ableitungshierarchie benötigt. Das ist für die Performanz von Queries durchaus positiv, da alle zur Konstruktion eines Objekts nötigen Attributwerte in einer Tabellenzeile enthalten sind. So sind sowohl für Queries nach den „konkretesten“ Objekten des Ableitungsbaums (z. B. *"select c from Car c …"*) – wie auch für polymorphe Queries auf Basisklassenebene (z. B. *"select v from Vehicle v …"*) nur einfache SQL-Befehle nötig.

Nachteilig ist hingegen, dass die Tabelle recht breit werden kann, da sie ja sämtliche Attribute der beteiligten Klassen enthält. Weiterhin können kaum einschränkende Constraints wie *"not null"* auf die Spalten der Tabelle angewendet werden, da der überwiegende Teil nicht für alle Eintragstypen gefüllt ist. Schließlich können Fremdschlüssel in anderen Tabellen nur auf die oberste Basisklasse referenzieren: Ein Foreign Key auf *Ship* ist in der Datenbank nicht darstellbar, da für *Ship* ja keine (separate) Tabelle existiert.

3.5.2 Mapping-Strategie „TABLE_PER_CLASS“

Hier bekommt jede nicht abstrakte Klasse der Ableitungshierarchie eine Tabelle zugeordnet (Abb. 3.23, Abb. 3.24).

car	
id	int
name	varchar(255)
noOfDoors	int

ship	
id	int
name	varchar(255)
tonnage	double

Abbildung 3.23: Tabellen bei Strategie „TABLE_PER_CLASS“

id	name	noOfDoors
1	Peugeot 407SW	5
2	Fiat Panda	3

id	name	tonnage
3	Queen Mary II	76000.0

Abbildung 3.24: Beispielinhalt bei Strategie „TABLE_PER_CLASS“

Jedes konkrete Java-Objekt ist nun in einem Tabelleneintrag ohne weiteren Ballast abgespeichert. Not-Null-Constraints können darin problemlos deklariert werden. Ebenso

sind Fremdschlüsselbeziehungen auf die konkreten Objekte möglich, nicht aber auf Basisklassen.

Während konkrete Queries hier wiederum performant umgesetzt werden können, sind für polymorphe Queries komplexere SQL-Befehle wie z. B. *UNION* notwendig, was sich je nach DB-Fabrikat stark bemerkbar macht. Sie sollten diese Strategie also vermeiden, wenn die Anwendung häufiger Basisklassenabfragen durchführt.

Eine Einschränkung besteht auch in Bezug auf automatisch generierte Schlüssel. Der Identity-Generator ist hier prinzipbedingt nicht einsetzbar, da für die betroffenen Klassen übergreifend eindeutige ID-Werte generiert werden müssen, der Identity-Generator dies aber nur für jede Tabelle tun würde[7].

Die Spezifikation verlangt in der aktuellen Version 2.0 nicht, dass die Strategie *TABLE_PER_CLASS* unterstützt wird. Das ist aber ein nur theoretischer Nachteil, da alle mir bekannten Provider dies tun.

3.5.3 Mapping-Strategie „JOINED“

Diese Strategie legt die Daten der Ableitungshierarchie voll normalisiert ab: Jede Klasse hat eine ihr zugeordnete Klasse, die neben dem Primärschlüssel nur die in ihr deklarierten Attribute enthält (Abb. 3.25, Abb. 3.26)[8].

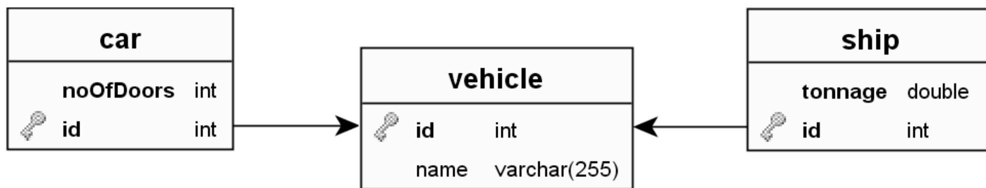


Abbildung 3.25: Tabellen bei Strategie „JOINED“

id	noOfDoors
1	5
2	3

id	name
1	Peugeot 407SW
2	Fiat Panda
3	Queen Mary II

id	tonnage
3	76000.0

Abbildung 3.26: Beispielinhalt bei Strategie „JOINED“

7 Wenn Sie die Generatorstrategie *AUTO* benutzen, wählt Hibernate (zumindest bis zur Version 4.0.0) bei einer Zieldatenbank, die Identity Columns unterstützt, den Generator *IDENTITY*, obwohl das für eine Ableitungshierarchie nicht möglich ist. Um dies zu umgehen, setzen Sie explizit *SEQUENCE* oder *TABLE* ein.

8 Die Spezifikation erlaubt dem Provider die Nutzung einer Diskriminatorspalte wie bei *SINGLE_TABLE* auch bei der Strategie *JOINED*. Davon macht derzeit nach meiner Beobachtung nur Eclipselink Gebrauch.

Durch das normalisierte Tabellenlayout sind Constraints auf den Spalten und Fremdschlüsselbeziehungen jeglicher Art unproblematisch. Nachteilig ist allerdings, dass für jeden Objektzugriff die Inhalte mehrerer Tabellen miteinander verknüpft werden müssen, was sich in teilweise dramatisch geringerer Performanz niederschlägt.

3.5.4 Non-Entity-Basisklassen

Nicht immer hat eine Basisklasse auch eine fachliche Bedeutung. So könnte man bspw. eine Basisklasse für Entities schaffen, die als ID eine UUID erhalten, d. h. einen (weitgehend) global eindeutigen *String*. Die Eigenschaften der Klasse sollen denen einer normalen Entity entsprechen, es sollen also die gleichen Möglichkeiten des Mappings bestehen etc. Wegen der nicht vorhandenen fachlichen Bedeutung wird aber keine eigene Tabelle benötigt, es werden keine Abfragen auf diese Klasse gerichtet usw. Für diesen Zweck bietet Java Persistence die Annotation *@MappedSuperclass*, die anstelle von *@Entity* verwendet wird. Im Beispiel stellt *UuidEntity* die UUID-basierte ID zusammen mit einem entsprechenden Getter und den Standardmethoden *equals* und *hashCode* zur Verfügung, sodass *Department* durch Ableitung auf einfache Art und Weise erstellt werden kann (Listing 3.84).

```
@MappedSuperclass
@Access(AccessType.FIELD)
public abstract class UuidEntity
{
  @Id
  protected String id;

  public UuidEntity()
  {
    this.id = java.util.UUID.randomUUID().toString();
  }

  public String getId() { … }
  public boolean equals(Object obj) { … }
  public int hashCode() { … }
}

@Entity
@Access(AccessType.FIELD)
public class Department extends UuidEntity
{
  private String name;
  …
}
```

Listing 3.84: Einsatz von „@MappedSuperclass“

@MappedSuperclass findet man häufig in Kombination mit *abstract*, was aber keine Bedingung ist. *@MappedSuperclass* darf in einer Ableitungshierarchie auch zwischendrin vorkommen – wenngleich das auch eine recht unübliche Situation ist.

Vorsicht ist geboten, wenn man eine Entity-Klasse von einer Klasse ableitet, die nicht mit *@Entity* oder *@MappedSuperclass* markiert ist: Deren Attribute werden nicht persistent eingebunden, verhalten sich also so, als wären sie *@Transient*!

3.6 Dies und das

3.6.1 Secondary Tables

Bislang war einer persistenten Klasse genau eine Tabelle zugeordnet, in der alle Attribute mit Ausnahme von *Collection*s abgespeichert wurden. Das muss aber nicht so sein: Eine Entity-Klasse kann neben ihrer Haupttabelle noch weitere Tabellen benutzen, die Attribute können also über mehrere Tabellen verteilt sein. Zur Deklaration einer Zusatztabelle dient die Annotation *@SecondaryTable*. Die Attribute der Klasse können dann der zweiten Tabelle zugeordnet werden, indem sie mit *@Column* annotiert werden und dabei der Tabellenname angegeben wird. Attribute ohne Tabellenangabe liegen in der Haupttabelle der Entity (Listing 3.85).

```
@Entity
@Access(AccessType.FIELD)
@Table(name = "EEDEMOS_COUNTRY")
@SecondaryTable(name = "EEDEMOS_COUNTRY_EXT")
public class Country
{
  @Id
  private String isoCode;

  private String name;

  @Column(table = "EEDEMOS_COUNTRY_EXT")
  private String phonePrefix;

  @Column(table = "EEDEMOS_COUNTRY_EXT")
  private String carCode;
  …
```

Listing 3.85: Deklaration einer Zusatztabelle

Die Zuordnung geschieht dabei über gemeinsame Primärschlüsselwerte. Entities mit Secondary Tables stellen somit in etwa eine implizite 1:1-Relation mit Join über die Primärschlüssel dar (Abb. 3.27).

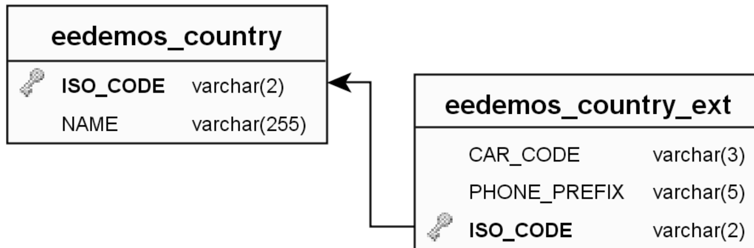


Abbildung 3.27: Tabellen zu Listing 3.85

Falls nötig, kann der Name der Primärschlüsselspalte(n) der Zusatztabelle angepasst werden. Dazu wird *@SecondaryTable* der Parameter *pkJoinColumns* mitgegeben.

Soll mehr als nur eine Zusatztabelle verwendet werden, müssen die entsprechenden *@SecondaryTable*-Annotationen in eine *@SecondaryTables*-Annotation verpackt werden (Listing 3.86).

```
@Entity
@Access(AccessType.FIELD)
@Table(name = "EEDEMOS_COUNTRY")
@SecondaryTables({
  @SecondaryTable(name = "EEDEMOS_COUNTRY_EXT"),
  @SecondaryTable(name = "EEDEMOS_COUNTRY_EXT2",
                pkJoinColumns = @PrimaryKeyJoinColumn(name = "IC"))
})
public class Country
{
  …
```

Listing 3.86: Zuordnung mehrerer Zusatztabellen

3.6.2 Zusammengesetzte IDs

Es ist nicht immer möglich, auf mehrteilige IDs zu verzichten. In einem solchen Fall muss eine ID-Klasse entworfen werden, die die gewünschten ID-Attribute enthält. Die Klasse muss serialisierbar sein, einen parameterlosen Konstruktor sowie gültige Implementierungen von *equals* und *hashCode* besitzen (Listing 3.87).

```
@Embeddable
@Access(AccessType.FIELD)
public class BranchId implements Serializable
{
  private int companyId;
  private int branchNo;

  public boolean equals(Object obj) { … }
```

```
    public int hashCode() { … }

    // Getter, Setter, Konstruktoren, …
  }
```

Listing 3.87: ID-Klasse

In der Entity-Klasse kann die ID-Klasse nun zur Deklaration eines ID-Attributs genutzt werden. Anstelle der sonst üblichen Annotation *@Id* wird nun *@EmbeddedId* verwendet (Listing 3.88).

```
@Entity
@Access(AccessType.FIELD)
public class SuperMarket
{
  @EmbeddedId
  private BranchId id;
  …
```

Listing 3.88: Eingebettetes ID-Attribut

Alternativ können die ID-Attribute auch in der Entity-Klasse einzeln aufgeführt werden. Die ID-Klasse muss dann mittels *@IdClass* deklariert werden. Sie benötigt für dieses Szenario die *@Embeddable*-Annotation nicht unbedingt. Wichtig ist hier die Namensgleichheit der Attribute in der ID-Klasse und der Entity-Klasse (Listing 3.89). Dieses Verfahren wird *Composite ID* genannt.

```
@Entity
@Access(AccessType.FIELD)
@IdClass(BranchId.class)
public class SuperMarket
{
  @Id
  private int    companyId;

  @Id
  private int    branchNo;
  …
```

Listing 3.89: Mehrfache Anwendung von „@Id“

3.6.3 Dependent IDs

Bei mehrteiligen IDs – aber nicht nur da – kommt es häufig vor, dass ein ID-Attribut selbst wieder eine Relation (1:1 oder n:1) zu einer anderen Entity darstellt. Schon das im vorigen Abschnitt verwendete Beispiel wäre in der Praxis vermutlich in der Art aufgebaut, dass

SuperMarket eine n:1-Beziehung zu einer Entity *Company* hätte, deren ID wiederum Teil des zusammengesetzten Schlüssels von *SuperMarket* wäre.

Eine solche Relation innerhalb der ID kann seit JPA 2.0 deklariert werden, wobei die konkrete Vorgehensweise von der Art der mehrteiligen ID abhängt.

Bei einer Composite ID, d. h. im Fall der Wiederholung der Attribute der ID-Klasse in der Entity-Klasse, wird das betroffene Attribut als *@OneToOne*- bzw. *@ManyToOne*-Relationsattribut deklariert. Die Namen der Attribute müssen wiederum übereinstimmen, während der Typ in der ID-Klasse der Typ des ID-Attributs der referenzierten Entity sein muss (Listing 3.90).

```
@Entity
@Access(AccessType.FIELD)
public class Department extends UuidEntity
{
  // erbt von UuidEntity eine String-ID
  …
}

@Entity
@Access(AccessType.FIELD)
@IdClass(ProjectId.class)
public class Project
{
  @Id @ManyToOne
  private Department department;

  @Id
  private String    prjId;
  …
}

@Embeddable
@Access(AccessType.FIELD)
public class ProjectId implements Serializable
{
  private String department;
  private String prjId;
  …
}
```

Listing 3.90: Relationsattribut als Teil einer Composite ID

Etwas unglücklich ist hier, dass aufgrund der Namensgleichheit nur eines der aufeinander bezogenen Attribute fachlich korrekt benannt werden kann: In der ID-Klasse des Beispiels etwa wäre statt des Feldnames *department* wohl eher *departmentId* sinnvoll. Dieser kleine Nachteil lässt sich natürlich durch passend benannte Zugriffsmethoden ausgleichen.

Arbeitet man dagegen mit einer Embedded ID, wird ein zusätzliches Relationsattribut benötigt, das mit der Annotation *@MapsId* dem entsprechenden ID-Attribut zugeordnet wird (Listing 3.91).

```
@Entity
@Access(AccessType.FIELD)
public class Department extends UuidEntity
{
  // erbt von UuidEntity eine String-ID
  …
}

@Entity
@Access(AccessType.FIELD)
@IdClass(ProjectId.class)
public class Project
{
  @EmbeddedId
  private ProjectId  id;

  @ManyToOne @MapsId("departmentId")
  private Department department;
  …
  public Project_EmbeddedId(Department department, String prjId, …)
  {
    this.id = new ProjectId(department.getId(), prjId);
    this.department = department;
    …
  }
}

@Embeddable
@Access(AccessType.FIELD)
public class ProjectId implements Serializable
{
  private String departmentId;
  private String prjId;

  public ProjectId(String departmentId, String prjId)
  {
    this.department = departmentId;
    this.prjId = prjId;
  }
  …
}
```

Listing 3.91: Relationsattribut als Teil einer Embedded ID

Hier ist man in der Benennung der Attribute flexibler, dafür gibt es nun einen kleinen Fallstrick: Für die Speicherung des Fremdschlüsselwerts in die Datenbanktabelle ist das

Relationsattribut zuständig. Im Beispiel muss also z. B. vor dem Aufruf von *persist* das Attribut *Project.department* besetzt sein. Während der Aktion des Entity Managers wird dann *Project.id.departmentId* passend gesetzt und in der Folge in der entsprechenden Spalte abgespeichert. Zwischen der Besetzung des Relationsattributs und dem Speichern des Objekts ist also der zugehörige Teil der ID nicht (richtig) gesetzt. Es empfiehlt sich daher, beide Attribute konsistent zu setzen. Die Beispielklasse tut dies in ihrem Konstruktor.

3.6.4 Locking

Enterprise-Anwendungen werden i. d. R. nicht von nur einem Benutzer verwendet. Vielmehr greifen ggf. mehrere Nutzer gleichzeitig auf die gleichen Daten zu. Dabei kann es zu Konflikten kommen, evtl. sogar zu inkonsistenten Daten, wenn zwei Änderungen zeitlich verzahnt durchgeführt werden. Betrachten wir einmal das folgende Szenario:

- User A liest Objekt x aus der DB
- User B liest Objekt x aus der DB
- User A ändert seine Kopie von x und speichert sie ab
- User B ändert seine Kopie von x und speichert sie ab

Die Änderungen von B sind am Ende dauerhaft gespeichert, wenn keine der im Folgenden beschriebenen Schutzmaßnahmen ergriffen wird. Problematisch ist nun nicht, dass B „gewinnt" (das sieht A vielleicht anders …), sondern dass er seine Änderungen zu einem Zeitpunkt abspeichert, zu dem die Voraussetzungen dafür ggf. gar nicht mehr vorliegen, ohne dass es dabei eine Warnung oder Fehlermeldung gibt.

Abhilfe schafft eine Blockierung der zu verarbeitenden Daten, wofür Java Persistence zwei Verfahren anbietet: Optimistic und Pessimistic Locking.

Das Setzen der Blockierung kann beim Lesen von Objekten geschehen. Dazu gibt es Varianten der Entity-Manager-Methoden *find* und *refresh*, die einen Parameter vom Typ *LockModeType* annehmen. Für eine *Query* kann vor ihrer Ausführung die Methode *setLockMode* aufgerufen werden (Listing 3.92).

```
EntityManager em = …
someEntity = em.find(…, LockModeType.PESSIMISTIC_WRITE);
em.refresh(…, LockModeType.NONE);

Query q = …
q.setLockMode(LockModeType.PESSIMISTIC_READ);
```

Listing 3.92: Anfordern eines Lock-Modus beim Lesen von Objekten

Für bereits gelesene Daten kann die Methode *lock* im *EntityManager* aufgerufen werden (Listing 3.93).

```
EntityManager em = …
someEntity = em.find(…);
em.lock(someEntity, LockModeType.PESSIMISTIC_WRITE);
```

Listing 3.93: Anfordern eines Lock-Modus für ein bereits gelesenes Objekt

Die folgenden *LockModeType*s stehen zur Verfügung:

- *LockModeType.NONE:* Keine Blockierung verwenden
- *LockModeType.OPTIMISTIC:* Optimistic Locking verwenden; das ist die Vorgabeeinstellung
- *LockModeType.OPTIMISTIC_FORCE_INCREMENT:* Wie OPTIMISTIC, allerdings wird das Versionsattribut (s. u.) inkrementiert bzw. aktualisiert, ohne dass zwingend eine Änderung der Daten vorgenommen wurde
- *LockModeType.PESSIMISTIC_READ:* Der Eintrag in der Tabelle wird mit einem Shared Lock versehen
- *LockModeType.PESSIMISTIC_WRITE:* Der Tabelleneintrag wird mit einem Exclusive Lock gesperrt
- *LockModeType.PESSIMISTIC_FORCE_INCREMENT:* Wie PESSIMISTIC_WRITE, allerdings mit gleichzeitiger Erhöhung bzw. Aktualisierung des Versionsattributs (s. u.)

Aus Kompatibilitätsgründen existieren zwei weitere Konstanten in *LockModeType*, die in neuen Anwendungen nicht mehr verwendet werden sollen:

- *LockModeType.READ* entspricht *OPTIMISTIC*
- *LockModeType.WRITE* entspricht *OPTIMISTIC_FORCE_INCREMENT*

Mit Ausnahme von *NONE* dürfen diese Modi nur innerhalb einer Transaktion aktiviert werden. Die Blockierung bleibt dann bis zum Transaktionsende bestehen.

Optimistic Locking

Beim Optimistic Locking wird keine echte Blockierung des Eintrags in der Datenbank genutzt, sondern statt dessen vor dem Abspeichern geprüft, ob die Daten in der Tabelle noch den Zustand haben, den sie beim Lesen zuvor hatten. Das wird in den allermeisten Fällen so sein, wodurch sich der Name des Verfahrens erklärt.

Der Vergleich der Daten geschieht allerdings nicht komplett. Vielmehr benötigt JPA dazu ein Versionsattribut, das i. A. zusätzlich zu den fachlichen Attributen in die Entity-Klasse und damit auch als Spalte in die Tabelle aufgenommen werden muss. Das Versionsattribut muss den Typ *short*, *Short*, *int*, *Integer*, *long*, *Long* oder *java.sql.Timestamp* haben und die Annotation @*Version* tragen (Listing 3.94).

```
@Entity
@Access(AccessType.FIELD)
public class Xyz
{
  …
  @Version
  private long version;
  …
}
```

Listing 3.94: Versionsattribut

Vor jedem Speichern wird nun automatisch geprüft, ob der Wert des Versionsattributs in der Datenbank noch mit dem im zuvor gelesenen Objekt übereinstimmt. Wenn nicht, wird die Operation mit Auswurf einer *OptimisticLockException*[9] abgebrochen. Andernfalls wird die Speicherung durchgeführt und dabei das Versionsattribut inkrementiert bzw. auf die aktuelle Zeit gesetzt. Im oben skizzierten Szenario würde B beim Speichern seiner Änderungen eine Fehlermeldung erhalten und der von A abgelegte Zustand weiter gelten.

Es ist sicher verlockend, ein Versionsattribut vom Typ *Timestamp* einzusetzen, da man damit ohne weiteren Aufwand einen Änderungszeitstempel in der DB abgelegt bekommt. Hier ist aber Vorsicht angesagt, da das Verfahren nur mit einer Datenbank funktionieren kann, die Timestamps auf Millisekunden genau abspeichert. Sicherer ist die Verwendung eines ganzzahligen Versionsattributs.

In einer Entity-Klasse darf es nur ein Versionsattribut geben. Bei der Nutzung von Zusatztabellen muss das Versionsattribut in der Haupttabelle platziert werden.

Optimistic Locking wird als vorgegebenes Verfahren verwendet, sobald ein Versionsattribut im bearbeiteten Entity-Objekt existiert. Die explizite Wahl ist mit den beiden Lock-Modi *OPTIMISTIC* und *OPTIMISTIC_FORCE_INCREMENT* möglich.

Die Spezifikation verlangt nicht, dass der Provider Optimistic Locking tatsächlich ohne Datenbanksperren implementiert. Die bekannten Provider tun das aber. Für Details sei auf den Abschnitt 3.4 (Locking and Concurrency) der Spezifikation verwiesen.

Pessimistic Locking

In manchen Fällen ist Optimistic Locking nicht ausreichend, z. B. wenn schon bei Beginn eines Geschäftsprozesses – wenn die verwendeten Daten gelesen werden –, möglichst sicher sein soll, dass die veränderten Daten später auch wieder gespeichert werden können. Dann kommt man um Pessimistic Locking nicht herum. Bei diesem Verfahren werden die betroffenen Sätze in der Datenbank mit einer Blockierung belegt, die parallele Zugriffe verhindert.

Pessimistic Locking wird durch die Lock-Modi *PESSIMISTIC_...* angewählt. Dabei darf ein Exclusive Lock (*PESSIMISTIC_WRITE*) für einen Eintrag existieren oder alternativ

9 *javax.persistence.OptimisticLockException*

beliebig viele Shared Locks (*PESSIMISTIC_READ*). Das genaue Verfahren ist durch die Spezifikation nicht festgelegt. Insbesondere kann die Datenbank ggf. mehr Zeilen als nötig sperren.

Wird ein mit *PESSIMISTIC_READ* gesperrtes Objekt im Verlaufe des Geschäftsprozesses verändert und abgespeichert, wird der Shared Lock in diesem Moment zu einem Exclusive Lock verändert.

Der ein Lock anfordernde Prozess wartet i. d. R., bis der ein Eintrag gesperrt werden kann. Sollte dies aus einem schwerwiegenden Grund (bspw. ein Dead Lock) nicht möglich sein, wird der entsprechende Methodenaufruf abgebrochen und eine *PessimisticLockException*[10] ausgeworfen. Die aktuelle Transaktion wird zudem für ein Rollback markiert. Wird die Lock-Anforderung dagegen durch eine Zeitüberschreitung abgebrochen, so wird eine *LockTimeoutException*[11] ausgeworfen, ohne die Transaktion ungültig zu machen. Die entsprechende Wartezeit kann durch eine Konfigurationseinstellung der DB vorgegeben sein oder durch den unten beschriebenen Hint *javax.persistence.lock.timeout*.

Für Pessimistic Locking ist es nicht nötig, dass ein Versionsattribut existiert. Wenn das aber der Fall ist, wird es wie beim Optimistic Locking beschrieben behandelt. Dadurch ist gewährleistet, dass zwei Prozesse mit unterschiedlichen Locking-Verfahren auf den gleichen Daten arbeiten können.

Locking Hints

Das Verhalten von Pessimistic Locking kann mithilfe zweier Hints beeinflusst werden: Mit *javax.persistence.lock.timeout* kann eine maximale Wartezeit in Millisekunden angegeben werden. Ohne Angabe des Hints gilt eine evtl. für die DB konfigurierte Timeoutzeit als Vorgabe.

Mit dem Hint *javax.persistence.lock.scope* kann bestimmt werden, ob ein Pessimistic Lock mehr als nur das jeweils bearbeitete Objekt blockiert. Die folgenden Werte können angegeben werden:

- *PessimisticLockScope.NORMAL:* Nur die zum Objekt gehörenden Tabelleneinträge werden blockiert. Das ist die Voreinstellung.
- *PessimisticLockScope.EXTENDED:* Es werden zusätzlich die Einträge der Tabellen blockiert, die Element Collections darstellen. Zudem erstreckt sich die Blockierung auch auf die Einträge in Verknüpfungstabellen, die zu Relationen gehören, deren Eigentümer das bearbeitete Objekt ist. Die dadurch referenzierten Einträge werden allerdings nicht blockiert.

Die Hints können als Parameter den Methoden *EntityManager.find*, *EntityManager.lock*, *EntityManager.refresh* und *Query.setHint* übergeben oder bei der Deklaration einer Named

10 *javax.persistence.PessimisticLockException*
11 *javax.persistence.LockTimeoutException*

Query eingetragen werden. *javax.persistence.lock.timeout* kann zudem als Property im Deskriptor *persistence.xml* eingetragen werden. Es sei an dieser Stelle nochmals darauf hingewiesen, dass Hints nur Hinweise sind, die vom Provider nicht beachtet werden müssen. Für eine portable Anwendung sollten Sie auf die Hints also nicht angwiesen sein.

3.6.5 Callback-Methoden und Listener

Mithilfe von Callback-Methoden (oder auch Lifecycle-Methoden) können in die Operationen des Persistenzproviders bzgl. eines Entity-Objekts zusätzliche Aktionen eingeschleust werden. So kann man z. B. vor dem Speichern der Daten Validitätstests durchführen, um zu verhindern, dass ungültige Daten in der Tabelle abgelegt werden. Dazu können Methoden mit der Signatur *void methodName()* in die Entity-Klasse aufgenommen und mit einer der Annotationen aus Tabelle 3.1 markiert werden.

Callback-Annotation	Annotierte Methode wird aufgerufen …
@PrePersist	in *EntityManager.persist* vor der Ablage in der DB. Dies gilt auch bei *merge*, wenn das betroffene Objekt neu in die DB eingefügt wird.
@PostPersist	nach dem Einfügen in der DB. Dies kann direkt nach dem Aufruf von *persist* geschehen, ist aber normalerweise verzögert. Generierte IDs sind in der Callback-Methode bereits gesetzt.
@PostLoad	nach dem Laden eines Objekts aus der DB im Zuge eines Aufrufs von *find* oder der Ausführung einer Query.
@PreUpdate	vor dem Speichern von Änderungen in der DB. Das kann durch bspw. durch Aufruf von *flush* ausgelöst werden oder bis zum Ende der Transaktion verzögert geschehen.
@PostUpdate	analog zu *@PreUpdate*, allerdings nach der DB-Operation.
@PreRemove	in *EntityManager.remove* vor dem Löschen aus der DB.
@PostRemove	nach dem Löschen aus der DB. Das kann direkt nach dem Aufruf von *remove* geschehen, ist aber normalerweise verzögert.

Tabelle 3.1: Callback-Annotationen

Die Methoden können eine beliebige Sichtbarkeit (*private*, …, *public*) haben, dürfen aber nicht *static* oder *final* sein.

Die Deklaration von Exceptions ist nicht erlaubt, daher können die Methoden nur Unchecked Exceptions auswerfen. In den Pre-Methoden wird dadurch die entsprechende DB-Aktion nicht durchgeführt. Geschah der Aufruf innerhalb einer Transaktion, wird sie als „Rollback Only" markiert.

Ein Beispiel für die oben angesprochene Validierung der Daten zeigt Listing 3.95[12].

12 Dazu werden wir später allerdings eine elegantere Möglichkeit kennen lernen: Bean Validation.

```
@Entity
@Access(AccessType.FIELD)
public class Country
{
  …
  @PrePersist
  @PreUpdate
  private void validate()
  {
    if (this.name == null || this.name.isEmpty())
    {
      throw new IllegalArgumentException("name darf nicht leer sein");
    }
    …
  }
```

Listing 3.95: Validierung vor dem Speichern

Die Callback-Methoden können auch in eine separate Listener-Klasse ausgelagert werden. Die Signatur der Methoden ändert sich dann in *void methodName(Object)*, da in diesem Fall das gerade in der Verarbeitung befindliche Entity-Objekt an die Methode übergeben wird. Einer Entity-Klasse können mithilfe der Annotation *@EntityListeners* die gewünschten Listener-Klassen mitgegeben werden (Listing 3.96).

```
public class DebugListener
{
  @PrePersist
  public void prePersist(Object entity)
  {
    System.out.println("prePersist(" + entity + ")");
  }

  @PostPersist
  public void postPersist(Object entity)
  {
    System.out.println("postPersist(" + entity + ")");
  }
  …
}

@Entity
@Access(AccessType.FIELD)
@EntityListeners(DebugListener.class)
public class Country
{
  …
}
```

Listing 3.96: Entity Listener und seine Verknüpfung mit einer Entity-Klasse

Entity Listener können sogar als Default Listener eingetragen werden. Sie gelten dann für alle Entity-Klassen der Persistence Unit. Zur Deklaration der Default Listener dient der in Listing 3.97 gezeigte Abschnitt des Mapping-Deskriptors *META-INF/orm.xml*. Für weitere Details dazu und zur Aufrufreihenfolge im Fall von mehrfach deklarierten Callbacks sei auf die Spezifikation verwiesen: Abschnitt 3.5 (Entity Listeners and Callback Methods).

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                    http://java.sun.com/xml/ns/persistence/orm_2_0.xsd"
  version="2.0">
  <persistence-unit-metadata>
    <persistence-unit-defaults>
      <entity-listeners>
        <entity-listener class="de.….DebugListener"/>
      </entity-listeners>
    </persistence-unit-defaults>
  </persistence-unit-metadata>
  …
</entity-mappings>
```

Listing 3.97: Eintrag eines Default Entity Listeners im Mapping-Deskriptor

3.6.6 Bulk Update/Delete

Neben Suchabfragen können mit JPQL- und SQL-Queries auch Veränderungen der Daten in der Datenbank durchgeführt werden. Statt *select* wird dazu im JPQL- bzw. SQL-Kommando *delete* oder *update* verwendet. Im zweiten Fall wird in einer Set-Klausel nach der From-Klausel angegeben, welche Modifikationen an den von der Query betroffenen Daten durchgeführt werden. Die Ausführung der Operation geschieht mit der Methode *executeUpdate*. Sie gibt die Anzahl betroffener Datenbank-Einträge zurück (Listing 3.98).

```
Query query = em.createQuery("update Country c set c.area=120 where …");
int count = query.executeUpdate();

query = em.createQuery("delete Cocktail c where c.name like 'B%'");
count = q.executeUpdate();
```

Listing 3.98: Bulk Update und Bulk Delete

Zu beachten ist hier allerdings, dass die Operationen den Entity Manager zwar benutzen, ihn aber dennoch weitestgehend umgehen:

- Derzeit gemanagte Objekte werden nicht verändert. Dadurch kann der Zustand der Objekte im Entity Manager nach der Operation ggf. vom Zustand in der DB abweichen.

- Ein Optimistic Locking findet nicht statt. Evtl. vorhandene Versionswerte bleiben also unverändert.
- Eine Kaskadierung der Operation findet nicht statt.

Die Bulk-Operationen benötigen zu ihrer Ausführung eine aktive Transaktion.

3.7 Caching

Datenbankoperationen sind im Vergleich zu Aktionen innerhalb des Java-Prozesses schleichend langsam. Daher versucht man, einmal gelesene Daten für eine zweite Verwendung möglichst nicht erneut aus der Datenbank zu holen, sondern sie stattdessen im Hauptspeicher zu halten. Da die Daten für eine konsistente Verarbeitung aktuell sein müssen, funktioniert Caching besonders gut für Daten, die im Wesentlichen konstant sind – die klassischen Konfigurations- oder Stammdaten –, während es für sich häufig ändernde Daten nutzlos oder sogar kontraproduktiv sein kann.

Java Persistence definiert im Standard zwei Caches: 1st und 2nd Level Cache. Die gängigen Provider unterstützen darüber hinaus noch den sog. Query Cache.

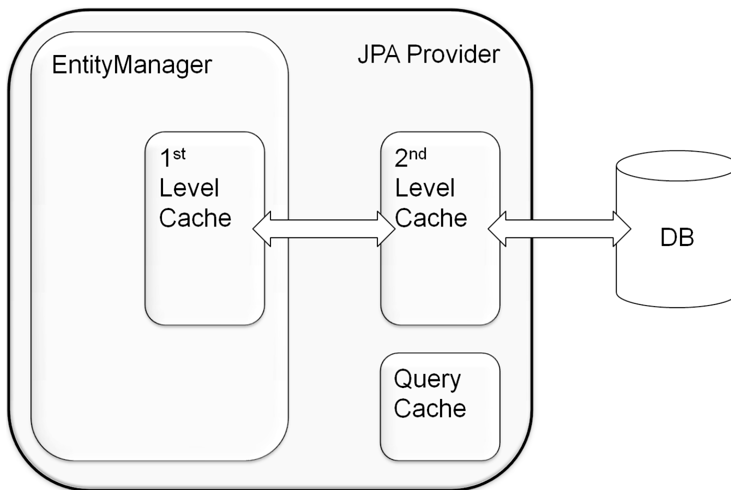


Abbildung 3.28: Caches

First Level Cache

Der Entity Manager hält alle von ihm gemanagten Objekte in einer internen *Map*-ähnlichen Datenstruktur, die mit der ID der darin befindlichen Objekte indiziert wird. Dieser 1st Level Cache ist stets vorhanden und lässt sich nicht deaktivieren. Er liegt sozusagen als Puffer zwischen der Java-Anwendung und der Datenbank. Die Entity-Manager-Me-

thoden wirken im Wesentlichen auf den Inhalt des Caches, die zugehörige Datenbankoperation ist damit nur lose gekoppelt. So führt *find* nur beim ersten Aufruf zum Laden des Objekts aus der Datenbank. Alle weiteren *find*-Aufrufe für die gleiche ID liefern das Objekt aus dem Cache. Umgekehrt fügt *persist* ein neues Objekt zunächst in den Cache ein, der DB-Eintrag passiert erst später.

Queries werden nicht aus dem 1st Level Cache bedient. Vielmehr führt eine Query immer zu einer Datenbankabfrage, wobei geänderte Objekte im Allgemeinen zuvor gespeichert werden.

Der 1st Level Cache ist Teil des Entity Managers. Wird er geschlossen, geht auch der Cache verloren. Löst man ein bislang gemanagtes Objekt aus dem Entity Manager – per *detach*, *clear* oder in Folge eines Transaktions-Rollbacks – wird der entsprechende Cache-Eintrag ebenfalls entfernt. Der 1st Level Cache ist also eher kurzfristig aktiv, dient somit im Wesentlichen zur Unterstützung von Geschäftsprozessen, die die betroffenen Daten in ihrem Verlauf einlesen, bearbeiten, die Veränderungen am Transaktionsende speichern und den Entity Manager anschließend schließen.

Second Level Cache

Sollen die verarbeiteten Daten über einen längeren Zeitraum im Cache gehalten werden, wird dazu ein Cache benötigt, der übergreifend über Geschäftsprozesse oder Transaktionen wirksam bleibt. Diese Aufgabe übernimmt der 2nd Level Cache. Er ist ähnlich dem 1st Level Cache eine Datenstruktur, deren Einträge mit ihrer ID adressiert werden, die aber nicht an einen Entity Manager oder eine Transaktion gebunden ist, sondern applikationsweit zur Verfügung steht.

Der Second Level Cache wird durch das Element *<shared-cache-mode>* im Deskriptor *persistence.xml* konfiguriert (Listing 3.99). Der Wert des Elements bestimmt, welche Entity-Klassen im 2nd Level Cache Berücksichtigung finden (Tabelle 3.2).

```
<persistence-unit name="…">
  <provider>…</provider>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  …
```

Listing 3.99: Konfiguration des 2nd Level Cache im Deskriptor „persistence.xml"

Wert von *shared-cache-mode*	2nd Level Cache aktiv für
ALL	alle Entity-Klassen
NONE	keine Klasse
ENABLE_SELECTIVE	Entities mit *@Cacheable(true)*
DISABLE_SELECTIVE	Entities ohne *@Cacheable(false)*

Tabelle 3.2: Verfügbare Cache-Modi

Es empfiehlt sich, Caching nicht „mit der Gießkanne“ zu betreiben, d. h. den 2nd Level Cache nicht unreflektiert für alle Entity-Klassen einzuschalten, da die Performanz der Anwendung nur bei sich nicht oder nicht häufig ändernden Daten vom Caching profitiert. In anderen Fällen kann es sogar nachteilig sein. In den meisten Fällen ist daher *ENABLE_SELECTIVE* der richtige Modus. Bei ihm nehmen nur die Entity-Klassen am 2nd-Level-Caching teil, die mit *@Cacheable(true)* annotiert sind (Listing 3.100).

```
@Entity
@Access(AccessType.FIELD)
@Cacheable(true)
public class Country
{
  …
```

Listing 3.100: Explizite Aktivierung des 2nd Level Cache für eine Klasse

In gleicher Weise, nur invertiert, arbeitet *DISABLE_SELECTIVE*. Wird *<shared-cache-mode>* nicht angegeben, gilt ein providerabhängiger Vorgabewert.

Je nach eingesetztem Provider sind zur Konfiguration des Caches noch weitere Parameter notwendig, die im *<properties>*-Anteil der Datei *persistence.xml* angegeben werden. Damit lassen sind bspw. Cache-Strategien, Verweilzeiten, Replikationsverfahren im Cluster usw. konfigurieren. In einigen Fällen stehen sogar mehrere Caching-Provider zur Auswahl. Für Details dazu sei auf die Dokumentation des jeweiligen Providers verwiesen. Das Beispielprojekt enthält eine Basiskonfiguration für Eclipselink mit dem standardmäßig eingebauten Cache-Provider sowie für Hibernate mit Infinispan als Cache-Provider.

Auf den 2nd Level Cache kann im Programm auch direkt zugegriffen werden. Dazu wird ein Objekt vom Typ *Cache* benötigt, das man sich z. B. von der Entity Manager Factory liefern lassen kann. Das Interface Cache enthält die Methode *contains*, mit der man abfragen kann, ob ein Objekt mit einer bestimmten ID im Cache vorhanden ist, sowie die Methoden *evict* und *evictAll*, mit denen Objekte aus dem Cache entfernt werden können (Listing 3.101).

```
EntityManager em = …
EntityManagerFactory emf = em.getEntityManagerFactory();
Cache secondLevelCache = emf.getCache();

// Ist Country "IT" im Cache?
boolean isCached = secondLevelCache.contains(Country.class, "IT"));

// Country "DE" aus dem Cache entfernen
secondLevelCache.evict(Country.class, "DE"));
```

Listing 3.101: Programmatischer Zugriff auf den 2nd Level Cache

entwickler.press

Die Nutzung des Cache-API wird sicher nur in speziellen Situationen oder zum Debugging benötigt, da der Cache eigentlich transparent für die Anwendung funktioniert. Aber Sie wissen ja bereits: „eigentlich“ ist ein Signalwort …

Query Cache

Queries führen jedes Mal zu einer Abfrage in der Datenbank, d. h. die zuvor beschriebenen Caches kommen dazu nicht zum Einsatz. Es gibt aber die Möglichkeit, auch dafür einen Cache einzurichten, der zu einer Query und den darin genutzten Parametern die Ergebnismenge speichert. Der Query Cache ist providerabhängig verfügbar und in aller Regel mit den 2nd Level Cache verbunden. Er speichert die Ergebnismenge daher meist nur in Form der IDs; die zugehörigen Objekte befinden sich dann im 2nd Level Cache.

Der Query Cache ist in der Spezifikation nur rudimentär erwähnt. Seine Konfiguration ist vom eingesetzten Provider abhängig. Details finden sich in der Dokumentation der Provider. Zudem muss die Nutzung des Caches für eine Query explizit aktiviert werden, und zwar mithilfe eines providerabhängigen Hints (Listing 3.102).

```
Continent continent = …;
TypedQuery<Kunde> query
  = em.createQuery("select c from Country c where c.continent=:cont",
                 Country.class);
query.setParameter("cont", continent);
query.setHint("eclipselink.cache-usage", "CheckCacheThenDatabase");
query.setHint("org.hibernate.cacheable", true);
…
```

Listing 3.102: Aktivierung des Query Cache für Eclipselink oder Hibernate

Einträge im Query Cache werden verworfen, wenn Änderungen an den beteiligten Entities vorgenommen werden. Das Verfahren ist hier aber wiederum vom Provider abhängig.

3.8 Erweiterte Entity Manager

Bislang wurden in diesem Kapitel transaktionsgebundene Entity Manager verwendet, die sich auf einfache Weise bereitstellen lassen, z. B. per Injektion in eine CDI Bean (Abschnitt 3.2.3, Listing 3.7). Damit lassen sich Abläufe modellieren, die dem folgenden Schema folgen:

a) Beginn der Transaktion

b) Lesen der benötigten Daten

c) Erzeugen, Modifizieren und Löschen von Daten

d) Transaktion abschließen (und damit Änderungen speichern)

Ein gesamter Geschäftsprozess setzt sich in der Regel aus mehreren Abläufen dieser Art zusammen, die z. B. durch Userinteraktionen voneinander getrennt sind.

Nach dem Abschluss einer Transaktion werden die verarbeiteten Objekte detached, müssen also mittels *EntityManager.merge* wieder attached warden, wenn sie in einem Folgeablauf wieder benötigt werden. Damit sind im Allgemeinen erneute Zugriffe auf die DB verbunden.

Zudem können im Detached-Zustand keine Lazy-Attribute nachgelesen werden. Soll z. B. eine Webanwendung Daten zur Anzeige bringen, die auf die beschriebene Art eingelesen wurden, so müssen vor Ende der Transaktion alle notwendigen Attribute bereits gelesen worden sein, damit die Anwendung nicht mit einer Lazy Load Exception abbricht.

Eine alternative Verarbeitungsmöglichkeit eröffnet sich mit einem erweiterten Entity Manager. Er ist nicht an die Transaktion gebunden, wodurch die Entity-Objekte über Transaktionsgrenzen hinweg gemanagt bleiben können.

Die EJB-Spezifikation sieht eine Möglichkeit zur Erzeugung eines Extended Entity Manager vor, und zwar wiederum durch Injektion mit der Annotation *@PersistenceContext*, nun aber mit einem Zusatzparameter zur Auswahl des erweiterten Typs.

```
@Stateful
public class SomeBusinessProcessBean
{
  @PersistenceContext(name = "ee_demos",
                      type=PersistenceContextType.EXTENDED)
  private EntityManager entityManager;
  …
```

Listing 3.103: Injektion eines Extended Entity Managers in eine Stateful EJB

Ein Extended Entity Manager ist an die Session gebunden, d. h. er „lebt" mit der jeweiligen Instanz der Stateful EJB, in die er injiziert wurde. Die Methoden *persist*, *merge* und *remove*, deren Aufruf bisher nur in einer aktiven Transaktion erlaubt war, dürfen nun auch ohne Transaktion verwendet werden. Wird später eine Transaktion begonnen und mit *Commit* beendet, werden die Änderungen abgespeichert. Damit wird es möglich, in mehreren Teilabläufen eines Geschäftsprozesses Daten zu lesen und zu modifizieren, ohne dabei Transaktionen zu nutzen. Die Daten bleiben über die ganze Zeit im Entity Manager gemanagt, d. h. unnötige DB-Zugriffe unterbleiben und Lazy-Attribute können jederzeit nachgelesen werden. Sollen die Daten schließlich gespeichert werden, reicht der Aufruf einer – ggf. sogar leeren – transaktionalen Methode (Listing 3.104).

```
@Stateful
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public class SomeBusinessProcessBean
{
  @PersistenceContext(name = "ee_demos",
                      type=PersistenceContextType.EXTENDED)
```

```
  private EntityManager entityManager;

  public void doStep1()
  {
    // Geschäftsprozess, Teil 1: Daten lesen, ggf. verändern
  }

  public void doStep2()
  {
    // Geschäftsprozess, Teil 2: Weiter lesen und verändern
  }

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void save()
  {
  }
}
```

Listing 3.104: Realisierung eines Geschäftsprozesses durch Methoden einer Stateful EJB

Dieses Pattern funktioniert recht gut, wenn die steuernde Instanz der EJB am Ende des Geschäftsprozesses zerstört wird, womit dann auch der Entity Manager abgebaut wird. Somit verbleiben keine „Leichen" im Speicher, das Big-Session-Problem wird also vermieden. Das Pattern ist allerdings darauf beschränkt, dass eine Stateful Session EJB die Teile des Geschäftsprozesses verkörpert.

Will man auf EJBs verzichten – zumindest für die Bereitstellung eines Entity Managers – kann man einen Application Managed Entity Manager verwenden. Hierbei wird ein Objekt vom Typ *EntityManager* in eigener Regie erzeugt und kontrolliert. Die Lebensdauer dieses Entity Managers sollte dem modellierten Geschäftsprozess entsprechen, um wiederum dem Big-Session-Problem aus dem Weg zu gehen. Es bietet sich an, den CDI Scope *Conversation* zu nutzen und eine Konversation genau über die Dauer des Geschäftsprozesses aktiv zu halten. Zur Bereitstellung des Entity Managers lässt sich sehr gut ein CDI Producer verwenden. Er liefert auf Anforderung eine neue Instanz vom Typ *EntityManager* und sieht auch eine passende Disposer-Methode zum Schließen des Entity Managers vor (Listing 3.105).

```
@ApplicationScoped
public class EntityManagerProducer implements Serializable
{
  @PersistenceUnit(unitName = "default")
  private EntityManagerFactory entityManagerFactory;

  @Produces @ConversationScoped
  public EntityManager createEntityManager()
  {
    return this.entityManagerFactory.createEntityManager();
  }
```

```
    public void disposeEntityManager(@Disposes EntityManager entityManager)
    {
      if (entityManager.isOpen())
      {
        entityManager.close();
      }
    }
  }
```

Listing 3.105: CDI-Producer für einen Application Managed Entity Manager

Die *EntityManager*-Instanz wird im gezeigten Producer hergestellt, indem auf ein Objekt des Typs *EntityManagerFactory* zurückgegriffen wird, das quasi die Persistence Unit darstellt. Im nächsten Abschnitt wird die Entity Manager Factory nochmals etwas näher beleuchtet. Im Kontext eines Applikationsservers kann sie mithilfe der Annotation *@PersistenceUnit* injiziert werden, wobei der Namensparameter entfallen kann, wenn *persistence.xml* nur eine Persistence Unit definiert.

Die Producer-Methode *createEntityManager* ist mit *@ConversationScoped* annotiert, der gelieferte Entity Manager wird also im *Conversation*-Kontext verwaltet. Am Ende der Konversation sorgt die Dispose-Methode *disposeEntityManager* dafür, dass der Entity Manager geschlossen wird.

Nach diesen Vorbereitungen kann eine *EntityManager*-Instanz an den gewünschten Stellen der Geschäftslogik injiziert werden. Diese ist an die Konversation gebunden, bleibt also beginnend mit dem Request, in dem *Conversation.begin* aufgerufen wird, aktiv, bis in einem späteren Request *Conversation.end* aufgerufen wird. Auch hier können Datenänderungen außerhalb von Transaktionen gesammelt werden, um dann in einer transaktionalen Methode gespeichert zu werden. Anders als bei einem Extended Entity Manager gibt es hier allerdings keine automatische Zuordnung zur aktiven Transaktion; diese muss explizit mit der Methode *joinTransaction* hergestellt werden. Es empfiehlt sich, dann ebenfalls *flush* aufzurufen. Das sollte zwar lt. Spezifikation am Transaktionsende ohnehin geschehen, unterbleibt aber erfahrungsgemäß bei einigen Implementierungen.

Die Anwendungsaspekte „Geschäftsprozess" und „Persistenzschicht" lassen sich in der Praxis gut trennen, um Wiederverwendbarkeit und Übersichtlichkeit zu erhöhen. Dem kommt entgegen, dass CDI Scopes sich problemlos mischen lassen. Man kann also wie im Beispiel einen Conversation Scoped Entity Manager in eine Request Scoped Bean injizieren, die wiederum von einer Conversation Scoped Bean verwendet wird (Listing 3.106).

```
@ConversationScoped
public class XyzController implements Serializable
{
  @Inject
  XyzRepository   xyzRepository;

  @Inject
  Conversation    conversation;
```

```
  public void doFirstStep()
  {
    // Geschäftsprozess, Teil 1: Daten lesen, ggf. verändern
    conversation.begin();
    … = xyzRepository.findById(…);
    …
  }

  public void doSecondStep()
  {
    // Geschäftsprozess, Teil 2: Weiter lesen und verändern
    xyzRepository.insert(…);
    …
  }

  public void doLastStep()
  {
    // Geschäftsprozess, letzter Teil: Speichern
    xyzRepository.saveAll();
    conversation.end();
  }
}

@RequestScoped
public class XyzRepository implements Serializable
{
  @Inject
  private EntityManager entityManager;

  @TransactionRequired
  public void saveAll()
  {
    entityManager.joinTransaction();
    entityManager.flush;
  }

  public Xyz findById(int id)
  {
    return entityManager.find(Xyz.class, id);
  }

  public void insert(Xyz xyz)
  {
    entityManager.persist(xyz);
  }

  // weitere, nicht-transaktionale Methoden
  …
}
```

Listing 3.106: Nutzung eines konversationsbezogenen Entity Managers

Es gibt mit der dargestellten Vorgehensweise, einen Entity Manager im Conversation Scope zu nutzen, eine Schwierigkeit, die Ihnen nicht verheimlicht werden soll: Objekte im Conversation Scope müssen serialisierbar sein. Da die JPA-Spezifikation darüber keinerlei Aussage macht, ist es providerabhängig, ob der konkrete Entity Manager tatsächlich *Serializable* implementiert. Bei Hibernate ist es beispielsweise der Fall, während Eclipselink einen nichtserialisierbaren Entity Manager hat. Damit bleibt derzeit nur die Möglichkeit, bei Bedarf ein serialisierbares Proxy als Hülle um den Entity Manager zu konstruieren (Listing 3.107).

```
@ApplicationScoped
public class EntityManagerProducer implements Serializable
{
  …
  @Produces @ConversationScoped
  public EntityManager createEntityManager()
  {
    EntityManager entityManager
      = this.entityManagerFactory.createEntityManager();
    if (!(entityManager instanceof Serializable))
    {
      ClassLoader classLoader = EntityManager.class.getClassLoader();
      Class<?>[] interfaces = new Class[] { EntityManager.class,
                                     Serializable.class };
      EntityManagerInvocationHandler handler
        = new EntityManagerInvocationHandler(entityManager);
      entityManager
        = (EntityManager) Proxy.newProxyInstance(classLoader,
                                          interfaces, handler);
    }
    return entityManager;
  }
…
  private static class EntityManagerInvocationHandler
    implements InvocationHandler
  {
    private transient EntityManager entityManager;

    public EntityManagerInvocationHandler(EntityManager entityManager)
    {
      this.entityManager = entityManager;
    }

    public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable
    {
      if (this.entityManager == null)
      {
        throw new IllegalStateException("Session discarded");
      }
```

```
            return method.invoke(this.entityManager, args);
        }
    }
}
```

Listing 3.107: Workaround für nichtserialisierbare Entity Manager

Dieser Workaround ist allerdings nicht ganz wasserdicht. Da man selbst in Java nicht wirklich zaubern kann, kann aus einem nichtserialisierbaren Entity Manager nicht ein serialisierbares Objekt gemacht werden. Stattdessen hält das Proxy den *EntityManager* als transientes Objekt, das demzufolge bei einer Serialisierung/Deserialisierung des Proxy-Objekts verloren gehen würde. Wir vertrauen also darauf, dass das nicht passieren wird. Wenn doch, wird von der betroffenen Methode eine sinnvolle Exception ausgeworfen.

Man kann sich des Themas aber auch durch Einsatz einer CDI Extension entledigen: JBoss Seam Persistence bietet die Möglichkeit der Injektion einen Conversation Scoped Entity Managers. Eine kurze Beschreibung dieser und weiterer Portable Extensions finden Sie am Ende des Kapitels über CDI.

Das übergreifende Projekt im letzten Buchkapitel verwendet einen Conversation Scoped Entity Manager.

3.9 Java Persistence in SE-Anwendungen

Wir haben uns in diesem Kapitel bislang mit Java Persistence im Enterprise-Kontext beschäftigt, d. h. mit der Nutzung innerhalb einer auf einem Applikationsserver laufenden Java-EE-Anwendung. Man kann JPA aber auch in SE-Anwendungen verwenden, sei es für Unit Tests oder Standalone-Anwendungen. Dabei ändern sich gegenüber einer EE-Umgebung nur die drei im Folgenden besprochenen Punkte.

3.9.1 Konfiguration der Persistence Unit im SE-Umfeld

Auch außerhalb eines Applikationsservers wird die Deskriptordatei *META-INF/persistence.xml* benötigt, allerdings mit etwas verändertem Inhalt: Das Element *<jta-data-source>* entfällt. Stattdessen werden mit den Properties *javax.persistence.jdbc.driver*, *javax.persistence.jdbc.url*, *javax.persistence.jdbc.user* und *javax.persistence.jdbc.password* die von JDBC bekannten Parameter zum Aufbau einer DB-Verbindung angegeben.

Eine Unschärfe in der JPA-Spezifikation führt zu einer weiteren Änderung des Deskriptors: Im Abschnitt 8.2.1.6.1 heißt es zwar „All classes contained in the root of the persistence unit are searched for annotated managed persistence classes …", aber später in 8.2.1.6.4 findet sich der Satz „The class element is used to list a managed persistence class. A list of all named managed persistence classes must be specified in Java SE environments to insure portability." Im allgemeinen Fall müssen also die von der Anwendung

verwendeten persistenten Klassen im Deskriptor einzeln aufgeführt werden, wenn auch Eclipselink und Hibernate eigentlich ohne diese Information auskommen.

Das Beispielprojekt nutzt JPA in der Standalone-Variante für Tests. Einen Ausschnitt der dazu im Test-Classpath liegenden *META-INF/persistence.xml* zeigt Listing 3.108. Darin sind mehrere Persistence Units spezifiziert, um unterschiedliche Provider zum Test verwenden zu können.

```
<persistence xmlns="http://java.sun.com/xml/ns/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">
  <persistence-unit name="eclipselink">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>

    <class>de.gedoplan.buch.eedemos.entity.Car</class>
    <class>de.gedoplan.buch.eedemos.entity.City</class>
    …
    <class>de.gedoplan.buch.eedemos.entity.Vehicle</class>

    <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>

    <properties>
      <property name="javax.persistence.jdbc.driver"
               value="com.mysql.jdbc.Driver" />
      <property name="javax.persistence.jdbc.url"
               value="jdbc:mysql://localhost:3306/ee_demos" />
      <property name="javax.persistence.jdbc.user"
               value="ee_demos" />
      <property name="javax.persistence.jdbc.password"
               value="ee_demos" />
  …
  </persistence-unit>

  <persistence-unit name="hibernate">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
  …
  </persistence-unit>
```

Listing 3.108: Deskriptor „META-INF/persistence.xml“ für eine SE-Anwendung

3.9.2 Erzeugung eines Entity Managers in SE-Anwendungen

Außerhalb von Applikationsservern stehen uns die Enterprise-Ressourcen nicht zur Injektion zur Verfügung. An die Stelle der bisher genutzten Annotationen *@PersistenceContext* bzw. *@PersistenceUnit* tritt nun eine Anweisungssequenz, die ausgehend von der Klasse

Persistence zunächst eine Entity Manager Factory erstellt, die in der Folge zur Erzeugung von *EntityManager*-Objekten genutzt wird.

Ein Objekt des Typs *EntityManagerFactory* repräsentiert eine der in *persistence.xml* definierten Persistence Units. Bei seinem Aufbau werden die im Deskriptor angegebenen Parameter eingelesen und interpretiert. Die Erzeugung einer Entity Manager Factory ist daher eine schwergewichtige Operation, die man im Programmverlauf aber auch nur einmalig benötigt. Die Methode *Persistence.createEntityManagerFactory* erhält als Parameter den Namen der Persistence Unit. Optional können als zweiter Parameter Properties mitgegeben werden, die die in *persistence.xml* eingetragenen ergänzen oder ersetzen.

Die Methode *EntityManagerFactory.createEntityManager* liefert einen neuen Entity Manager auf Basis der zuvor aufgebauten Factory. Diese Operation ist leichtgewichtiger, kann also ohne Bedenken häufiger ausgeführt werden.

EntityManagerFactory ist threadsave, *EntityManager* nicht. Will man also mit mehreren Threads im Programm arbeiten, wird nur eine *EntityManagerFactory* benötigt, aber für jeden Thread jeweils ein *EntityManager*.

Die Entity-Manager-Erzeugung lässt sich zum bequemen Aufruf leicht in eine Helferklasse auslagern (Listing 3.109).

```
public class JpaUtil
{
  private static EntityManagerFactory entityManagerFactory;

  public static synchronized
    EntityManagerFactory getEntityManagerFactory()
  {
    if (entityManagerFactory == null)
    {
      entityManagerFactory
        = Persistence.createEntityManagerFactory("eclipselink");
    }

    return entityManagerFactory;
  }

  public static EntityManager createEntityManager()
  {
    return getEntityManagerFactory().createEntityManager();
  }
}
```

Listing 3.109: Erzeugung von „EntityManagerFactory“ und „EntityManager“

3.9.3 Transaktionssteuerung in Java-SE-Anwendungen

Während der JPA-Provider in einer Enterprise-Umgebung die gerade aktive Transaktion des Applikationsservers übernimmt, muss zur Transaktionssteuerung in einem Java-SE-Programm selbst Hand angelegt werden. Dazu bietet *EntityManager* die Methode *getTransaction* an, die das im Entity Manager eingebaute Transaktionsobjekt vom Typ *EntityTransaction* liefert. Darauf können dann die klassischen Transaktionssteuerungsmethoden *begin*, *commit*, *rollback* etc. angewendet werden (Listing 3.110).

```
EntityManager em = JpaUtil.createEntityManager();
EntityTransaction tx = em.getTransaction();
tx.begin();
City city = new City("Berlin", 3500000, 892);
em.persist(city);
tx.commit();
em.close();
```

Listing 3.110: Transaktionssteuerung in SE-Anwendungen

4 Bean Validation

4.1 Aufgabenstellung

Die Aufgabenstellung ist schnell umrissen: Anwendungen sollen i. A. mit validen Daten arbeiten, d. h. Eingabedaten, aber auch Verarbeitungsergebnisse müssen gewisse Regeln erfüllen, um konsistent zu sein. Die entsprechenden Bedingungen können sich auf einzelne Feldinhalte beziehen („Lebensalter>0") oder mehrere Werte in einen Zusammenhang stellen („Wenn die Postleitzahl ausgefüllt ist, dann muss auch der Ort angegeben sein – und umgekehrt"). Neben einfachen Prüfungen auf Wertebereiche oder Muster werden häufig auch Tests benötigt, die individuell ausprogrammiert werden.

Die Validierung der Daten wird an unterschiedlichen Stellen der Anwendung benötigt, zum Beispiel bei der Erfassung der Daten in einer JSF-basierten Webanwendung, bei der Verarbeitung der Daten in der CDI-basierten Geschäftslogik und bei der Speicherung der Daten mittels JPA. Die genannten Frameworks bieten teilweise einen eigenen Support zur Prüfung von Daten an. So kann man z. B. in JavaServer Faces Eingabefelder entsprechend parametrisieren und vordefinierte oder selbst entwickelte Validatoren verwenden (Listing 4.1). Details zur Validierung in JSF werden im entsprechenden Buchkapitel später noch dargestellt.

```
<h:inputText ... required="true">
  <f:validateLength minimum="1" maximum="50"/>
</h:inputText>
```

Listing 4.1: Beispielhafte Validierung in JavaServer Faces

Ein weiteres Beispiel: Java Persistence erlaubt die Angabe von Constraints durch Parameter auf Feldebene oder durch individuelle Programmierung einer Lifecycle-Methode (Listing 4.2).

```
@Basic(optional = false)
private String name;
…
@PrePersist @PreUpdate
protected void checkValid()
{
  if (this.name != null)
  {
    if (this.name.length() < 1 || this.name.length() > 50)
    {
```

```
      throw new IllegalArgumentException();
    }
  }
}
```

Listing 4.2: Beispielhafte Validierung in Java Persistence

Diese Art dezentraler Validierung ist einerseits aufwändig – wie bereits der gezeigte triviale Code zeigt – und birgt die große Gefahr, dass die einzelnen Tests eine nicht übereinstimmende Semantik haben: Man stelle sich im Beispiel vor, dass die Beschränkung der Feldlänge im Zuge einer Programmerweiterung von 50 auf 100 heraufgesetzt wird. Es gibt kein Werkzeug, das sicherstellt, dass alle testenden Programmteile gleichbedeutend geändert werden.

4.2 Plattformen und benötigte Bibliotheken

Das im Folgenden Beschriebene basiert auf der Spezifikation Bean Validation 1.0[1], die Ergebnis der JSR 303 ist. Die Version 1.1 wird im JSR 349 entwickelt.

Das zur Nutzung von Bean Validation benötigte Validation API kann mit der in Listing 4.3 gezeigten Maven Dependency bereitgestellt werden.

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>1.0.0.GA</version>
  <scope>provided</scope>
</dependency>
```

Listing 4.3: Maven Dependency für das Bean Validation API

Der Scope *provided* ist für Enterprise-Anwendungen ausreichend, da die Bibliothek in Java-EE-6-Applikationsservern vorhanden ist.

Die Referenzimplementierung Hibernate Validator ist z. B. in Glassfish 3 und JBoss 7 enthalten. Andere Implementierungen sind u. a. Apache Bean Validation und Spring Bean Validation.

1 JSR 303: Bean Validation,
Bean Validation Expert Group, 2009-10-12,
http://jcp.org à Search JSR 303 à Final Release Download à bean_validation-1_0-final-spec.pdf

4.3 Validation Constraints

Die redundante Implementierung von Validierungsregeln lässt sich vermeiden, wenn man den entsprechenden Programmcode aus den verschiedenen Schichten der Anwendung herauszieht und in den Domänendaten verankert, den Geschäftsobjekten also die Gültigkeitsregeln für die eigenen Daten mitgibt. Das geschieht mithilfe von Annotationen, die den zu validierenden Objekten, d. h. ihren Klassen und Attributen, hinzugefügt werden können (Listing 4.4).

```
public class Fragebogen
{
  @NotNull
  @Size(min = 1, max = 50)
  private String name;
  …
```

Listing 4.4: Deklaration von Bean Validation Constraints für eine Instanzvariable[2]

Auf diese Art können Datenfelder mit Konsistenzregeln versehen werden. Alternativ können die Annotationen auch an Getter-Methoden platziert werden. Für diesen sog. Property Access wird vorausgesetzt, dass die Methode die JavaBeans-Konventionen erfüllt, d. h. die Signatur *Type getProperty()* bzw. *boolean isProperty()* hat.

Eine Mischung beider Zugriffsarten ist erlaubt. Für die Sichtbarkeit der annotierten Felder oder Methoden macht die Bean-Validation-Spezifikation keine Einschränkung.

Wie üblich kann man anstelle der Annotationen auch XML verwenden. Der Deskriptor *META-INF/validation.xml* ist in der Spezifikation im Abschnitt 7, XML Deployment Descriptor, beschrieben.

JSF und Java Persistence nutzen Bean Validation im Wesentlichen ohne weitere Konfiguration ganz automatisch. Zudem kann die Validierung über ein einfaches API auch programmatisch erfolgen. Es ergibt sich damit eine Struktur mit zentraler, redundanzfreier Definition der Validierungsregeln, die von allen Teilen der Anwendung konsistent genutzt werden können (Abb. 4.1).

2 Den in diesem Kapitel gezeigten Beispielcode finden Sie im Begleitprojekt *ee-demos-bv*

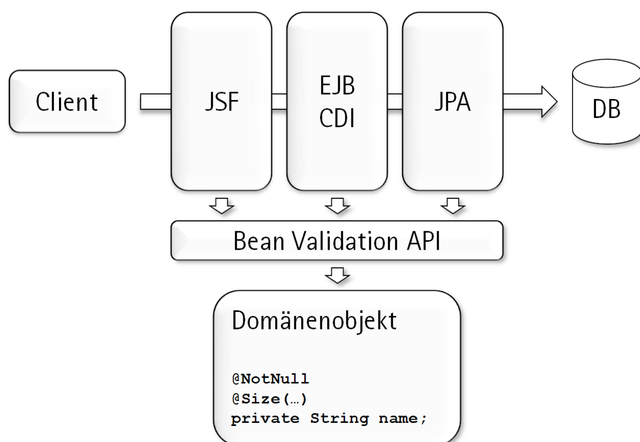


Abbildung 4.1: Zentralisierte Validierung von Geschäftsdaten

Vordefinierte Constraints

Im Bean-Validation-Standard sind 13 Annotationen für Constraints auf Attributebene vordefiniert (Tabelle 4.1).

Annotation[3]	Parameter	Akzeptierte Datentypen	Bedeutung
@Null *@NotNull*		alle	Wert muss *null* bzw. nicht-*null* sein
@AssertTrue *@AssertFalse*		*Boolean*	Wert muss *true* bzw. *false* sein
@Min *@Max*	*long value*	*Number*	Wert muss mindestens bzw. höchstens *value* sein
@DecimalMin *@DecimalMax*	*String value*	*Number* *String*	Wert muss mindestens bzw. höchstens *value* (*BigDecimal*-Wert als *String*) sein
@Size	*int min* *int max*	*String* *Collection* *Map*	Textlänge bzw. Anzahl Einträge muss im angegebenen Bereich liegen
@Digits	*int integer* *int fraction*	*Number* *String*	Wert muss numerisch mit der angegebenen maximalen Anzahl Ziffern sein
@Past *@Future*		*Date* *Calendar*	Wert muss in der Vergangenheit bzw. Zukunft liegen
@Pattern	*String regexp* *Flag[] flags*	*String*	Text muss dem regulären Ausdruck entsprechen (analog zu *java.util.regexp.Pattern*)

Tabelle 4.1: Vordefinierte Validation Constraints

3 Alle im Paket *javax.validation.constraints*

Beim Blick in die Definition der Standard-Constraints fällt auf, dass alle mit Ausnahme von *@NotNull* einen *null*-Wert als gültig erachten. Diese Ortogonalität der Constraints macht eine Kombination möglich. Im Beispiel von Listing 4.4 muss der Wert ein *String* der Länge *1* bis *50* sein. Er darf insbesondere nicht *null* sein. Ohne *@NotNull* wäre der Wert optional, müsste aber – wenn gefüllt – die restlichen Regeln (hier *@Size*) erfüllen.

Transitive Gültigkeit

Die Validierung berücksichtigt zunächst nur das ihr übergebene Objekt. Sind darin Attribute enthalten, deren Klassen ebenfalls Validierungsregeln enthalten, so kann die Gültigkeitsprüfung durch Angabe der Annotation *@Valid*[4] transitiv gemacht werden, d. h. bei Prüfung eines Objekts werden auch die so markierten Attribute einer Prüfung unterzogen (Listing 4.5).

```
public class Fragebogen
{
  …
  @Valid
  private Adresse adresse;
  …
```

Listing 4.5: Transitive Validierung

Constraint Composition

Die vordefinierten Constraints decken schon viele Fälle ab. Dennoch wird es in der Praxis schnell den Bedarf geben, eigene Validierungsregeln zu schreiben. So ist eine Regel für gültige Telefonnummern mittels *@Pattern(regexp = "\\+\\d{1,3}\\s\\d+\\s\\d+(-\\d+)?")* ausgedrückt nicht gerade gut lesbar und verständlich. Zudem entspräche eine mehrfache Verwendung dieser Annotation nicht dem Ziel der Wiederverwendbarkeit.

Eine Anwendung selbst entwickelter Constraints ist die, ein oder mehrere bestehende Constraints zu einer neuen Einheit zu kombinieren. Dazu wird eine Annotation entwickelt, die selbst mit *@Constraint*[5] und den gewünschten Constraints annotiert ist und eine Retention Policy von zumindest *RUNTIME* hat. Als Target eignen sich *FIELD* und *METHOD* für attributbezogene Prüfungen (Listing 4.6).

```
@Constraint(validatedBy = {})
@NotNull
@Size(min = 1)
@Target({ FIELD, METHOD })
@Retention(RUNTIME)
public @interface NotEmpty
{
```

4 *javax.validation.Valid*
5 *javax.validation.Constraint*

```
  String message() default "";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

Listing 4.6: Komposition bestehender Constraints zu einem neuen Constraint

Diese Annotation kann nun anstelle von *@NotNull @Size(min=1)* verwendet werden, um sicherzustellen, dass das betroffene Attribut mindestens ein Zeichen enthält (Listing 4.7).

```
public class Fragebogen
{
  @NotEmpty
  private String name;
  …
```

Listing 4.7: Nutzung eines selbstdefinierten Constraints (gleichwertig zu Listing 4.4)

Der obligatorische Parameter *message* der Constraints definiert jeweils die Fehlermeldung, die bei einer Regelverletzung durch die spätere Prüfung erzeugt wird. Bei der Definition eigener Constraints kann gewählt werden, ob die Meldungen der enthaltenen Teil-Constraints verwendet werden sollen oder eine einzelne Meldung für das neue Constraint erzeugt werden soll. In diesem Fall wird das neue Constraint zusätzlich mit *@ReportAsSingleViolation*[6] annotiert und ein geeigneter Text als Vorgabewert der Meldung angegeben (Listing 4.8).

```
@Constraint(validatedBy = {})
@NotNull
@Size(min = 1)
@ReportAsSingleViolation
@Target({ FIELD, METHOD })
@Retention(RUNTIME)
public @interface NotEmpty
{
  String message() default "darf nicht leer sein";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}
```

Listing 4.8: ÛDefinition einer Fehlermeldung für ein zusammengesetztes Constraint

Ein weiterer verpflichtender Parameter ist *groups* zur Zuordnung der weiter unten beschriebenen Validierungsgruppen. Schließlich muss auch *payload* vorgesehen werden. Mit diesem selten genutzten Parameter können Metadaten zwischen Constraint und nutzender Klasse ausgetauscht werden.

6 *javax.validation.ReportAsSingleViolation*

Durch weitere, optionale Parameter ist es zudem möglich, Werte an die enthaltenen Teil-Constraints weiterzuleiten. Details dazu finden sich in der Spezifikation (Abschnitt 2.3. Constraint Composition).

Constraint Programming

Eine andere Anwendung selbst entwickelter Constraints ist die programmatische Gültigkeitsprüfung. Dazu kann dem Parameter *validatedBy* der Annotation *@Constraint* ein Array von Klassen mitgegeben werden, die das Interface *ConstraintValidator*[7] implementieren. Darin ist u. a. die Methode *isValid* enthalten, die den Wert des betreffenden Attributs überprüft. Diese Art selbstentwickelter Constraints bietet sich an, wenn attributübergreifende Validierungen gefordert sind. Nehmen wir z. B. an, dass Objekte des Typs *Addresse* nur dann gültig sind, wenn entweder alle Attribute darin Werte enthalten oder alle *null* sind. Dazu können die in Listing 4.9 gezeigte Annotation und Validatorklasse genutzt werden.

```
// Constraint
@Constraint(validatedBy = AdresseValidator.class)
@Target({ FIELD, METHOD, TYPE })
@Retention(RUNTIME)
public @interface ValidAdresse
{
  String message() default "muss komplett gefüllt oder leer sein";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}



// Zugehörige Validatorklasse
public class AdresseValidator
  implements ConstraintValidator<ValidAdresse, Adresse>
{
  public void initialize(ValidAdresse constraint)
  {
  }

  public boolean isValid(Adresse adresse,
                         ConstraintValidatorContext validationContext)
  {
    // null ist ok
    if (adresse == null)
      return true;

    // alles null ist ok
    if (adresse.getOrt() == null && adresse.getPlz() == null
                                 && adresse.getStrasse() == null)
```

7 *javax.validation.ConstraintValidator*

```
      return true;

    // alles gefüllt ist ok
    if (adresse.getOrt() != null && adresse.getPlz() != null
                              && adresse.getStrasse() != null)
      return true;

    return false;
  }
}

// Beispielhafte Nutzung für eine Klasse
@ValidAdresse
public class Adresse
{
  …
```

Listing 4.9: Selbstdefiniertes Constraint mit Validatorklasse

Eine solche attributübergreifende Prüfung legt nahe, die Annotation – wie am Ende von Listing 4.9 gezeigt – für die betroffene Klasse zu nutzen. Dementsprechend ist *TYPE* im Target der Annotation aufgenommen worden. Die Annotation könnte aber auch für einzelne Attribute des passenden Typs genutzt werden.

Der Parameter *validationContext* der Methode *isValid* kann genutzt werden, um zusätzliche oder andere Fehlermeldungen zu erzeugen, als es der *message*-Parameter der Annotation vorgibt. Damit können insbesondere bei mehrteiligen Prüfungen aussagekräftige Meldungen für Validierungsfehler erzeugt werden. Beispiele dazu finden sich in der Dokumentation der Klasse *ConstraintValidatorContext*[8].

Der Parameter *validatedBy* der Annotation *@Constraint* darf mehrere Klassen enthalten. Dadurch ist es möglich, für verschiedene Zieltypen des Constraints jeweils eine Validatorklasse anzugeben. Listing 4.10 zeigt als Beispiel ein Constraint zur Prüfung von Ziffernfolgen nach dem Prüfziffernverfahren von Hans Peter Luhn, das für *String*- und *Integer*-Werte genutzt werden kann.

```
@Constraint(validatedBy = { LuhnStringValidator.class,
                            LuhnIntValidator.class })
@Target({ FIELD, METHOD, TYPE })
@Retention(RUNTIME)
public @interface Luhn
{
  …
}

public class LuhnStringValidator
  implements ConstraintValidator<Luhn, String>
```

8 *javax.validation.ConstraintValidatorContext*

```
{
  …
}

public class LuhnIntValidator
  implements ConstraintValidator<Luhn, Integer>
{
  …
}
```

Listing 4.10: Constraint mit mehreren Validatorklassen

4.4 Objektprüfung

Sind die Felder oder Properties einer Java-Klasse wie gezeigt mit Validation Constraints versehen, kann die Gültigkeit von Objekten mithilfe einer Instanz des Typs *Validator*[9] ermittelt werden, die man in CDI Beans mittels Injektion erhalten kann. Ist das geprüfte Objekt valide, gibt *validate* eine leere Menge zurück. Andernfalls zeigt der Inhalt der Menge die verschiedenen erkannten Regelverletzungen in Form jeweils eines Elements vom Typ *ConstraintViolation* an. Deren Methoden erlauben den Zugriff u. a. auf den Namen des regelverletzenden Attributs und eine Fehlermeldung. Listing 4.11 zeigt ein Beispiel für eine Objektprüfung inklusive der Sammlung der erkannten Fehler in einer Meldung.

```
public class BeanValDemoController
{
  @Inject
  private Validator     validator;

  private StringBuilder message;
  …
  // Objekt validieren
  private <T> void validate(T object)
  {
    this.message = new StringBuilder();
    Set<ConstraintViolation<T>> constraintViolations
      = this.validator.validate(object);
    for (ConstraintViolation<T> violation : constraintViolations)
      this.message.append(violation.getPropertyPath())
                  .append(' ')
                  .append(violation.getMessage())
                  .append('\n');
  }
  …
```

Listing 4.11: Validierung eines Objekts

9 *javax.validation.Validator*

4.5 Internationalisierung der Validierungsmeldungen

Wie schon angesprochen, akzeptieren die Constraints einen Parameter namens *message*, mit dem der Fehlertext spezifiziert wird, der im Fall einer Regelverletzung genutzt werden soll. Die Fehlermeldungen werden in aller Regel nicht als statische Texte im Programm angegeben, sondern als Referenzen auf das Message Bundle namens *ValidationMessages*. Dazu wird als *message* ein Wert der Form *"{key}"* verwendet, wobei *key* der Schlüssel der Meldung im Message Bundle ist. Der Parameter hat für die Standard-Constraints jeweils einen Default-Wert, der den qualifizierten Klassennamen des Constraints mit angehängten *.message* als Schlüssel nutzt. Diese Namenskonvention ist auch für selbstdefinierte Constraints anzuraten (Listing 4.12).

```
// Validation Constraint für (deutsche) Postleitzahlen
@Constraint(validatedBy = {})
@Pattern(regexp = "\\d{5}")
@ReportAsSingleViolation
@Target({ FIELD, METHOD })
@Retention(RUNTIME)
public @interface ValidPlz
{
  String message() default "{de.gedoplan.….ValidPlz.message}";
  Class<?>[] groups() default {};
  Class<? extends Payload>[] payload() default {};
}

# Auszug aus der Classpath-Ressource ValidationMessages.properties
de.gedoplan.….ValidPlz.message=must contain exactly 5 digits

# Auszug aus der Classpath-Ressource ValidationMessages_de.properties
de.gedoplan.….ValidPlz.message=muss genau 5 Ziffern enthalten
```

Listing 4.12: Nutzung des Message Bundles „ValidationMessages“

Die Texte im Message Bundle dürfen Platzhalter für die Parameter des jeweiligen Constraints haben, um die Meldung damit aussagekräftiger gestalten zu können. So ist bspw. die englische Default-Meldung des Standard-Constraints *@Size*: *size must be between {min} and {max}*. Bei der Erzeugung einer konkreten Meldung aufgrund einer Regelverletzung werden die darin enthaltenen Platzhalter *{min}* und *{max}* durch den konkret genutzten Parameterwert des Constraints ersetzt.

Die Spezifikation macht keine Aussage darüber, welche Sprachen standardmäßig unterstützt werden. Die Referenzimplementierung Hibernate Validator enthält englische, deutsche und französische Texte. In der Praxis wird man die Texte aber ohnehin überarbeiten und ergänzen wollen. Dazu fügt man der Anwendung im einfachsten Fall wie in Listing

4.12 angedeutet die Ressourcendatei *ValidationMessages.properties* sowie je eine weitere *ValidationMessages_xx.properties* für jede Sprache *xx* hinzu. Ausgangspunkt können die entsprechenden Dateien aus Hibernate Validator (dort im Paket *org.hibernate.validator*) sein.

4.6 Validierungsgruppen

Zu verschiedenen Zeiten der Verarbeitung von Daten können unterschiedliche Ausprägungen der Gültigkeit von Objekten existieren. So ist es z. B. denkbar, dass bei der ersten Erfassung eines Fragebogens nur einige Felder benötigt werden, für die spätere Verarbeitung aber sämtliche Attribute mit Werten versehen sein müssen. Wünschenswert ist also eine Gruppierung der Gültigkeitsregeln und damit verbunden die Möglichkeit, die Validierung eines Objekts auf solche Regelgruppen einzuschränken.

Bean Validation bietet dazu ein im Kern einfaches Vorgehen an: Als Validierungsgruppen können beliebige Interfaces verwendet werden. In den meisten Fällen sind dies reine Marker Interfaces, es können aber auch „echte“ Interfaces mit in ihnen deklarierten Methoden verwendet werden. Die Constraint-Annotationen akzeptieren – wie oben bereits angesprochen – den Parameter *groups*. Als Wert kann ein Array von Validierungsgruppen angegeben werden. Das vordefinierte Interface *Default*[10] gilt implizit als Wert, wenn der Parameter *groups* nicht angegeben wird.

```
public interface InitialInput {}

public class Fragebogen
{
  …
  @NotNull(groups = { InitialInput.class, Default.class })
  @Past(groups = { InitialInput.class, Default.class })
  private Date    umfrageDatum;
  …
  @NotNull // implizit: groups = Default.class
  @Size(min = 10, max = 140,
        groups = { InitialInput.class, Default.class })
  private String  bemerkungen;
  …
```

Listing 4.13: Deklaration von Validierungsgruppen

Die bereits angesprochene Validierungsmethode *Validator.validate* akzeptiert als zusätzliche Parameter eine beliebige Menge von Validierungsgruppen. Der Test berücksichtigt dann nur die Regeln, die mindestens einer der angegebenen Gruppen zugeordnet sind (Listing 4.14).

10 *javax.validation.groups.Default*

```
// Normale Prüfung
constraintViolations = validator.validate(fragebogen);

// Prüfung für Gruppe InitialInput
constraintViolations = validator.validate(fragebogen,
                                          InitialInput.class);


// Prüfung für Gruppen Group1 und Group2
constraintViolations = validator.validate(fragebogen,
                                          Group1.class, Group2.class);
```

Listing 4.14: Validierung für bestimmte Gruppen

Ableitungsbeziehungen der Validierungsgruppen sind erlaubt und werden unterstützt. Wäre also bspw. das Interface *AllGroups* von *Group1* und *Group2* abgeleitet, würde *validator.validate(someObject, AllGroups.class)* die Regeln von *Group1* und *Group2* prüfen.

Es gibt noch weitere Einsatzbereiche von Validierungsgruppen: zum einen können Validierungen mithilfe von Gruppen in eine bestimmte Reihenfolge gebracht werden, um z. B. Tests mit hohem Ressourcenbedarf nur dann durchzuführen, wenn zuvor andere Prüfungen erfolgreich waren. Zum anderen kann die Bedeutung der Gruppe *Default* für einzelne Klassen neu definiert werden. Details dazu finden sich in der Spezifikation (Abschnitt 3.4. Group and Group Sequence).

4.7 Integration in JPA und JSF

Bislang wurde in diesem Kapitel die explizite, programmatische Nutzung von Bean Validation in der Geschäftslogik – bspw. in CDI Beans – betrachtet. Es gibt aber weitere, implizite Auswirkungen auf andere Teile von Enterprise-Applikationen. Das betrifft die Speicherung von Daten mittels Java Persistence. Die Integration von Bean Validation geschieht im Entity Lifecycle direkt nach der Stelle, an der *@PrePersist-*, *@PreUpdate* und *@PreRemove-*Methoden aufgerufen werden. Mithilfe des Elements *validation-mode* der Konfigurationsdatei *persistence.xml* kann bestimmt werden, ob Bean Validation benutzt werden soll:

- *CALLBACK:* automatisch validieren
- *NONE:* keine automatische Validierung
- *AUTO:* wie *CALLBACK*, wenn Bean Validation vorhanden, sonst wie *NONE*

Voreingestellt ist *AUTO*. In Java-EE-6-Umgebungen ist stets ein Provider für Bean Validation vorhanden, sodass mit dieser Voreinstellung die automatische Validierung aktiviert ist. Mithilfe von ebenfalls in *persistence.xml* einzutragen Properties kann zudem eingestellt werden, welche Validierungsgruppen für die automatische Validierung an den drei genannten Lifecycle-Stellen verwendet werden sollen. Als Wert kann jeweils eine kommaseparierte Liste von Validierungsgruppen (voll qualifizierte Klassennamen) angegeben werden. Vor-

eingestellt ist *javax.validation.groups.Default* für das Speichern von Daten und ein Leerstring – gleichbedeutend mit „keine Validierung" – für das Löschen von Einträgen. Listing 4.15 zeigt einen Ausschnitt des Persistenzdeskriptors mit den voreingestellten Werten.

```
<persistence …>
  <persistence-unit name="…">
    …
    <validation-mode>AUTO</validation-mode>
    <properties>
      <property name="javax.persistence.validation.group.pre-persist"
                value="javax.validation.groups.Default"/>
      <property name="javax.persistence.validation.group.pre-update"
                value="javax.validation.groups.Default"/>
      <property name="javax.persistence.validation.group.pre-remove"
                value=""/>
      …
```

Listing 4.15: Voreingestellte Validierungsparameter in „META-INF/persistence.xml"

Eine besondere Behandlung erfahren noch nicht geladene Entity-Attribute: Sie werden nicht nachgeladen und damit auch nicht validiert. Zudem wird für Relationsattribute die Annotation @*Valid* ignoriert, weil – entsprechende Kaskadierung vorausgesetzt – die referenzierten Entities in ihrem Lifecycle selbst eine Validierung ausführen. Somit wird verhindert, dass noch nicht geladene „Lazy"-Attribute alleine für die Validierung nachgeladen oder Validierungen doppelt ausgeführt werden.

Bei einem Validierungsfehler werfen die Persistence-Operationen eine *ConstraintViolationException*[11]. Dadurch wird insbesondere die normalerweise ablaufende Operation in der Datenbank für invalide Daten unterdrückt. Die Validierungsmeldungen sind in der Exception eingetragen und können mit der Methode *getConstraintViolations* abgeholt werden (Listing 4.16).

```
try
{
  entityManager.persist(someObject);
}
catch (ConstraintViolationException e)
{
  Set<ConstraintViolation<?>> constraintViolations
    = e.getConstraintViolations();

  for (ConstraintViolation<?> constraintViolation : constraintViolations)
    System.out.println("  " + constraintViolation.getPropertyPath()
                       + " " + constraintViolation.getMessage());
}
```

Listing 4.16: Verarbeitung von Validierungsfehlern bei JPA-Operationen

11 *javax.validation.ConstraintViolationException*

Inwieweit der Java-Persistence-Provider die Validierungsconstraints auch für DDL verwendet, lässt die Spezifikation offen. Üblich ist, dass zumindest *@NotNull* und *@Size.max* beachtet und in ein *not null* bzw. eine entsprechende Längenangabe bei der Definition der Tabellenspalten umgesetzt werden.

Der zweite Java-EE-Bereich, der implizit von Bean Validation profitiert, ist JavaServer Faces. Die JSF-eigenen Validierungsmöglichkeiten lassen sich in den allermeisten Fällen komplett durch Bean Validation ablösen. Details dazu folgen im entsprechenden Buchkapitel.

4.8 Bean Validation in SE-Umgebungen

Bean Validation lässt sich auch außerhalb von Applikationsservern einsetzen. Dann muss sich neben der in Abschnitt 4.2 erwähnten Standardbibliothek eine Implementierung im Classpath befinden. Die Referenzimplementierung HibernateValidator kann mit der in Listing 4.17 gezeigten Maven Dependency eingebunden werden.

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>4.1.0.Final</version>
  <scope>runtime</scope> <!-- oder test -->
</dependency>
```

Listing 4.17: Maven Dependency für Hibernate Validator

Außerhalb eines Java-EE-Containers steht natürlich keine Injektionsmöglichkeit für den Validator zur Verfügung. Stattdessen wird hier eine Factory verwendet. Die Beschaffung der Validator-Instanz erinnert an die Erzeugung eines *EntityManager*s für Java Persistence: Zunächst wird mithilfe der Standardklasse *Validation*[12] eine *ValidatorFactory*[13] erzeugt, die später einen *Validator* liefern kann. Die Erzeugung der Factory ist eine aufwändige Operation, das erzeugte Objekt ist thread-safe. Im Gegenzug ist die Lieferung des Validators einfach, der Validator aber nicht thread-safe. Demzufolge wird man die Factory einmalig vorweg erzeugen, um sich daraus bei Bedarf jeweils den Validator geben zu lassen (Listing 4.18).

```
// Einmalig
ValidatorFactory validatorFactory
  = Validation.buildDefaultValidatorFactory();

// Jeweils bei Bedarf
Validator validator = validatorFactory.getValidator();
```

Listing 4.18: Erzeugung eines Validators mithilfe einer Factory

12 *javax.validation.Validation*
13 *javax.validation.ValidatorFactory*

5 JavaServer Faces

5.1 Einsatzzweck von JSF

Der überwiegende Teil von Enterprise-Anwendungen ist mit einem grafischen Benutzerinterface ausgestattet, mit dem Anwendungsdaten visualisiert, Eingaben durch den Benutzer gemacht und Aktionen ausgelöst werden können. Häufig wird zu diesem Zweck ein Webbrowser verwendet, d. h. die Präsentation ist z. B. HTML-basiert und die Kommunikation zwischen Browser und Anwendung geschieht mittels HTTP.

Der Vorteil dieser Konstellation liegt u. a. darin, dass der Anwender keinerlei Softwareinstallation über die Bereitstellung eines Webbrowsers hinaus durchführen muss. Im Gegenzug muss die Anwendung die Aufbereitung des zur Präsentation genutzten HTML-Texts übernehmen und die vom Browser ausgelösten HTTP Requests verarbeiten.

Technisch lässt sich diese Anforderung mit Webanwendungen und den darin enthaltenen Servlets realisieren. Der nächste Abschnitt geht auf diese Anwendungsbasis im Überblick ein. Der Aufbau einer kompletten Anwendung ausschließlich mit Servlets wäre allerdings sehr aufwändig. Hier wird eher eine Anwendungsschicht mit einem höheren Abstraktionsgrad benötigt, die in geeigneter Weise die Präsentation in HTML mit Daten und Methoden von Java-Objekten verbindet und eine Verknüpfung der Views der Anwendung untereinander ermöglicht. Der Java-EE-Standard bietet dazu JavaServer Faces – kurz JSF – an. Anwendungen lassen sich damit aus HTML-ähnlichen Seitenbeschreibungen aufbauen, die Daten und Logik von Java-Objekten verwenden und referenzieren.

5.2 Die Basis: Java-Webanwendungen

5.2.1 Grundlegender Aufbau

Webanwendungen bestehen zunächst aus einer Menge von Dokumenten, die im Browser zur Anzeige gebracht werden sollen. Es können HTML-Dokumente sein oder auch Grafiken, PDF-Dateien etc. Sie befinden sich im Startverzeichnis der Anwendung oder in passend benannten Unterverzeichnissen.

Der statische Teil der Webanwendung wird ergänzt um kompilierte Java-Klassen und Ressourcedateien. Sie finden im Verzeichnis *WEB-INF/classes* Platz oder im Fall von Bibliotheken in *WEB-INF/lib*. Schließlich wird die Anwendung mit einem Deployment Descriptor namens *web.xml* und ggf. weiteren Parameterdateien im Verzeichnis *WEB-INF* konfiguriert.

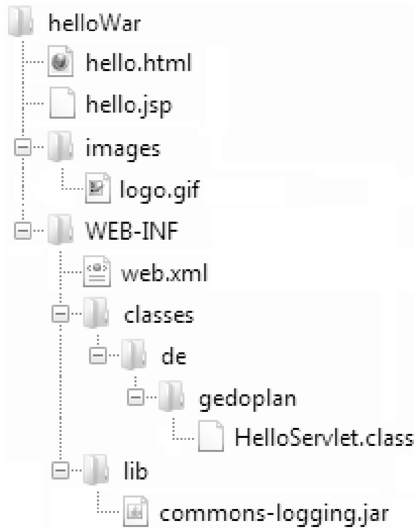


Abbildung 5.1: Aufbau einer Webanwendung

Ein Einstiegsbeispiel zeigt Abbildung 5.1: Im willkürlich benannten Startverzeichnis *helloWar* und dem Unterverzeichnis *images* befinden sich einige statische Dokumente. *WEB-INF* enthält den Deployment Descriptor, eine kompilierte Klasse und eine Bibliothek.

Webanwendungen werden zum Deployment auf einem geeigneten Server mit *jar* gepackt und erhalten dabei die Dateiendung *.war*. Das Deployment-Verfahren ist abhängig vom Server. In vielen Fällen reicht es, eine Kopie der *war*-Datei in ein dafür bestimmtes Verzeichnis zu speichern.

Der Name der Deployment-Datei ohne ihre Endung bestimmt den sog. Web Context, der Teil des URL zur Adressierung der Anwendungsteile wird. Würde obiges Beispiel als Datei *helloWar.war* auf einem lokalen JBoss-Server deployt, müsste man im Browser die Adresse *http://localhost:8080/helloWar/hello.html* nutzen, um die entsprechende Webseite angezeigt zu bekommen.

5.2.2 Servlets

Die gezeigte Struktur wird allerdings erst dann zu einer „richtigen" Anwendung, wenn sie dynamische Anteile enthält, d. h. Java-Klassen, deren Methoden während der Request-Verarbeitung aufgerufen werden, sodass zu diesem Zeitpunkt dynamischer Inhalt erzeugt und im Browser angezeigt werden kann. Diese Aufgabe übernehmen Servlets. Sie werden im Folgenden skizziert, allerdings ohne auf Details einzugehen, die in der Servlet-Spezifikation[1] nachgelesen werden können.

1 JavaTM Servlet Specification, Version 3.0, Rajiv Mordani, December 2009, *http://jcp.org* –> Search JSR 315 –> Final Release Download –> servlet-3_0-final-spec.pdf

entwickler.press

Unter Servlets versteht man Klassen, die i. d. R. von *HttpServlet*[2] abgeleitet werden. Ihre Methode *doGet* wird zur Verarbeitung eines HTTP-GET-Requests aufgerufen. Ähnliche Methoden sind auch für die anderen HTTP-Verben (POST etc.) vorgesehen. Die beiden Parameter vom Typ *HttpServletRequest*[3] und *HttpServletResponse*[4] stellen Ein- und Ausgabe des Servlets dar: Die jeweilige Methode verarbeitet die Request-Parameter und füllt das Response-Objekt u. a. mit dem Text, den der Browser schließlich anzeigen soll (Listing 5.1).

```
public class HelloServlet extends HttpServlet
{
  protected void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
  {
    resp.setContentType("text/html");   // Response ist HTML
    PrintWriter out = resp.getWriter();
    out.println("<html>");             // Anzeigetext ausgeben
    out.println("<body>");
    out.println("  Hallo, Welt!");
    out.println("</body>");
    out.println("</html>");
    out.close();
  }
}
```

Listing 5.1: Servlet[5]

Eines fehlt allerdings noch: Die Klasse muss als Servlet registriert und mit einem URL verknüpft werden. Das ist mithilfe der Annotation *@WebServlet* möglich (Listing 5.2) oder alternativ durch Einträge im Descriptor *web.xml* (Listing 5.3).

```
@WebServlet(urlPatterns = "/helloServlet")
public class HelloServlet extends HttpServlet
{
  …
```

Listing 5.2: Servlet-Registrierung per Annotation

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">
```

2 *javax.servlet.http.HttpServlet*
3 *javax.servlet.http.HttpServletRequest*
4 *javax.servlet.http.HttpServletResponse*
5 Den in diesem Kapitel gezeigten Beispielcode finden Sie im Begleitprojekt *ee-demos-jsf*

```
  <servlet>
    <servlet-name>helloServlet</servlet-name>
    <servlet-class>de.gedoplan.….HelloServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>helloServlet</servlet-name>
    <url-pattern>/helloServlet</url-pattern>
  </servlet-mapping>
</web-app>
```

Listing 5.3: Servlet-Registrierung im Descriptor „web.xml“

Ein Servlet kann wie im Listing gezeigt mit einem exakten Pfad in der Webanwendung verknüpft werden: Das URL-Pattern */helloServlet* bewirkt, dass das Servlet in unserer Beispielanwendung unter dem URL *http://localhost:8080/helloWar/helloServlet* aufrufbar ist. Alternativ sind Verknüpfungen mit Verzeichnissen (*/somePath/**) oder Endungen (**.xhtml*) möglich. Einem Servlet können mehrere URL-Patterns zugeordnet werden.

5.2.3 JavaServer Pages

Es stellte sich in der Vergangenheit schnell heraus, dass Servlets zwar ein mächtiges Werkzeug für die dynamische Request-Verarbeitung sind, die Gestaltung von Webseiten damit aber recht mühsam ist. Schließlich muss der gesamte HTML-Text mithilfe von Ausgabeanweisungen im Servlet erstellt werden.

Abhilfe verspricht die Umkehrung der Vorgehensweise: Statt im Java-Code HTML-Text auszugeben, gestaltet man eine HTML-Seite mit speziellen Tags, die Java-Code enthalten. Der Java-EE-Standard bietet dazu JavaServer Pages – kurz JSP – an. Sie werden in Webanwendungen wie statische Dokumente integriert, allerdings mit der Dateiendung *.jsp* statt *.html*. Zur Laufzeit der Anwendung werden JSPs spätestens bei der ersten Benutzung in äquivalente Servlets übersetzt.

```
<%@ page contentType="text/html; charset=UTF-8" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <body>
    <h2>"Hallo, Welt!"</h2>
    <br/>
    Dies ist eine JSP. Es ist <%= new java.util.Date() %>
  </body>
</html>
```

Listing 5.4: JavaServer Page

Wird das in Listing 5.4 gezeigte Beispiel als *hello.jsp* in unsere Beispielanwendung integriert, so führt der URL *http://localhost:8080/helloWar/hello.jsp* zur Anzeige einer Seite im Browser, in der im Text der Zeitpunkt des Requests eingeflochten ist. Das geschieht durch

entwickler.press

den im Tag <%= *new java.util.Date()* %> enthaltenen Java-Code. JSP kennt diverse weitere Tags zur Integration von Deklarationen und Anweisungen, die für die weitere Betrachtung aber unerheblich sind. Bei Interesse finden Sie die Details in der JSP-Spezifikation[6].

JSPs verfolgen zwar durch die beschriebene Umkehr der Vorgehensweise den richtigen Ansatz, bescheren dem Entwickler aber neue Probleme: Zum einen wird der in JSPs enthaltene Java-Code zur Laufzeit der Anwendung kompiliert. Übersetzungsfehler treten somit erst dann auf. Zudem beziehen sich die Fehlermeldungen nicht direkt auf das JSP-Dokument, sondern auf einen vom Server temporär daraus erstellten Java-Quelltext. Zum anderen ist die Gefahr sehr groß, durch die Mischung von HTML- und Java-Code unübersichtliche Programme zu schreiben. Viele reale Projekte sahen sich dadurch am Ende mit praktisch unwartbarem Code konfrontiert.

Eine Lösung dieser Problematik besteht darin, Java-Code nicht mehr direkt in die Webdokumente zu integrieren, sondern ihn in Bibliotheken auszulagern und mit speziellen Tags zu referenzieren. Somit entsteht auf der Seite der Webdokumente eine erweiterte, HTML-ähnliche Sprache, was sich mithilfe von XML und Namespaces gut und ohne konzeptionellen Bruch realisieren lässt. Die andere Seite des referenzierten Java-Codes ist nun in normalen Bibliotheken enthalten, die mit herkömmlichen Mitteln im normalen Build-System erstellt werden können. Ein Ausprägung dieser Vorgehensweise sind die weiter unten beschriebenen Facelets, die in JavaServer Faces als präferierte View-Beschreibung eingesetzt werden.

5.3 JSF im Überblick

5.3.1 Model View Controller

Model View Controller oder auch kurz MVC ist ein Architekturmuster für die Softwareentwicklung. Es trennt drei Aspekte der Software und weist ihnen klare Gestaltungsarten zu:

- Das Model enthält die bearbeiteten Daten. Die Geschäftslogik ist hier mit enthalten oder wird von hier aufgerufen.
- Die View dient der Darstellung der Daten. Sie übernimmt auch die Interaktion mit dem Benutzer, also insbesondere die Annahme von Benutzeraktionen wie Eingaben und Kommandos, ist aber nicht für die eigentliche Verarbeitung zuständig.
- Der Controller verwaltet Views und Models und verknüpft dabei insbesondere die von den Views gelieferten Benutzerkommandos mit den Daten und der Logik der Models.

6 JavaServer Pages™, Version 2.1, Pierre Delisle, Jan Luehe, Mark Roth, May 2006, *http://jcp.org* –> Search JSR 245 –> Final Release Download –> jsp-2_1-fr-spec.pdf

Das Ziel von MVC ist eine höhere Flexibilität für Änderungen und Erweiterungen und eine größere Wiederverwendbarkeit der einzelnen Komponenten. Das Konzept wurde Ende der 1970er Jahre zunächst für Benutzeroberflächen in Smalltalk beschrieben (Abb. 5.2).

Abbildung 5.2: Model View Controller

Im Fall von JavaServer Faces kommt MVC in der folgenden Ausprägung zum Einsatz: Der Controller wird durch das Faces Servlet implementiert. Die Views sind Facelets, d. h. XHTML-Dokumente, die einige besondere Tag-Bibliotheken verwenden. In JSF können auch andere View-Technologien zum Einsatz kommen, worauf aber hier nicht weiter eingegangen werden soll. Die Models schließlich werden als POJOs (Plain Old Java Objects) bereitgestellt, d. h. als einfache Java-Objekte.

Alle Anwendungs-Requests werden vom Faces Servlet behandelt. Aufgrund des Status der Anwendung wird entschieden, welche Beans und Views verwendet werden sollen. Die Antwort wird schließlich von einer View erzeugt und im Browser angezeigt (Abb. 5.3).

Abbildung 5.3: MVC in JSF

5.3.2 Facelets

Die Sprache zur Seitenbeschreibung ist in der Spezifikation nicht festgelegt. Jede Implementierung muss aber zumindest Facelets und JSP unterstützen. Seit JSF 2.0 werden Facelets präferiert. Das sind XHTML-Dokumente entsprechend der Definition des W3C. Sie können um Tags aus weiteren Namensräumen ergänzt werden, um die spezielle Funktionalität von Facelets zu nutzen:

- *http://java.sun.com/jsf/html:* UI-Komponenten (Ein/Ausgabe-Elemente, Aktionselemente, …)
- *http://java.sun.com/jsf/core:* Strukturelemente, Parameterelemente. Modifizieren das Verhalten von anderen Tags, die sie enthalten oder umschließen
- *http://java.sun.com/jsf/facelets:* Tags zur Definition und Nutzung von Templates
- *http://java.sun.com/jsp/jstl/core* und *http://java.sun.com/jsp/jstl/functions*: Facelet-Version der JSTL-Tags (JSP Standard Tag Library)
- *http://java.sun.com/jsf/composite*: Tags zur Definition von eigenen Komponenten

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
<head>
  <title>First Facelet</title>
</head>
<body>
  <h:outputText value="Hello World!" />
  <br/>
  <h:outputText value="Ihr Browser: #{header['User-Agent']}" />
</body>
</html>
```

Listing 5.5: Einfaches Facelet

Ein Einstiegsbeispiel für ein Facelet zeigt Listing 5.5. Es benutzt das Tag *<h:outputText>* aus der HTML-Bibliothek zur Ausgabe von Text. Der Ausdruck *#{…}* stammt aus der JSF Expression Language – kurz JSF-EL – und wird zur Request-Zeit ausgewertet. Die weiteren Abschnitte dieses Kapitels gehen genauer auf die verfügbaren Tags und die JSF-EL ein.

5.3.3 Request-Verarbeitung

Wie oben dargestellt, werden Anwendungs-Requests vom zentralen Faces Servlet angenommen. Die Verarbeitung geschieht dann im sog. Request Processing Lifecycle. Dieser Lebenszyklus umfasst sechs Phasen von der Initialisierung der Verarbeitung über die Annahme von Request-Parametern bis schließlich zum Rendern der Antwort (Abb. 5.4).

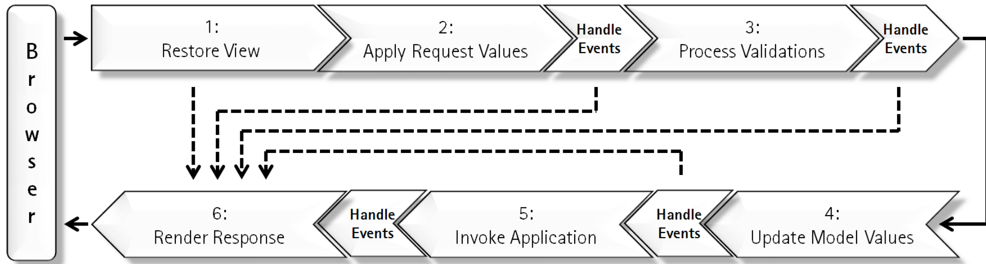


Abbildung 5.4: Request Processing Lifecycle

An die Phasen schließt sich meist eine Event-Verarbeitung an. Hierin kann man mit verschiedenen Event Listenern auf Zustandswechsel o. ä. reagieren.

Beim Auftreten von Verarbeitungsfehlern (Validierungsfehler etc.) wird die Request-Verarbeitung direkt mit der letzten Phase zur Anzeige der View abgeschlossen. Das kann in der Event-Verarbeitung auch programmatisch ausgelöst werden.

Während der Event-Verarbeitung kann zudem auf das Rendern einer Antwort komplett verzichtet werden (in der Abbildung nicht dargestellt). Das ist beispielsweise der Fall, wenn binäre Daten als Antwort an den Client gesendet werden, z. B. Bilder oder PDF-Dokumente.

In Phase 1 (Restore View) wird die interne Darstellung der aktuellen View hergestellt. Darunter versteht man eine baumartige Objektstruktur, die den ineinander verschachtelten Komponenten der View entspricht. Wird eine View zum ersten Mal benutzt, existiert dieser interne Komponentenbaum noch nicht. Dann wird er entsprechend der Struktur der Seite aufgebaut und die restlichen Verarbeitungsschritte bis auf die Rendering-Phase entfallen dann. Bei Folgeaufrufen wird der Komponentenbaum aus dem gespeicherten Status heraus wieder hergestellt und der komplette Lebenszyklus durchlaufen.

Der Komponentenbaum einer View wird zwischen zwei Requests gespeichert. Die Ablage geschieht i. d. R. serverseitig in der Session, kann aber auch clientseitig mithilfe von versteckten Feldern in den Views geschehen.

Einige Komponenten lassen die Eingabe von Werten zu (Texte, Listenauswahl, …). Die Eingabewerte werden im Request als Parameter mitgeliefert. Phase 2 (Apply Request Values) übernimmt die Eingabewerte als sog. Submitted Values in die zugehörigen Objekte des Komponentenbaums. Die alten Werte der Komponenten werden nicht verändert, um einen späteren Vergleich noch zu ermöglichen.

In Phase 3 (Process Validations) werden die übermittelten Werte in den gewünschten Zieldatentyp konvertiert und auf Gültigkeit geprüft. Einige Komponenten haben implizite Konverter und Validatoren, weitere können bei Bedarf registriert werden. Verlaufen die Konvertierung und die Validierung positiv, werden die Eingabewerte endgültig in den Komponenten abgespeichert. Ergibt sich dabei eine Werteänderung, wird ein ent-

sprechender Event ausgelöst. Bei Konvertierungs- oder Validierungsfehlern werden die Phasen 4 und 5 übersprungen und die Request-Bearbeitung wird mit der letzten Phase fortgesetzt.

In den bisherigen Phasen waren nur die Objekte des Komponentenbaums beteiligt. In Phase 4 (Update Model Values) werden nun die Werte von dort in die assoziierten Model-Objekte kopiert. Durch die bisherige Verarbeitung ist sichergestellt, dass die Werte valide sind.

Einige Komponenten haben einen natürlichen Aktionscharakter, z. B. Buttons oder Links. Ihnen werden normalerweise Methoden zugeordnet, die die entsprechende Anwendungslogik enthalten. Sie werden in Phase 5 (Invoke Application) aufgerufen, wenn der Request von einem solchen Aktionselement ausgelöst wurde. Anschließend findet die Navigation in der Anwendung statt, d. h. die Festlegung der als Nächstes anzuzeigenden View.

Als Abschluss der Request-Bearbeitung wird in Phase 6 (Render Response) der anzuzeigende Inhalt erzeugt. In dieser Phase wird zudem der Komponentenbaum für weitere Anfragen der gleichen View abgespeichert.

Der dargestellt Lebenszyklus kann über die Attribute der beteiligten Komponenten noch partiell modifiziert werden. Darauf gehen die nachfolgenden Abschnitte 5.18 (Immediate-Komponenten) und 5.19 (Ajax) noch ein.

5.4 Konfiguration der Webanwendung

Eine JSF-Anwendung unterscheidet sich im Aufbau nicht von einer herkömmlichen Webanwendung, wie sie oben in Abbildung 5.1 dargestellt wurde. In ihrem Deployment Descriptor *WEB-INF/web.xml* wird das Faces Servlet registriert und es werden alle Request-URLs zugeordnet, die auf *.xhtml* enden (Listing 5.6).

```
<web-app
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  version="3.0">

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.xhtml</url-pattern>
```

```
    </servlet-mapping>

    <context-param>
      <param-name>javax.faces.PROJECT_STAGE</param-name>
      <param-value>Development</param-value>
    </context-param>
  </web-app>
```

Listing 5.6: Grundkonfiguration der Webanwendung zur Nutzung von JSF

Das Faces Servlet wird durch die Angabe *<load-on-startup>1</load-on-startup>* schon beim Start der Anwendung geladen, was die jeweilige Implementierung zur Initialsierung des Systems nutzen kann. Die Verknüpfung mit der URL-Endung *.xhtml* ist üblich, aber nicht zwingend. Häufig findet man auch die Endungen *.faces* oder *.jsf* vor.

Mit *<context-param>*-Elementen kann die Anwendung weiter konfiguriert werden. Die Angaben sind optional, meist existieren gute Default-Werte. Das Listing zeigt beispielhaft den Parameter *javax.faces.PROJECT_STAGE*, mit dem der Entwicklungsstand der Anwendung deklariert werden kann. Die möglichen Werte sind *Development*, *UnitTest*, *SystemTest* und *Production*. Die gewählte Einstellung führt dazu, dass umfangreichere Meldungen im Fehlerfall angezeigt werden, was während der Entwicklung recht hilfreich ist. Einige praxisrelevante Parameter sind in Tabelle 5.1 zusammengefasst. Weitere Informationen finden Sie in der JSF-Spezifikation (Abschnitt 11.1.3, Application Configuration Parameters).

Parameter *javax.faces....*	**Bedeutung**	**Default**
DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE		*false*
	Für die Konvertierung von Date-Werten nicht UTC, sondern die Systemzeitzone verwenden	
FACELETS_SKIP_COMMENTS		*false*
	XML-Kommentare in den Views nicht zum Client senden	
INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL		*false*
	Leere Eingabetexte als null weiterverarbeiten	
PROJECT_STAGE	Entwicklungsstand der Anwendung	*Production*

Tabelle 5.1: JSF-Konfigurationsparameter

Zusätzlich zu dieser allgemeinen Konfiguration des JSF-Systems in der Webanwendung kann die Datei *WEB-INF/faces-config.xml* genutzt werden, um das Verhalten der JSF-Anwendung einzustellen. So finden dort bspw. die später beschriebenen Navigationsregeln oder Internationalisierungsparameter ihren Platz. Als Ausgangspunkt kann die in Listing 5.7 gezeigte quasi-leere Datei dienen. Seit JSF 2.0 ist sie sogar optional, kann also im ersten Schritt auch komplett entfallen.

```
<faces-config
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-facesconfig_2_0.xsd"
  version="2.0">
</faces-config>
```

Listing 5.7: Quasi-leere Konfigurationasdatei „faces-config.xml“

5.5 Benötigte Bibliotheken und Plattformen

Zur Entwicklung von JSF-Anwendungen wird die JSF-Standard-Bibliothek benötigt. Sie kann z. B. als Maven Dependency im Projekt eingebunden werden (Listing 5.8). Die Bibliothek ist im Zielserver bereits vorhanden, sodass der Scope *provided* ausreicht, wodurch die Bibliothek nicht in die Webanwendung integriert wird.

```
<dependency>
  <groupId>javax.faces</groupId>
  <artifactId>jsf-api</artifactId>
  <version>2.0</version>
  <scope>provided</scope>
</dependency>
```

Listing 5.8: Maven Dependency für das JSF API

Als Plattform kann jeder Java-EE-6-Server dienen, wobei das Web Profile ausreichend ist, also z. B. GlassFish 3, JBoss 6, Oracle Weblogic 12 oder IBM WebSphere 8. Die darin eingesetzte JSF-Implementierung ist zumeist die Referenzimplementierung JSF-RI oder Apache MyFaces. Bei einigen Servern sind sogar mehrere Implementierungen enthalten. Für die Entwicklung von JSF-Anwendungen spielt das aber keine Rolle – soweit die Implementierung keine Fehler enthält.

5.6 Programmierung der Views

Die JSF-Views werden in einer geeigneten Template-Sprache beschrieben. Alle Implementierungen müssen hier JSP und Facelet unterstützen. In diesem Buch werden ausschließlich Facelets verwendet. Sie bilden seit JSF 2.0 den Standard und bedürfen keiner weiteren Konfiguration.

Die Seiten werden als Dateien mit der Endung *.xhtml* programmiert, wobei die im Folgenden beschriebenen JSF Tags eingesetzt werden können.

Der Aufruf geschieht allerdings nicht direkt, sondern über einen URL mit der Endung, auf die das Faces Servlet gemappt ist, also z. B. **.xhtml*. Selbst wenn sie mit der Endung der Seitendateien übereinstimmt, werden diese nicht direkt ausgeliefert. Vielmehr wird immer das Faces Servlet aufgerufen, was zum Rendern der Ergebnisseite die entsprechende Seitendatei benutzt.

5.6.1 JSF Tag Libraries

Die Views benötigen i. d. R. zwei grundlegende Tag-Bibliotheken, die über die weiter oben angeführten Namensräume referenziert werden: Die HTML-Bibliothek enthält Komponenten wie Ein- und Ausgabefelder, Buttons etc., aus denen die Benutzeroberfläche zusammengestellt wird. Ihr Namensraum *http://java.sun.com/jsf/html* wird meist mit dem Präfix *h* in die Views eingebunden. Die Tags der Core-Bibliothek werden nicht direkt zur Anzeige gebracht. Vielmehr dienen sie der Gruppierung oder Parametrierung von anderen Elementen, die sie umschließen oder enthalten. Der zugehörige Namensraum *http://java.sun.com/jsf/core* wird üblicherweise mit dem Präfix *f* in die Seiten integriert. Ein Facelet hat somit grundsätzlich den in Listing 5.9 gezeigten Aufbau.

```
<html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:f="http://java.sun.com/jsf/core">
<h:head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
  <title>…</title>
</h:head>
<h:body>
  …


</h:body>
</html>
```

Listing 5.9: Grundsätzlicher Aufbau eines Facelets

HTML-Bibliothek

Die HTML-Bibliothek enthält Tags für UI-Komponenten wie Textausgabe oder diverse Varianten von Eingabe- und Aktionselementen. In der Rendering-Phase werden sie durch die entsprechenden HTML-Elemente dargestellt. Facelets dürfen darüber hinaus beliebigen HTML-Text enthalten, wodurch sich automatisch Überschneidungen mit der HTML-Bibliothek ergeben. In einem solchen Fall muss dem jeweiligen Tag aus der Bibliothek der Vorzug gegeben werden, also z. B. <*h:head*> und <*h:body*> genutzt werden und nicht <*head*> bzw. <*body*>. Einige Tags werden nämlich nicht nur als entsprechendes HTML-Element gerendert, sondern führen beispielsweise zur Integration von Stylesheets oder Skripten in die Ausgabe.

Die HTML-Bibliothek enthält u. a. einige Tags zur Ausgabe von Texten, Links und Grafiken (Tabelle 5.2).

Tag	Beschreibung	Beispiel
h:outputText	Textausgabe	*<h:outputText value="Hallo"/>*
h:outputFormat	Ausgabe eines parametrisierten Texts	*<h:outputFormat value="Sehr geehrte{0} {1}">* *<f:param value="…"/>* *<f:param value="…"/>* *</h:outputFormat>*
h:outputLabel	Ausgabe eines Labels	*<h:outputLabel for="price" value="Preis"/>* *<h:inputText id="price" …/>*
h:outputLink	Ausgabe eines Hyperlinks	*<h:outputLink value="http://www.gedoplan.de"/>*
h:graphicImage	Ausgabe einer Grafik	*<h:graphicImage url="/images/logo.gif"/>*

Tabelle 5.2: Ausgabe-Tags in der HTML-Bibliothek

Sie funktionieren alle in ähnlicher Weise: Ihr Attribut *value* (*url* bei *graficImage*) bestimmt, was sie zur Anzeige bringen. Mit weiteren Parametern kann das Verhalten weiter gesteuert werden. Als Referenz für die Tags kann die – allerdings nicht sehr übersichtliche – Dokumentation der JSF-Referenzimplementierung genutzt werden: *http://javaserverfaces.java.net/nonav/docs/2.0/pdldocs/facelets/index.html*.

Alle Tags akzeptieren das Attribut *id* zum Setzen der Komponenten-ID sowie den Boole'schen Parameter *rendered*. Wenn er *false* ist, wird die Anzeige der Komponente unterdrückt.

```
<html …>
<h:head>
  <meta http-equiv="content-type" content="text/html; charset=UTF-8"/>
  <title>Ausgabe-Tags h:outputXxx</title>
</h:head>
<h:body>
  <h:outputText value="Einfacher Text" />
  <hr />
  <h:outputFormat value="Sehr geehrte{0} {1}" rendered="false">
    <f:param value=" Frau" />
    <f:param value="Mustermann" />
  </h:outputFormat>
  <hr />
  <h:outputLabel for="farbe" value="Farbe:" />
  <h:outputText id="farbe" value="rot" />
  <hr />
  <h:outputLink value="http://www.gedoplan.de">
```

```
    <h:graphicImage url="/resources/logos/gedoplan-logo.gif" />
  </h:outputLink>
</h:body>
</html>
```

Listing 5.10: Demonstration diverser Ausgabe-Tags

Die Nutzung der Ausgabe-Tags ist in Listing 5.10 beispielhaft gezeigt. Die angezeigten Werte sind allerdings sämtlich statisch, was natürlich in praktischen Anwendungen nur begrenzt sinnvoll ist. Wir werden später sehen, dass an die Stelle fester Werte Referenzen zu Java-Objekten treten können.

Die HTML-Bibliothek beherbergt auch zwei Tags, mit denen andere Elemente gruppiert werden können (Tabelle 5.3): *h:panelGroup* packt seine Unterelemente zu einem neuen Element zusammen. Die HTML-Ausgabe für den Browser enthält dann abhängig vom Attribut *layout* ein *span*- oder *div*-Element. *h:panelGrid* erzeugt eine Tabelle mit der angegebenen Spaltenanzahl. Die Unterelemente füllen die Spalten in der gegebenen Reihenfolge auf, wodurch sich implizit eine Zeilenanzahl für die Tabelle ergibt.

Tag	Beschreibung	Beispiel
h:panelGroup	Gruppierung von mehreren Elementen zu einem	*<h:panelGroup> <h:outputText value="Hallo, "/> <h:outputText value="Welt"/> </h:panelGroup>*
h:panelGrid	Gruppierung und Gitternetz-anordnung	*<h:panelGrid columns="2"> <h:outputLabel value="Bezeichnung"/> <h:outputText value="DVD-Rohling"/> <h:outputLabel value="Preis"/> <h:outputText value="0,60"/> </h:panelGrid>*

Tabelle 5.3: Gruppierungs-Tags in der HTML-Bibliothek

Zu diesen Gruppierungselementen oder sogar zur HTML-Bibliothek allgemein muss man allerdings einwerfen, dass die Gestaltungsmöglichkeiten und der Umfang in der Standardbibliothek eher gering sind. Wenn Sie Ihre Anwendung attraktiv gestalten wollen, müssen Sie in der Praxis Zusatzbibliotheken wie RichFaces oder PrimeFaces einsetzen.

Weitere Bestandteile der HTML-Bibliothek betreffen Eingabe- und Aktionselemente (Tabelle 5.4 und Tabelle 5.5). Sie sind nur innerhalb eines *h:form*-Elements nutzbar. Die Eingabeelemente haben alle eine ähnliche Grundfunktion: Sie zeigen Werte in einer Form an und übermitteln sie wieder beim nächsten Request, wenn dieser durch ein Aktionselement in der Form ausgelöst wurde. Die Nutzung dieser Elemente wird weiter unten in den Abschnitten 5.7 Managed Beans und 5.8 Unified Expression Language an einigen Beispielen erläutert.

Tag	Beschreibung	Beispiel
h:inputText *h:inputTextarea* *h:inputSecret* *h:inputHidden*	Eingabefeld standard mehrzeilig geheim versteckt	*<h:inputText value="#{bean.valueProperty}"/>*
h:selectBooleanCheckbox	Checkbox	*<h:selectBooleanCheckbox value="#{bean.valueProperty}" />*
h:selectOneListbox *h:selectOneMenu* *h:selectOneRadio*	Einfachauswahl Listbox Drop-Down Radioboxes	*<h:selectOneListbox value="#{bean.valueProperty}" >* *<f:selectItems value="#{bean.listProperty}" />* *</h:selectOneListbox>*
h:selectManyListbox *h:selectManyMenu* *h:selectManyCheckbox*	Mehrfachauswahl Listbox Drop-Down Checkboxes	*<h:selectManyListbox value="#{bean.valueProperty}" >* *<f:selectItems value="#{bean.listProperty}" />* *</h:selectManyListbox>*

Tabelle 5.4: Eingabe-Tags in der HTML-Bibliothek

Tag	Beschreibung	Beispiel
h:commandButton *h:commandLink*	Button Link	*<h:commandButton value="text" action="#{bean.method}" />*

Tabelle 5.5: Aktionselemente in der HTML-Bibliothek

Der Vollständigkeit halber zeigt Tabelle 5.6 die restlichen Bestandteile der HTML-Bibliothek. Sie wurden entweder schon erwähnt (*h:head*, *h:body*, *h:form*) oder haben Spezialaufgaben, die erst später in diesem Kapitel thematisiert werden können.

Tag	Beschreibung	Beispiel
h:head *h:body*	HTML-Rahmen-elemente	*<html >* *<h:head>* *…* *</h:head>* *<h:body>* *…* *</h:body>* *</html>*

Tag	Beschreibung	Beispiel
h:form	Formular	<h:form> <h:inputText …/> <h:commandButton …" /> </h:form>
h:dataTable *h:column*	Tabellarische Datendarstel-lung	<h:dataTable var="item"…> <h:column> <h:outputText value="#{item.name}"/> </h:column> … </h:dataTable>
h:message *h:messages*	Meldungsaus-gabe	<h:inputText id="name" …/> <h:message for="name"/> … <h:messages/>
h:button *h:link*	Erzeugung von GET-Requests	<h:button value="…" outcome="…">
h:outputScript	Script-Ausgabe	<h:outputScript name="…" library="…" target="head"/>
h:outputStylesheet	Stylesheet-Ausgabe	<h:outputStylesheet name="…" library="…"/>

Tabelle 5.6: Weitere Elemente in der HTML-Bibliothek

Core-Bibliothek

Die Tags der Core-Bibliothek kommen nicht unmittelbar zur Anzeige, sondern fügen den UI-Komponenten Metadaten, Parameter oder Funktionalität hinzu. Ein Beispiel ist in Tabelle 5.2 bereits zu sehen: *f:param* übergibt dort Werte, die die Platzhalter des Texts im Tag *h:outputMessage* füllen.

Ein zentrales Tag ist *f:view*. Es ist der Container für die anderen JSF-Tags, muss also den kompletten Seiteninhalt umschließen – mal abgesehen vom *html*-Tag. In Facelets ist es allerdings optional, d. h. es wird implizit an der richtigen Stelle platziert, wenn eine View es nicht explizit enthält. Insofern können Sie dieses Tag auch gleich wieder vergessen…

Ein interessanteres Tag ist *f:facet*. Es kann als Unterelement diverser *h*-Tags eingesetzt werden, um ihnen bestimmte Aspekte hinzuzufügen. Jeder Aspekt ist benannt mit einem vom Einsatzzweck abhängigen Namen. So kann man z. B. einer mit *h:panelGrid* erstellten Tabelle die Aspekte *header* und *footer* mitgeben, um eine Tabellenüberschrift bzw. Fußnote zu erzeugen. Die Aspektinhalte können einfache Texte oder auch beliebige Kombinationen von JSF-Tags sein.

```
<h:panelGrid …>
  <f:facet name="header">
    <h:outputText value="#{bean.headerText}" />
  </f:facet>
  <f:facet name="footer">
    Footer-Text
  </f:facet>
…
```

Listing 5.11: Aspekte für Header und Footer Tabelle 5.7 fasst die Tags der Core-Bibliothek zusammen. Ihr Einsatz wird später an den entsprechenden Stellen erläutert.

Tag	Bedeutung	s. Abschnitt
f:actionListener, f:event, f:phaseListener, f:setPropertyActionListener, f:valueChangeListener	Registrierung von Listenern für diverse Events	5.15
f:convertDateTime, f:convertNumber, f:converter	Konvertierer angeben	5.16
f:attribute	Attribute zu umgebendem Tag hinzufügen	
f:ajax	Ajax-Funktionalität hinzufügen	5.19
f:facet	Aspekt hinzufügen	5.6.1, 5.11
f:metadata, f:viewParam	Metadaten und Parameter für die View definieren	5.14
f:loadBundle	Resource Bundle laden (besser: Resource Bundle global deklarieren)	5.12
f:param	Parameter hinzufügen	
f:selectItem, f:selectItems	Selektionswerte bestimmen	5.8
f:subview	Neuen ID-Namensraum definieren	
f:validateDoubleRange, f:validateLength, f:validateLongRange, f:validateBean, f:validateRegex, f:validateRequired, f:validator	Validatoren angeben (besser: Bean Validation einsetzen)	5.17
f:verbatim	HTML- und JSP-Text zu JSF-Komponente zusammenfassen (wird in Facelets nicht benötigt)	
f:view	View definieren (wird in Facelets nicht benötigt)	5.6.1

Tabelle 5.7: Tags der Core-Bibliothek

5.7 Managed Beans

Die bisherigen Einstiegsbeispiele waren eher statischer Natur. Um dort Dynamik hineinzubringen, benötigen wir Daten und Logik, die wir über sog. Managed Beans mit den Views verknüpfen. Den Managed Beans fällt somit eine entscheidende Rolle für die klare Trennung von Präsentation und Geschäftslogik zu. Man kann für gut strukturierte Software sogar noch weiter gehen und Seitenbeschreibung, Präsentationslogik und Geschäftslogik voneinander trennen:

- Die Seitenbeschreibung deklariert den Seitenaufbau, komponiert eine View also aus ihren Einzelkomponenten. Dazu stehen uns Facelets zur Verfügung.
- Die Geschäftslogik umfasst Abläufe, Persistenz, Berechnungen – eben alles, was eine Anwendung unabhängig von ihrer konkreten Oberfläche tut. Hierfür stehen uns z. B. CDI und Java Persistence zur Verfügung.
- Managed Beans im Sinne von JSF verknüpfen diese beiden Anwendungsseiten miteinander und implementieren die ggf. noch fehlende Präsentationslogik, d. h. sie bereiten Daten aus der Geschäftslogik für die Präsentationskomponenten auf oder entscheiden aufgrund von Aufrufen der Geschäftslogik über die Navigation innerhalb der Seiten der Anwendung.

Zur Bereitstellung von Managed Beans gibt es durch die Historie der Java-EE-Plattform bedingt mehrere Möglichkeiten: Zum einen hat JSF eigene Managed Beans, die mithilfe der Annotation *@ManagedBean*[7] oder gleichwertiger Einträge im Konfigurations-File *faces-config.xml* definiert werden. Zum anderen können CDI Beans mit der Annotation *@Named* diese Rolle übernehmen. Die Überschneidung ist nahezu hundertprozentig. Vermutlich wird die zweite Variante künftig die einzig relevante sein. Im Folgenden werden daher ausschließlich CDI Beans verwendet.

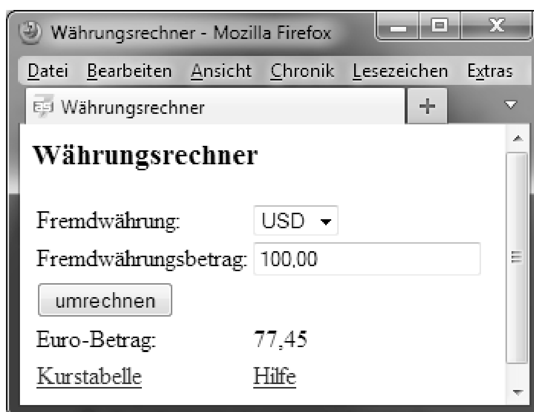


Abbildung 5.5: Währungsrechner

7 *javax.faces.bean.ManagedBean*

Angenommen, wir wollten den in Abbildung 5.5 gezeigten Dialog als Webanwendung realisieren. Die HTML-Bibliothek bietet uns dazu einige Eingabekomponenten an (Tabelle 5.4), von denen wir hier zwei verwenden können: *h:selectOneMenu* für die Währungsauswahl und *h:inputText* für die Eingabe des Betrags.

Eingabeelemente erhalten ihren aktuellen Wert während der Rendering-Phase, indem eine Property einer Managed Bean gelesen wird. Umgekehrt wird der eingegebene Wert während der Phase 4 (*Update Model Values*) in die Property gespeichert. Der Begriff Property entstammt der JavaBeans-Spezifikation und bezeichnet ein Methodenpaar: Eine Getter-Methode zum Lesen und eine Setter-Methode zum Setzen des Werts. Diese Methoden haben die vorgegebenen Namen *getName* und *setName*, wobei der Name der Property *name* ist. Unsere Währungsrechner-View wird später auf die Property *fremdWaehrungsBetrag* zugreifen, d. h. in Phase 4 wird *setFremdWaehrungsBetrag* aufgerufen und in Phase 6 *getFremdWaehrungsBetrag*.

Analog benötigen wir für die Währungsauswahl und den Eurobetrag die Properties *fremdWaehrungsKuerzel* und *euroBetrag*. Im letzten Fall können wir auf die Setter-Methode verzichten, da der Betrag nur ausgegeben wird.

Für den Button – eines der Aktionselemente aus der HTML-Bibliothek – gilt eine ähnliche Überlegung: Hier benötigen wir eine Methode, die in Phase 5 (Invoke Application) aufgerufen wird, wenn der Button betätigt wurde.

Für die Managed Bean des Währungsrechners ergibt sich somit die in Listing 5.12 gezeigte Klasse.

```
@Named @SessionScoped
public class WaehrungsRechnerModel implements Serializable
{
  private String fremdWaehrungsKuerzel;
  private double fremdWaehrungsBetrag;
  private double euroBetrag;

  public String getFremdWaehrungsKuerzel()
  {
    return this.fremdWaehrungsKuerzel;
  }

  public void setFremdWaehrungsKuerzel(String fremdWaehrungsKuerzel)
  {
    this.fremdWaehrungsKuerzel = fremdWaehrungsKuerzel;
  }

  // weitere Getter und Setter
  // …

  public void umrechnen()
```

```
  {
    …
  }
…
```

Listing 5.12: Managed Bean mit Properties und Aktionsmethode

Einen Fallstrick stellen die Scope-Annotationen dar – wir haben sie im Kapitel über CDI bereits kennen gelernt. Leider gibt es im Paket *javax.faces.bean* gleichnamige Annotationen. Achten Sie daher bei Verwendung von CDI Beans darauf, dass sie die CDI-Annotationen aus *javax.enterprise.context* verwenden.

Die Seitenbeschreibung für den Eingabeteil zeigt Listing 5.1. Nur angedeutet sind darin allerdings die Verknüpfungen zwischen View und Managed Bean, die mithilfe der Unified Expression Language realisiert werden.

```
<html …>
…
<h:body>
  …
  <h:form>
    <h:panelGrid columns="2">
      <h:outputLabel for="fremdWaehrung" value="Fremdwährung: " />
      <h:selectOneMenu id="fremdWaehrung" value="#{…}">
        <f:selectItems value="#{…}" …"/>
      </h:selectOneMenu>

      <h:outputLabel for="fremdBetrag" value="Fremdwährungsbetrag: " />
      <h:inputText id="fremdBetrag" value="#{…}"/>

      <h:commandButton value="umrechnen" action="#{…}" />
      <h:panelGroup />

      <h:outputLabel for="euroBetrag" value="Euro-Betrag: " />
      <h:outputText id="euroBetrag" value="#{…}"/>
…
```

Listing 5.13: Eingabeteil der Währungsrechner-View

5.8 Unified Expression Language

Die JSF Expression Language ist das Bindeglied zwischen der textbasierten Seitenbeschreibung und der objektorientierten Welt der Managed Beans. Mit ihr ist es möglich, die Werte von UI-Komponenten mit Properties von Managed Beans zu verknüpfen oder auch Aktionselemente mit den Methoden zu verbinden, die bei ihrer Betätigung aufgerufen werden sollen.

Die grundlegende Syntax von JSF-EL-Ausrücken ist *#{expression}*. Als *expression* können darin eine Wertebindung, eine Methodenbindung oder eine arithmetischer Ausdruck stehen.

Wenn hier von JSF-EL gesprochen wird, ist das eigentlich nicht mehr ganz richtig: Früher definierten sowohl die JSP-Spezifikation als auch JSF getrennte Expression Languages. JSP-EL-Ausdrücke folgen dabei dem Format *${expression}*, sind also syntaktisch bis auf das Startzeichen identisch mit JSF-EL-Ausdrücken. Der Unterschied lag darin, dass JSP-EL-Ausdrücke an jeder Stelle einer JSP genutzt werden konnten und schon beim Seitenaufbau ausgewertet wurden. JSF-EL-Ausdrücke sind dagegen nur innerhalb der JSF-Tags erlaubt und werden in den verschiedenen Phasen der Request-Verarbeitung ausgewertet.

Mittlerweile (seit JSP 2.1 und JSF 1.1) hat man diese beiden Sprachen aber vereinigt zur Unified Expression Language. Das einleitende Zeichen (# oder $) kann nun frei gewählt werden, und die Platzierung innerhalb der Seitenbeschreibung bestimmt den Auswertezeitpunkt.

Methodenbindung

Um eine Aktionskomponente wie z. B. einen Button mit der nach Betätigung auszuführenden Methode zu verknüpfen, bedient man sich eines EL-Ausdrucks zur Methodenbindung mit der Syntax *#{bean.method}*. *bean* verweist darin auf eine Managed Bean mit dem entsprechenden Bean-Namen. Im CDI-Kapitel wurde beschrieben, wie einer CDI Bean mithilfe der Annotation *@Named* ein Name zugewiesen wird.

method bezeichnet die Methode innerhalb der Bean. Sie muss die Signatur *public void method()* oder *public Object method()* haben. Es dürfen auch Parameter übergeben werden. Der EL-Ausdruck ist dann *#{bean.method(parameter)}* und die Methode muss eine passende Parameterliste aufweisen.

Um den Button des Währungsrechners mit der Methode *WaehrungsRechnerModel.umrechnen* zu verbinden, muss das Attribut *action* des Buttons den Wert *#{waehrungsRechnerModel.umrechnen}* haben (Listing 5.14). Betätigt der Benutzer den Button zur Laufzeit, wird damit einerseits ein Request ausgelöst. Durch die Methodenbindung kommt es dann in Phase 5 zum Aufruf der gewünschten Methode.

```
<h:commandButton value="umrechnen"
               action="#{waehrungsRechnerModel.umrechnen}" />
```

Listing 5.14: Listing 5.14: Bindung des Buttons an die aufzurufende Bean-Methode

Mit dem Attribut *actionListener* und mit den Unterelementen *f:actionListener* lassen sich weitere Methoden registrieren, die ebenfalls aufgrund der Betätigung des Buttons aufgerufen werden. Darauf geht der Abschnitt über die Event-Behandlung weiter unten ein.

Wertebindung

Möchte man eine UI-Komponente mit einem Wert in einer Managed Bean verbinden, kommt ein EL-Ausdruck in Form einer Wertebindung zum Einsatz. Er folgt der allgemeinen Syntax *#{bean.property}*. Darin verweist *bean* auf eine Managed Bean mit dem entsprechenden Bean-Namen, *property* ist der Name einer Property innerhalb der referenzierten Bean.

Im Währungsrechner werden in dieser Art die beiden Währungsfelder mit den zugehörigen Properties aus *WaehrungsRechnerModel* verknüpft (Listing 5.15). Die Wertebindung ist bidirektional, d. h. sie wirkt sowohl lesend als auch schreibend: In Phase 6 wird die Getter-Methode der Property aufgerufen, um den Wert zu lesen. Bei einem Eingabeelement wird die Setter-Methode in Phase 4 aufgerufen, um den aktuellen Wert in die Bean Property zu speichern.

```
<h:inputText value="#{waehrungsRechnerModel.fremdWaehrungsBetrag}"/>
…
<h:outputText value="#{waehrungsRechnerModel.euroBetrag}">
```

Listing 5.15: Wertebindung für Ein- und Ausgabefelder

Das Auswahlelement für die Währung wird analog behandelt, allerdings wird hier noch eine Liste der zur Verfügung stehenden Werte benötigt. Dazu dient das Element *f:selectItems* im Body des Auswahlelements. Sein Attribut *value* bestimmt die Menge der Auswahlwerte. Über eine Wertebindung werden hier ein Array oder eine *Collection* von Objekten zugeordnet. Die entsprechende Property wird nur gelesen, d. h. es reicht aus, die Getter-Methode in der Bean bereitzustellen.

Die Auswahlelemente *h:selectXxx* unterscheiden zwischen dem *label*, das dem Benutzer angezeigt wird, und dem *value*, der durch die Auswahl schließlich eingegeben wird. Je nach Typ der mittels *f:selectItems* angelieferten Werte bestimmen sich *Label* und *Value* in unterschiedlicher Weise:

- Bei einem Array oder einer Collection von Werten des Typs SelectItem[8] bestimmen deren Attribute label und value, was angezeigt und eingegeben wird. SelectItem bietet passende Konstruktoren und Zugriffsmethoden zum Setzen der Attribute an. Listing 5.16 zeigt eine beispielhafte Property, in der eine solche Auswahlliste mit numerischen Eingabewerten und Beschriftungstexten erstellt wird.

```
public List<SelectItem> getNotenListe()
{
  List<SelectItem> notenList = new ArrayList<>();
  notenList.add(new SelectItem(1, "sehr gut"));
  notenList.add(new SelectItem(2, "gut"));
  notenList.add(new SelectItem(3, "befriedigend"));
```

8 *javax.faces.model.SelectItem*

entwickler.press

```
  notenList.add(new SelectItem(4, "unbefriedigend"));
  return notenList;
}
```

Listing 5.16: Mithilfe von „SelectItem"-Objekten aufgebaute Auswahlliste

- Bei einem Array oder einer *Collection* von anderen Java-Objekten entsprechen Anzeige- und Eingabewerte der String-Repräsentation der Objekte (d. h. *toString()*). *f:selectItems* kann allerdings mit weiteren Parametern versehen werden, um *label* und *value* explizit zu bestimmen. Dazu wird mit *var* ein nur innerhalb des Tags gültiger Variablenname deklariert und mit *itemLabel* und *itemValue* jeweils eine Bindung an eine Property in Bezug auf diese Variable angegeben. Die Variable wirkt ähnlich einer Schleifenvariablen, mit der bei jedem Schleifendurchlauf ein *label* und ein *value* für die Auswahl ermittelt werden.

Im Währungsrechner steht die Methode *getWaehrungen* zur Verfügung, die alle bekannten Währungen in einer Liste liefert. Deren Elemente vom Typ *Waehrung* haben wiederum Getter-Methoden *getId* und *getEuroValue*, die das Währungskürzel (z. B. *CHF*) und den zugehörigen Eurowert liefern. Damit ist eine Parametrierung des Auswahlelements in der View wie in Listing 5.17 gezeigt möglich. *itemLabel* und *itemValue* werden hier beide aus der Währungs-ID gefüllt. Es wäre aber auch möglich gewesen, *itemValue="#{waehrung.euroValue}"* anzugeben, um in die Ziel-Property der Eingabe den Umrechnungsfaktor anstelle des Währungskürzels zu speichern.

```
<h:selectOneMenu
       value="#{waehrungsRechnerModel.fremdWaehrungsKuerzel}">
  <f:selectItems value="#{waehrungsRechnerModel.waehrungen}"
              var="waehrung"
              itemLabel="#{waehrung.id}"
              itemValue="#{waehrung.id}"/>
</h:selectOneMenu>
```

Listing 5.17: Auswahlelement mit expliziter Angabe der Anzeige- und Eingabewerte

Im Fall einer Property, die nur gelesen wird, kann statt einer Wertebindung auch ein Methodenaufruf im EL-Ausdruck genutzt werden. Im Beispiel oben wäre somit statt *<h:outputText value="#{waehrungsRechnerModel.euroBetrag}">* auch *<h:outputText value="#{waehrungsRechnerModel.getEuroBetrag()}">* möglich gewesen. Diese Variante ermöglicht es, auf Methoden zuzugreifen, die nicht dem Namensschema von Properties folgen, z. B. *value="#{bean.list.size()}"* oder *rendered="#{bean.text.contains('abc')}"*.

Vordefinierte Variablen Die Unified Expression Language definiert einige Variablen vor (Tabelle 5.7). Sie können in EL-Ausdrücken wie Bean-Namen verwendet werden, werden allerdings nicht häufig benötigt.

Variable	Type	Bedeutung
initParam	*Map*	Initialisierungswerte (Context-Parameter der Webanwendung)
facesContext	*FacesContext*[1]	Statusinformationen der aktuellen Request-Verarbeitung
requestScope *viewScope* *sessionScope* *applicationScope*	*Map<String, Object>*	Scope-Objekte
view	*UIViewRoot*[2]	Aktuelle View
header *headerValues*	*Map<String, String>* *Map<String, String[]>*	Header-Werte des Requests als Strings bzw. String[]
param *paramValues*	*Map<String, String>* *Map<String, String[]>*	Parameterwerte des Requests als Strings bzw. String[]
cookie	*Map<String, Object>*	Cookies des Requests

Tabelle 5.8: Vordefinierte EL-Variablen

Arithmetische Ausdrücke

Für EL-Ausdrücke sind einige Operatoren definiert, mit denen sich arithmetische Ausdrücke kombinieren lassen (Tabelle 5.8).

Bei ihrem Einsatz sollten Sie allerdings Vorsicht walten lassen, da Berechnungen in den allermeisten Fällen Teil der Geschäftslogik sind und somit in einer View fehlplatziert wären. Bedenklich in diesem Sinne sind sämtliche Beispiele der Tabelle mit Ausnahme von *rendered="#{empty bean.list}"* und ggf. auch *rendered="#{bean.number != 0}".*

Für den Zugriff auf Map-Elemente gibt es noch eine weitere Möglichkeit: *#{(bean.map.key}* ist äquivalent zu *#{(bean.map['key']}.*

Operator	Bedeutung	Beispiel
*+, -, *, /, %*	Grundrechenarten	*value="#{bean.number * 1.19}"*
==, !=, <, <=, >, >= *eq, ne, lt, le, gt, ge*	Vergleiche	*rendered="#{bean.number != 0}"*
empty	Test auf Leerheit	*rendered="#{empty bean.list}"*
?:	Bedingter Ausdruck	*value="#{(bean.number%2)==0 ? 'even' : 'odd'}"*
&&, ||, ! *and, or, not*	Logische Operatoren	*rendered="#{bean.number1 != 0* *&& bean.number2 > 100}"*
()	Klammerung	*value="#{(bean.number + 1) * 2}"*
[]	Array-, Listen-, Map-Zugriff	*value="#{(bean.array[2]}"* *value="#{(bean.list[5]}"* *value="#{(bean.map['key']}"*

Tabelle 5.9: EL-Operatoren

5.9 Navigation

Anwendungen besitzen in den meisten Fällen nicht nur eine View. Übergänge zwischen den Anzeigeseiten sind immer dann möglich, wenn ein Request ausgelöst wird, wenn der Benutzer also eines der Aktionselemente bedient.

Regelbasierte Navigation

Ein wichtiger Parameter zur Festlegung der nächsten View ist das sog. Outcome der auslösenden Aktion. Darunter versteht man einen Text, der als Ergebnis der aufgerufenen Aktionsmethode geliefert oder auch direkt im Attribut *action* des Aktionselements angegeben wird (Listing 5.18). Das Outcome ist implizit *null*, wenn die Aktionsmethode kein Ergebnis liefert (*void*-Methode) bzw. das Attribut *action* fehlt.

```
<h:form>
  <h:commandButton value="ok" action="#{someBean.doOk}" />
  <h:commandLink value="Hilfe" action="help" />
</h:form>

@Named
public class SomeBean
{
  public String doOk()
  {
    return "goOn";
  }
  …
```

Listing 5.18: Outcome als Aktionsmethodenergebnis oder direkte Angabe

Vor der Rendering-Phase wird festgelegt, welche View als Nächstes anzuzeigen ist. Dazu werden bei der regelbasierten Navigation Regeln der Form „Falls auf Seite *x* das Outcome *a* erzeugt wird, geht es auf Seite *y* weiter“ verwendet. Diese befinden sich in der Konfigurationsdatei *WEB-INF/faces-config.xml* in *navigation-rule*-Elementen (Listing 5.19).

```
<faces-config … >
  <navigation-rule>
    <from-view-id>/pages/waehrungsRechner*</from-view-id>
    <navigation-case>
      <from-outcome>help</from-outcome>
      <to-view-id>/pages/waehrungsRechner_help.xhtml</to-view-id>
    </navigation-case>
    …
```

Listing 5.19: Navigation Rule

Jede View der Anwendung besitzt eine View ID, die dem Pfad der Seitenbeschreibung innerhalb der Anwendung entspricht.

Jedes dieser Elemente fasst die Regeln für eine oder mehrere Ausgangsseiten zusammen, wobei das Element *from-view-id* bestimmt, für welche. Hier kann entweder eine exakte View ID angegeben werden – z. B. */pages/waehrungsRechner.xhtml* – oder wie im Beispiel ein Präfix der gewünschten View IDs, gefolgt von einem *. Ein *navigation-rule* ohne *from-view-id* oder mit *<from-view-id>*</from-view-id>* passt auf alle Views der Anwendung.

Die Regel kann beliebig viele *navigation-case*-Elemente enthalten, die jeweils ein Outcome mit einer Zielseite verknüpfen. Das Beispiel in Listing 5.18 liest sich also so: Falls auf einer der Währungsrechner-Seiten das Outcome *help* erzeugt wird, ist die nächste View */pages/waehrungsRechner_help.xhtml*.

Ein *navigation-case* kann durch weitere Elemente ergänzt werden:

- *from-action:* Angabe einer Aktionsmethode in Form einer EL-Methodenbindung. Die Regel gilt dann nur, wenn das Outcome von der genannten Methode erzeugt wurde. Bei Angabe von *from-action* kann *from-outcome* sogar entfallen, wenn die Regel für alle Outcomes der genannten Methode gelten soll.
- *if:* Angabe einer Bedingung in Form eines Boole'schen EL-Ausdrucks. Die Regel gilt dann nur, wenn der Audruck *true* ergibt.
- *redirect:* Wird dieses Element (ohne Inhalt) angegeben, geschieht der Übergang zur nächsten Seite durch Redirect, d. h. der Browser wird angewiesen, die nächste Seite durch einen neuen Request anzufordern. Ohne *redirect* wird ein serverseitiges Forward durchgeführt, wodurch kein neuer Request benötigt wird, der im Browser angezeigte URL allerdings um einen Zyklus hinterherhinkt.

Durch die Möglichkeit, die Ausgangs-View mit einem Präfix bestimmen und in den *navigation-case*s unterschiedliche Bedingungen angeben zu können, kann es zu Mehrdeutigkeiten kommen, d. h. auf eine Ausgangssituation passen ggf. mehrere Regeln. In der JSF-Spezifikation ist exakt beschrieben, welche Regel dann angewendet wird. Grob kann man das so zusammenfassen: Die Regel mit der genauesten Angabe gewinnt.

Gibt es keine passende Regel, wird die aktuelle Seite erneut angezeigt.

Inline-Navigation

Es ist möglich, auf die Navigationsregeln in faces-config.xml zu verzichten. Dann müssen anstelle von Outcomes direkt View IDs verwendet werden – entweder im Attribut action der Aktionselemente oder als Ergebnis der Aktionsmethoden (Listing 5.20). Mit dem Zusatz ?faces-redirect=true kann wiederum ein Redirect angefordert werden.

```
<h:commandButton value="tue was" action="#{someBean.doSomething}" />
<h:commandLink value="Kurstabelle"
               action="/pages/waehrungsRechner_rates.xhtml" />

@Named
public class SomeBean
{
```

```
  public String doSomething()
  {
    return "/somePage.xhtml?faces-redirect=true";
  }
  …
```

Listing 5.20: Inline-Navigation

Inline-Navigation führt schnell zu unübersichtlichem und schlecht wartbarem „Spaghetticode“. Setzen Sie sie daher nur mit Bedacht ein!

Programmgesteuerte Navigation

Manchmal ist es wünschenswert, in einer Methode der Anwendung eine Navigation auszulösen, z. B. um in einer der weiter unten gezeigten Listener-Methoden auf besondere Situationen mit einer Umschaltung auf eine neue Seite zu reagieren. Für solche Fälle kann der *NavigationHandler* der JSF-Implementierung direkt aufgerufen werden. Die Methode *handleNavigation* erhält als zweiten und dritten Parameter die Werte, die den oben angesprochenen Elementen *from-action* und *from-outcome* entsprechen (Listing 5.21).

```
if (this.fremdWaehrungsBetrag < 0)
{
  FacesContext facesContext = FacesContext.getCurrentInstance();
  NavigationHandler navigationHandler
    = facesContext.getApplication().getNavigationHandler();
  navigationHandler.handleNavigation(facesContext, null, "help");
}
```

Listing 5.21: Aufruf des Navigation Handlers

5.10 Scopes

Die Lebensdauer der Managed Beans wird über ihren Scope festgelegt. Da in diesem Buch CDI Beans als Managed Beans genutzt werden, stehen die im CDI-Kapitel beschriebenen Scopes zur Verfügung. Tabelle 5.9 fasst das dort Gesagte nochmals zusammen.

Scope-Annotation	Bean-Lebensdauer
@RequestScoped	Ein Request
@Conversation-Scoped	Bei transienter Konversation wie *@RequestScoped*, sonst bis zum Ende der Konversation durch explizite Terminierung oder durch Timeout
@SessionScoped	Vom ersten Zugriff bis zum Ende der Sitzung durch explizite Terminierung oder durch Timeout
@ApplicationScoped	Laufzeit der Anwendung

Tabelle 5.10: Scopes

Da JSF auch eigene Scope-Annotationen im Paket *javax.faces.bean* enthält, müssen Sie darauf achten, diejenigen aus *javax.enterprise.context* zu verwenden.

Es sei nicht verschwiegen, dass die Überdeckung zwischen JSF und CDI in Bezug auf die Scopes nicht vollständig ist: JSF kennt neben den genannten Scopes noch View- und Flash-Scopes. Zudem können neue Scopes auf recht einfache Weise mit *@CustomScoped* definiert werden. Die Lücke wird allerdings durch diverse CDI Extensions geschlossen – z. B. durch die im CDI-Kapitel erwähnten Ergänzungen Apache MyFaces CODI oder JBoss Seam.

5.11 Verarbeitung tabellarischer Daten

Viele Anwendungen verarbeiten Daten, die übersichtlich in Form einer Tabelle dargestellt werden können. Das Tag *h:panelGrid* erzeugt zwar eine HTML-Tabelle, aber deren Dimensionen werden in der Seitenbeschreibung festgelegt und nicht dynamisch entsprechend dem Umfang der anzuzeigenden Daten.

Hier kommt *h:dataTable* zum Einsatz. Dieses Tag verarbeitet ein Array oder eine Liste von Objekten zur Anzeige einer entsprechend langen Tabelle. Sein Attribut *var* definiert einen nur innerhalb des Tags gültigen Namen für eine Variable, die wie eine Schleifenvariable arbeitet: *h:dataTable* iteriert über die als *value* übergebenen Werte. Für jeden Eintrag wird der Inhalt des Tags zur Anzeige gebracht, wobei die Variable den aktuellen Wert enthält.

Die Unterelemente *h:column* beschreiben je eine Spalte der Tabelle. Die darin befindlichen Ein- und Ausgabeelemente können die Iterationsvariable in ihren EL-Ausdrücken verwenden, um auf eine Property des jeweils aktuellen Eintrags zuzugreifen.

Listing 5.22 zeigt eine beispielhafte Anwendung: Das Element *h:dataTable* iteriert mithilfe der lokalen Variablen *bank* über eine Liste von *Bank*-Objekten. Diese haben u. a. die Properties *blz* und *name*, die in den beiden *h:column*-Elementen ausgegeben werden.

```
<h:dataTable var="bank" value="#{bankModel.searchResult}" >
  <h:column>
    <h:outputText value="#{bank.blz}" />
  </h:column>
  <h:column>
    <h:outputText value="#{bank.name}" />
  </h:column>
  …
</h:dataTable>

@Named
public class BankModel
{
  public List<Bank> getSearchResult() { … }
  …
}
```

```
public class Bank
{
  public String getBlz() { … }
  public void setBlz(String blz) { … }

  public String getName() { … }
  public void setName(String name) { … }
  …
}
```

Listing 5.22: Anzeige tabellarischer Daten

h:dataTable erlaubt mit seinen Attributen *rowClasses* und *columnClasses* die Angabe von Stilinformationen für die Zeilen und Spalten der Tabelle. Es kann jeweils eine kommagetrennte Liste von CSS-Stilen angegeben werden, die für die Zeilen bzw. Spalten der Reihe nach verwendet werden. So kann z. B. durch Angabe von zwei verschiedenen Zeilenstilen ein alternierender Zeilenhintergrund erzeugt werden, um das Lesen der Tabelle zu erleichtern (Listing 5.23, Abb. 5.6).

```
<h:dataTable var="bank" value="#{bankModel.searchResult}"
             rowClasses="standardTable_Row1,standardTable_Row2" >
```

Listing 5.23: Angabe alternierender Zeilenstile

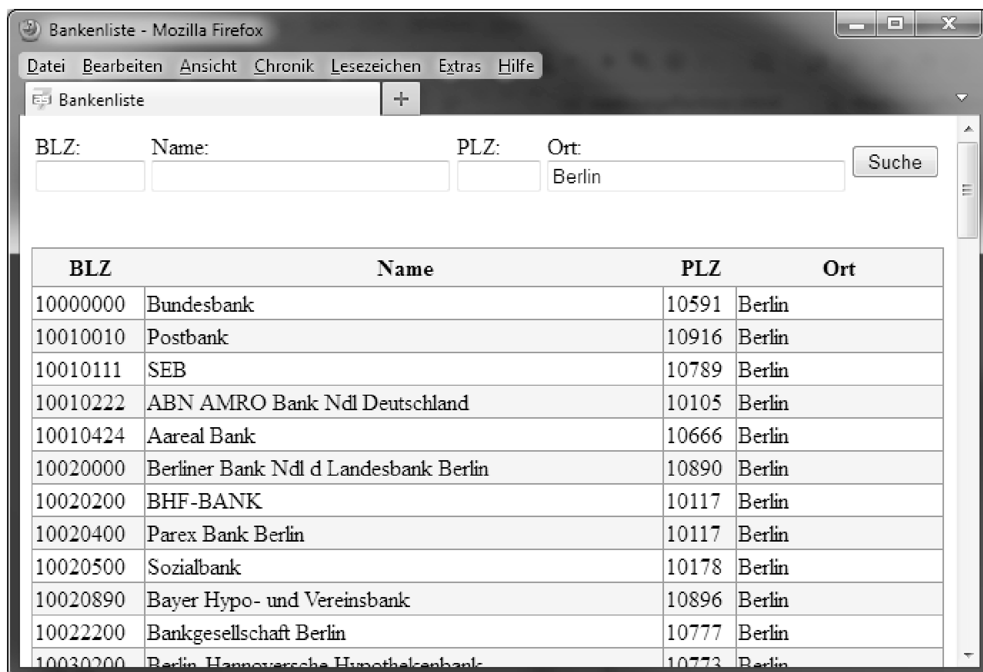


Abbildung 5.6: Beispiel für eine Datentabelle

Die ausgegebene Tabelle kann mit *f:facet*-Elementen um Header und Footer ergänzt werden. Das kann wie bei *h:panelGrid* für die Gesamttabelle geschehen oder innerhalb der *h:column*-Elemente für jede Spalte getrennt (Listing 5.24).

```
<h:dataTable …>
  <f:facet name="footer">Quelle: Deutsche Bundesbank</f:facet>
  <h:column>
    <f:facet name="header">BLZ</f:facet>
    <h:outputText value="#{bank.blz}" />
  </h:column>
…
```

Listing 5.24: Datentabelle mit Header und Footer

Die Inhalte der h:column-Elemente können beliebige JSF Tags sein, inkl. Eingabe- oder Aktionselementen. Damit ist es z. B. problemlos möglich, in die Tabellenzeilen einen Button aufzunehmen, der zu einer Bearbeitung des aktuellen Werts navigiert (Listing 5.25).

```
<h:dataTable var="bank" …>
  <h:column>
    <h:commandButton value="bearbeiten"
                     action="#{bankModel.edit(bank)}" />
    </h:column>
…
</h:dataTable>

@Named
public class BankModel
{
  public String edit(Bank bank) { … }
  …
}
```

Listing 5.25: Datentabelle mit Aktionselement

5.12 Internationalisierung

Viele Tags lassen sich lokalisieren, d. h. den Gepflogenheiten einer Sprache und eines Landes anpassen. Das umfasst Formate von Zahlen oder Datumsangaben und natürlich Texte, die von einer internationalisierten Anwendung in mehreren Übersetzungen vorgehalten werden müssen.

Locale

Der virtuelle Schauplatz einer Anwendung wird durch ein Objekt des Typs *Locale*[9] angegeben. Es kann vielen Tags mithilfe des Attributs *locale* mitgegeben werden. Dabei können ein *Locale*-Objekt oder auch eine übliche *String*-Repräsentation der Form *ll*, *ll_CC* oder *ll_CC_VV* übergeben werden. Darin wird die Sprache mit zwei Kleinbuchstaben *ll* angegeben, optional ein Land mit zwei Großbuchstaben *CC* und ggf. zusätzlich eine Variante VV (Tabelle 5.11).

de	Deutsch	en	Englisch
de_AT	Deutsch (Österreich)	en_AU	Englisch (Australien)
de_CH	Deutsch (Schweiz)	en_GB	Englisch (Großbritannien)
de_DE	Deutsch (Deutschland)	en_IN	Englisch (Indien)
de_LU	Deutsch (Luxemburg)	en_US	Englisch (USA)

Tabelle 5.11: Auszug aus den im Java-Standard verfügbaren Locales

Tags ohne explizite *Locale*-Angabe übernehmen die Lokalisierung von ihren umschließenden Tags. Ist auch dort keine Angabe vorhanden, wird die *Locale* aus dem Request ermittelt. Der Browser übermittelt dazu im Header *Accept-Language* eine Liste von akzeptierten *Locale*s. Welche das sind, lässt sich in der Konfiguration des Browsers einstellen.

Für die Anwendung lässt sich umgekehrt in der Konfigurationsdatei *faces-config.xml* einstellen, welche *Locale*s sie bedienen kann. Das geschieht mit dem Element *locale-config*, in dem eine Vorgabe-*Locale* eingetragen werden kann und eine Liste aller unterstützter *Locale*s (Listing 5.26).

```
<faces-config …>
  <application>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>de_CH</supported-locale>
      <supported-locale>de</supported-locale>
      <supported-locale>en</supported-locale>
```

Listing 5.26: Konfiguration der unterstützten „Locales“

Zwischen diesen beiden Angaben wird nach der besten Übereinstimmung gesucht: Die vom Browser übermittelten Locales werden in der angegebenen Reihenfolge mit denen der Anwendung verglichen. Dabei wird bei Bedarf jeweils auch ein Fallback innerhalb der Familie gemacht, d. h. die Wunsch-Locale *en_US* passt zur angebotenen Locale *en*.

9 *java.util.Locale*

Resource Bundles

Internationalisierte Texte werden üblicherweise in Form von Properties zur Verfügung gestellt, d. h. die Anwendung referenziert die Texte über logische Schlüssel, deren zugehörige Werte die anzuzeigenden Texte darstellen. Für jede Sprache wird ein Satz von Properties bereitgestellt, die die gleichen Schlüssel verwenden. Zur Laufzeit wird der zur aktuellen Sprache passende Satz von Properties ausgewählt.

Dieses Verfahren wird von der Java-Standard-Klasse *ResourceBundle* implementiert. Eine Möglichkeit der Ablage der Texte sind Properties-Dateien im Classpath: Eine Datei namens *basename.properties* definiert das Resource Bundle und enthält i. d. R. alle benötigten Texte als Default-Werte. Weitere Dateien namens *basename_locale.properties* enthalten die Übersetzungen für eine bestimmte *Locale*. Abbildung 5.7 zeigt eine solche Struktur. Das oberste Verzeichnis liegt im Classpath.

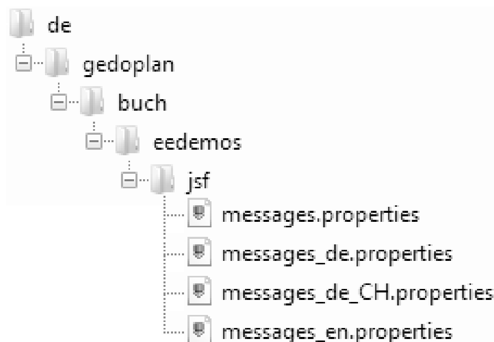


Abbildung 5.7: Properties-Dateien eines Resource Bundles

Die Dateien spannen durch ihre Zuordnung zu Locales eine Art Suchbaum auf. Wird ein Wert in der Datei der aktuellen Locale nicht gefunden, wird die Suche im nächsthöheren Knoten des Baums fortgesetzt, z. B.: *messages_de_CH.properties* –> *messages_de.properties* –> *messages.properties*.

Resource Bundles können in *faces-config.xml* global deklariert werden. Dabei wird ein Variablenname frei gewählt, der in JSF-EL-Ausdrücken verwendet werden kann, um auf die lokalisierten Texte zuzugreifen (Listing 5.27).

```
<faces-config …>
  <application>
    <resource-bundle>
      <base-name>de.gedoplan.buch.eedemos.jsf.messages</base-name>
      <var>messages</var>
    </resource-bundle>
…
```

Listing 5.27: Deklaration der Variablen „messages“ für ein Resource Bundle

Durch diese zentrale Deklaration kann in jeder View der Anwendung mit #*{messages.key}* auf den lokalisierten Text mit dem Schlüssel *key* zugegriffen werden.

Alternativ zur globalen Deklaration kann ein Resource Bundle in einer View explizit geladen werden. Dazu dient das Tag *f:loadBundle*, das bereits vor JSF 2 vorhanden war. Es sollte in neuen Anwendungen allerdings nicht mehr verwendet werden.

Ein besonderes Resource Bundle ist das sog. Application Message Bundle. In ihm befinden sich die Meldungen des JSF-Systems, die beispielsweise bei Validierungsfehlern generiert werden. Es ist Bestandteil der JSF-Implementierung, kann aber mit einer Deklaration in *faces-config.xml* durch ein eigenes Resource Bundle ersetzt werden (Listing 5.28).

```
<faces-config …>
  <application>
    <message-bundle>
      de.gedoplan.buch.eedemos.jsf.messages
    </message-bundle>
…
```

Listing 5.28: Deklaration eines eigenen Application Message Bundles

Programmgesteuerter Zugriff auf Texte

Manchmal benötigt man innerhalb einer Managed Bean Zugriff auf die lokalisierten Texte. Neben der Standardmöglichkeit mithilfe von *ResourceBundle.getBundle* gibt es seit JSF 2 die in Listing 5.29 gezeigte Möglichkeit, auf ein Resource Bundle über seinen in *faces-config.xml* eingetragenen Variablennamen zuzugreifen.

```
FacesContext ctx = FacesContext.getCurrentInstance();
ResourceBundle bundle
  = ctx.getApplication().getResourceBundle(ctx, "messages");
String text = bundle.getString("key");
```

Listing 5.29: Zugriff auf lokalisierte Texte im Code einer Managed Bean

5.13 Ressourcenverwaltung

Neben dem (X)HTML-Code werden zum Seitenaufbau häufig weitere Informationen benötigt: Bilder, CSS- oder JavaScript-Dateien. Diese sog. Ressourcen können zentral abgelegt werden, und zwar in Verzeichnissen namens resources entweder im Wurzelverzeichnis der Webanwendung oder im Verzeichnis *META-INF* in einem beliebigen Classpath-Verzeichnis (Abb. 5.).

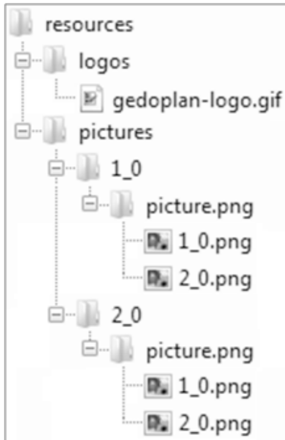


Abbildung 5.8: Ressourcenablage

Das Tag *h:graficImage* zur Ausgabe einer Grafik adressiert diese Ablagestruktur mit den folgenden Attributen:

- *library* ist optional und bezeichnet das Verzeichnis unterhalb von *resources*,
- *name* bezeichnet die Ressourcendatei selbst.

Zur Ausgabe von Stylesheets und Skripten dienen die Tags *h:outputStylesheet* und *h:outputScript*. Das Besondere an diesen Tags ist, dass sie keine Doppelausgabe erzeugen, auch wenn das Tag für eine Ressource mehrfach vorkommen sollte. *h:outputScript* kann zudem die Ausgabe an *h:head* oder *h:body* delegieren, und zwar gesteuert durch das Attribut *target*. Für *h:outputStylesheet* ist das Target *head* voreingestellt (Listing 5.30).

```
<!-- Beispiele fuer Ressourcenzugriffe -->
<h:graphicImage library="logos" name="gedoplan-logo.gif" />
<h:outputStylesheet name="ee-demos.css"/>
<h:outputScript name="someFunction.js" target="body"/>

<!–Alternative Nutzung von EL -->
<h:graphicImage value="#{resource['logos:gedoplan-logo.gif']}" />
```

Listing 5.30: Beispiele für die Ausgabe von Ressourcen

Alternativ zur direkten Adressierung der Ressourcendatei mittels *library* und *name* kann auch die vordefinierte EL-Variable *resource* verwendet werden. An dieser Stelle – und nicht nur hier – bemerkt man, dass JSF häufig mehrere Möglichkeiten zur Lösung einer Aufgabenstellung bereithält. Im Sinne der Einfachheit wäre nach meiner Meinung hier weniger mehr gewesen.

Ressourcen können versioniert werden, d. h. sowohl die Library-Verzeichnisse als auch die Ressourcendateien selbst können mit einer Versionsnummer angereichert werden. Ab-

bildung 5. zeigt eine solche erweiterte Ablagestruktur. Bei einem Zugriff mit den gezeigten Tags wird automatisch die höchste verfügbare Version der referenzierten Ressourcen verwendet, wenn keine Version explizit adressiert wird. Ressourcenversionierung findet sich sehr selten in Anwendungen. Bei Bedarf finden Sie Details dazu in der Spezifikation (Abschnitt 2.6.1.4, Libraries of localized and versioned Resources).

Internationalisierung von Ressourcen

Seit JSF 2 können auch Ressourcen internationalisiert werden. Dazu muss im Application Message Bundle die Property *javax.faces.resource.localePrefix* eingetragen werden. Ihr Wert gibt den Namen des Unterverzeichnisses von *resources* an, in dem die Ressourcendateien gesucht werden (Abb. 5.9).

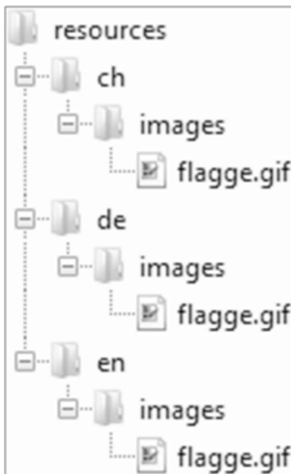


Abbildung 5.9: Internationalisierte Ressourcen

Wäre in die aktuelle Locale *de* und im deutschen Teil des Application Message Bundles *javax.faces.resource.localePrefix=de* eingetragen, so würde *<h:graphicImage library="images" name="flagge.gif" />* die deutsche Flagge rendern.

5.14 GET Support

In JSF geschieht die Verarbeitung von Eingabeparametern normalerweise in POST-Requests. Dadurch sind die im Browser sichtbaren URLs i. d. R. um einen Klick verzögert – wenn nicht mit Redirect gearbeitet wird – und nicht Bookmark-fähig.

Verarbeitung von GET-Request-Parametern

Seit JSF 2 können auch GET-Requests verarbeitet werden. Dazu muss am Anfang der View (als direktes Unterelement von *f:view* bzw. *html*) ein Element *f:metadata* eingefügt werden. Darin können beliebig viele GET-Parameter durch entsprechende *f:viewParam*-Elemente angenommen werden. Ihr *value*-Attribut bindet sie an eine Managed Bean Property (Listing 5.31).

```
<html …>
<f:metadata>
  <f:viewParam name="name" value="#{getSupportModel.name}" />
</f:metadata>
<h:head>
…
```

Listing 5.31: Verarbeitung von Parametern in GET-Requests

Im gezeigten Beispiel würde somit bei einem Request auf den URL *http://…/getSupport.xhtml?name=hugo* am Anfang der Request-Verarbeitung die Property *name* der referenzierten Bean mit dem Wert *hugo* belegt.

Erzeugung von GET-Requests

Die bekannten Elemente *h:commandButton* und *h:commandLink* erzeugen POST-Requests. Seit JSF 2 stehen zusätzlich *h:button* und *h:link* zur Verfügung. Sie benötigen kein *h:form*-Element und erzeugen GET-Requests.

Ihr Attribut *outcome* wird nach den bekannten Regeln schon in der Renderphase in einen entsprechende URL umgesetzt. Das Navigationsziel wird also präemptiv schon bei der Anzeige des jeweiligen Elements festgelegt, nicht erst bei der Verarbeitung des von ihm ausgelösten Requests. Request-Parameter können mithilfe von *f:param*-Elementen angefügt werden.

Listing 5.32 zeigt als Beispiel ein *h:link*-Element, das schon in der Render-Phase zu dem URL umgesetzt wird, der dem Outcome *getSupport2* entspricht. Zudem wird der Query-Parameter *name=otto* angefügt.

```
<h:link value="getSupport2" outcome="getSupport2">
  <f:param name="name" value="otto" />
</h:link>
```

Listing 5.32: Erzeugung eines Links für einen GET-Request

Da *h:button* und *h:link* GET-Requests erzeugen, sind die entsprechenden URLs Bookmark-fähig.

5.15 Event-Verarbeitung

JSF enthält ein System zur Event-Verarbeitung ähnlich dem, das man in vielen GUI-Frameworks wie z. B. Swing findet. Es gibt diverse Event-Quellen, die bei gewissen Zuständen oder Zustandsänderungen Event-Objekte erzeugen und an alle sog. Listener verteilen, die für den jeweiligen Event registriert wurden.

Faces Events

Einige UI-Komponenten lösen Events aus. In der Standardbibliothek sind dies die Aktionselemente, die *ActionEvents* versenden, und die Eingabekomponenten, die *ValueChangeEvents* auslösen. Die Auslieferung der Events an die Listener geschieht zu definierten Zeitpunkten bei der Request-Verarbeitung.

Action Listener können als Methode einer Managed Bean mit der Signatur *public void method(ActionEvent)*[10] definiert werden. Ihre Registrierung erfolgt durch Angabe einer Methodenbindung als Attribut *actionListener* des Aktionselements. Alternativ oder auch zusätzlich können Action Listener als Klassen definiert werden, die *ActionListener*[11] implementieren. Deren Registrierung erfolgt durch Angabe ihres Klassennamens in *f:actionListener*-Elementen (Listing 5.33).

```
<h:commandButton … actionListener="#{eventDemoModel.handleAction}">
  <f:actionListener
    type="de.gedoplan.….DemoActionListener" />
</h:commandButton>

@Named
public class EventDemoModel
{
  public void handleAction(ActionEvent event)
  {
    …
  }
  …
}

public class DemoActionListener implements ActionListener
{
  public void processAction(ActionEvent event)
  {
    …
  }
}
```

Listing 5.33: Registrierung von Action Listenern als Methode oder Klasse

10 *javax.faces.event.ActionEvent*
11 *javax.faces.event.ActionListener*

Der Aufruf der Methoden geschieht in Phase 5 (*Invoke Application*) vor der Ausführung der Aktionsmethode des auslösenden Aktionselements. Das übergebene Event-Objekt enthält dabei u. a. die Information, welches Aktionselement der Auslöser des Events war.

Value Change Listener werden analog als Bean-Methoden mit der Signatur *public void method(ValueChangeEvent)*[12] oder als Implementierung von *ValueChangeListener*[13] definiert und an einem Eingabeelement mit dem Attribut *valueChangeListener* bzw. mit *f:valueChangeListener*-Elementen registriert (Listing 3.34).

```
<h:inputText …
  valueChangeListener="#{eventDemoModel.handleValueChange}">
  <f:valueChangeListener
    type="de.gedoplan.….DemoValueChangeListener" />
</h:inputText>

@Named
public class EventDemoModel
{
  public void handleValueChange(ValueChangeEvent event)
  {
    …
  }
  …
}

public class DemoValueChangeListener implements ValueChangeListener
{
  public void processValueChange(ValueChangeEvent event)
  {
    …
  }
}
```

Listing 5.34: Registrierung von Value Change Listenern als Methode oder Klasse

Hier werden die Methoden am Ende von Phase 3 (*Process Validations*) aufgerufen, wenn der Wert des entsprechenden Elements im aktuellen Request verändert wird. Das übergebene Event-Objekt enthält neben der Event-Quelle auch den alten und neuen Wert. Denken Sie daran, dass zu diesem Zeitpunkt der Request-Verarbeitung die Properties in den zugeordneten Beans noch nicht gesetzt sind.

Phase Events

Bei Eintritt in eine Phase der Request-Bearbeitung und an ihrem Ende werden ebenfalls Events ausgelöst. Diese Phase Events sind weniger für die Entwicklung von Anwendungen interessant als für die Erweiterung des Frameworks. Die Bearbeitung geschieht mit

12 *javax.faces.event.ValueChangeEvent*
13 *javax.faces.event.ValueChangeListener*

Klassen, die *PhaseListener*[14] implementieren. Die darin enthaltenen Methoden *beforePhase* und *afterPhase* werden zu den beschriebenen Zeitpunkten aufgerufen, wobei das übergebene Event-Objekt u. a. Aufschluss über die aktuelle Phase gibt. Die Methode *getPhaseId* dient der Angabe der Phase, für die der Listener aufgerufen werden soll. Der Rückgabewert muss eine der in *PhaseId*[15] definierten Konstanten sein und meldet das Interesse an einer bestimmten Phase (z. B. *INVOKE_APPLICATION*) oder an allen Phasen (*ANY_PHASE*) an (Listing 3.35).

```
public class DemoPhaseListener implements PhaseListener
{
  public PhaseId getPhaseId()
  {
    return PhaseId.ANY_PHASE;
  }

  public void beforePhase(PhaseEvent event) { … }
  public void afterPhase(PhaseEvent event) { … }
}
```

Listing 5.35: Beispiel für einen Phase Listener

Die Registrierung eines Phase Listeners geschieht entweder in Bezug auf eine View durch das Element *f:phaseListener* (Listing 5.36) oder global für alle Views der Anwendung durch Eintrag in der Datei *faces-config.xml* (Listing 5.37).

```
<html …>
<f:phaseListener type="de.gedoplan.….listener.DemoPhaseListener"/>
…
```

Listing 5.36: Registrierung eines Phase Listeners für eine View

```
<faces-config …>
  <lifecycle>
    <phase-listener>de.gedoplan.….DemoPhaseListener</phase-listener>
  </lifecycle>
…
```

Listing 5.37: Registrierung eines globalen Phase Listeners

System Events

Mit JSF 2 sind Events hinzugekommen, die entweder die Teilnahme bestimmter Komponenten an einer Verarbeitungsphase anzeigen oder den allgemeinen Status des JSF-Systems melden.

14 *javax.faces.event.PhaseListener*
15 *javax.faces.event.PhaseId*

Mit *f:event* können UI-Komponenten Listener-Methoden zugeordnet werden. Das Attribut *type* wählt dabei den gewünschten Event aus: Die Werte *postAddToView*, *preValidate*, *postValidate* und *preRenderComponent* entsprechen den entsprechend benannten Event-Typen aus *javax.faces.event* und führen zum Aufruf der Methoden am Ende der Restore-Phase, zu Beginn und Ende der Validierungsphase bzw. zum Beginn der Render-Phase für die jeweilige Komponente (Listing 5.38).

```
<h:form>
  <f:event listener="#{eventDemoModel.handleComponentSystemEvent}"
           type="postAddToView" />
  <f:event listener="#{eventDemoModel.handleComponentSystemEvent}"
           type="preValidate" />
  <f:event listener="#{eventDemoModel.handleComponentSystemEvent}"
           type="postValidate" />
  <f:event listener="#{eventDemoModel.handleComponentSystemEvent}"
           type="preRenderComponent" />
</h:form>

@Named
public class EventDemoModel
{
  public void handleComponentSystemEvent(ComponentSystemEvent event)
  {
    …
  }
}
```

Listing 5.38: Registrierung von System Event Listenern für eine UI-Komponente

Weitere System Events können verwendet werden, um in den gesamten Lebenszyklus des JSF-Systems einzugreifen. So wird bspw. ein *PostConstructApplicationEvent* erzeugt, wenn die JSF-Anwendung initialisiert worden ist. Diese Events sind wiederum eher für Frameworkentwickler interessant und sollen daher hier nicht weiter thematisiert werden. Bei Bedarf finden Sie Informationen in der JSF-Spezifikation (Abschnitt 3.4, Event and Listener Model).

5.16 Konvertierung

Die HTML-Oberfläche ist textbasiert, d. h. auch alle Ein- und Ausgabewerte sind auf dieser Ebene Strings, während in der Anwendungslogik die fachlich notwendigen Datentypen Verwendung finden. Im Request-Bearbeitungszyklus wird die somit notwendige Wandlung von Werten aus der Oberfläche in die von den Managed Beans erwarteten Typen durchgeführt und umgekehrt.

Vordefinierte Konverter

Für Bean Properties der Typen *Boolean*, *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *Character*, *BigInteger* und *BigDecimal* sind Konverter vorhanden, die implizit benutzt werden, wenn nicht explizit ein Konverter registriert wird. Sie kapseln die Funktionalität der *valueOf-* und *toString*-Methoden der jeweiligen Datentypen. Sie sind nicht parametrisierbar und können nicht lokalisiert werden.

Für Properties primitiver Datentypen werden die Konverter der korrespondierenden Wrapper-Typen verwendet. Für alle weiteren Typen ist zumindest ein halber impliziter Konverter verfügbar, der in der Rendering-Phase die *toString*-Methode nutzt.

Für Werte der Typen *Byte*, *Short*, *Integer*, *Long*, *Float*, *Double*, *BigInteger* und *BigDecimal* steht mit *f:convertNumber* ein Tag zur Verfügung, mit dem ein flexibel konfigurierbarer Konverter für ein Ein- oder Ausgabefeld registriert werden kann. Das geschieht durch die Platzierung des Tags innerhalb des Ein- bzw. Ausgabeelements (Listing 5.39).

```
<h:inputText value="#{converterModel.someNumber}">
  <f:convertNumber pattern="#,##0.00"/>
</h:inputText>
```

Listing 5.39: Registrierung des vordefinierten Konverters für numerische Werte

Der Konverter ist internationalisiert, nutzt also sprach- und landesabhängige Formate, und lässt sich mit diversen Attributen weiter beeinflussen:

- *type:* Konvertierungstyp *number* (Default), *currency* oder *percentage*
- *currencyCode, currencySymbol:* Währung (bei *type currency*)
- *groupingUsed:* Soll ein Gruppierungszeichen genutzt werden?
- *integerOnly:* Nur ganzzahligen Anteil berücksichtigen?
- *minFractionDigits, maxFractionDigits, minIntegerDigits, maxIntegerDigits:* Stellenanzahl nach und vor dem Dezimaltrennzeichen
- *pattern:* Zahlenmuster wie bei *java.text.DecimalFormat*
- *locale:* Locale

Für *Date*-Werte steht ein weiterer vordefinierter Konverter zur Verfügung. Er wird mit *f:convertDateTime* registriert (Listing 5.40).

```
<h:inputText value="#{converterModel.someDate}">
  <f:convertDateTime type="both" dateStyle="short" timeStyle="short"/>
</h:inputText>
```

Listing 5.40: Registrierung des vordefinierten Konverters für Datums- und Zeitwerte

f:convertDateTime kann mit den folgenden Attributen beeinflusst werden:

- *type:* Konvertierungstyp *date* (Default), *time* oder *both*
- *dateStyle, timeStyle:* Präsentationsstil *default, short, medium, long* oder *full* für Datum bzw. Zeit. Die Bedeutung dieser Werte ist sprach- und landesabhängig
- *pattern:* Muster wie bei *java.text.SimpleDateFormat*
- *timeZone:* Zeitzone
- *locale:* Locale

Der Kontextparameter *javax.faces.DATETIMECONVERTER_DEFAULT_TIMEZONE_IS_SYSTEM_TIMEZONE* der Webanwendung bestimmt die voreinstellte Zeitzone des Konverters: Ist der Parameter *false*, wird UTC verwendet, sonst die Systemzeitzone.

Custom Converter

Sollten die vordefinierten Konverter nicht ausreichend sein, z. B. weil andere Datentypen verarbeitet werden sollen, können eigene Konverter programmiert werden. Dazu wird eine Klasse als Implementierung von *Converter*[16] entworfen. Sie enthält die beiden Methoden *getAsObject* für die Konvertierung vom UI- zum Bean-Wert und *getAsString* für die umgekehrte Richtung. Die Klasse wird mithilfe der Annotation *@FacesConverter*[17] im System registriert und erhält dadurch eine ID, mit der der neue Konverter in einer View referenziert werden kann. Dazu nutzt man dort das Tag *f:converter* (Listing 5.41).

```
@FacesConverter("de.gedoplan.jsf.TextToIntConverter")
public class TextToIntConverter implements Converter
{
  public Object getAsObject(FacesContext facesContext,
                            UIComponent uiComponent,
                            String uiValue) { … }

  public String getAsString(FacesContext facesContext,
                            UIComponent uiComponent,
                            Object beanValue) { … }
}

<h:inputText value="#{converterModel.someInt}">
  <f:converter converterId="de.gedoplan.jsf.TextToIntConverter" />
</h:inputText>
```

Listing 5.41: Definition und Nutzung eines Custom Converters

16 *javax.faces.convert.Converter*
17 *javax.faces.convert.FacesConverter*

Die ersten beiden Parameter der Converter-Methoden erlauben den Zugriff auf den allgemeinen Kontext des Systems und auf die UI-Komponente, der der Converter zugeordnet ist. In vielen Fällen werden diese Parameter nicht benötigt.

Sollte bei der Übernahme eines UI-Texts in der Methode *getAsObject* ein Fehler auftreten, i. d. R., weil der angelieferte Text dem erwarteten Format nicht entspricht, so kann er in Form einer *ConverterException*[18] zum Aufrufer geliefert werden (Listing 5.42). Die darin enthaltene Meldung wird vom System mit der betroffenen UI-Komponente assoziiert und kann mithilfe der weiter unten besprochenen Tags *h:message* und *h:messages* zur Anzeige gebracht werden.

```
public Object getAsObject(…)
{
  // Falls Konvertierung nicht möglich:
  FacesMessage msg = new FacesMessage("Fehler: …");
  msg.setSeverity(FacesMessage.SEVERITY_ERROR);
  throw new ConverterException(msg);
}
```

Listing 5.42: Melden eines Konvertierungsfehlers

Die Annotation *@FacesConverter* kann statt zur Vergabe einer ID auch zur Registrierung des Converters für einen bestimmten Datentyp verwendet werden: Wird statt mit dem Parameter *forClass* eine Klasse übergeben, wird der Converter für Daten dieses Typs standardmäßig verwendet (Listing 3.43).

```
@FacesConverter(forClass=Color.class)
public class TextToColorConverter implements Converter
{
  …
```

Listing 5.43: Custom Converter als Default für einen Datentyp

Ausgabe von Converter- oder Validierungsmeldungen

Die durch Converter erzeugten Meldungen können mithilfe der folgenden Tags angezeigt werden:

- *h:message* zeigt eine Meldung zu einer durch das Attribut for referenzierten UI-Komponente an. Sollten dieser Komponente mehrere Meldungen zugeordnet sein, wird nur die erste ausgegeben.
- *h:messages* zeigt alle Meldungen zu allen Komponenten der Seite an. Mit dem Attribut *globalOnly* kann die Ausgabe auf die Meldungen eingeschränkt werden, die keiner UI-Komponente zugeordnet sind.

18 *javax.faces.convert.ConverterException*

Die Meldungen haben eine der Dringlichkeiten *SEVERITY_INFO, SEVERITY_WARN, SEVERITY_ERROR* oder *SEVERITY_FATAL*[19]. Die beiden Tags können mit den Attributen infoClass, infoStyle, warnClass, warnStyle etc. so konfiguriert werden, dass sie für die Severities unterschiedliche CSS-Klassen bzw. -Stile verwenden.

Zudem haben Meldungen einen Detailtext und einen zusammenfassenden Text. Mit den Attributen *showDetail* und *showSummary* können die Tags angewiesen werden, welche Meldungsteile ausgegeben werden sollen.

Eine typische Anwendung der beiden Tags zeigt Listing 3.44.

```
<h:inputText id="id1" …>…</h:inputText>
<h:message for="id1" errorStyle="color: red"/>

<h:inputText id="id2" …>…</h:inputText>
<h:message for="id2" errorStyle="color: red"/>

<h:messages globalOnly="true" errorStyle="color: red"/>
```

Listing 5.44: Meldungsausgabe

Die Meldungen umfassen nicht nur Konvertierungsmeldungen. Vielmehr gehen hier auch Validierungsmeldungen und allgemeine Meldungen ein.

Jede View sollte zumindest ein *h:messages*-Element enthalten, damit Meldungen nicht unbemerkt bleiben. Im Development-Modus (Kontextparameter *javax.faces.PROJECT_STAGE*) wird in Facelets ohne eines der beiden Tags automatisch am Ende ein *h:messages* eingefügt.

5.17 Validierung

In der Validierungsphase schließt sich an die Konvertierung der Eingabewerte die Validierung an, d. h. die Prüfung, ob die Werte bestimmten Regeln entsprechen. JSF kann mit den Validierungs-Tags *f:validateXyz* und ähnlichen Parametern JSF-eigene oder auch individuell programmierte Validierer ansprechen. Hier gibt es allerdings mit der ebenfalls im Standard vorhandenen Spezifikation Bean Validation eine nahezu vollständige Überdeckung. Es gibt daher zumindest für neuere Anwendungen keinen Grund mehr, die JSF-spezifischen Validierungsmöglichkeiten einzusetzen. Sollten Sie dennoch Informationen dazu benötigen, finden Sie sie in der JSF-Spezifikation (Abschnitt 3.5, Validation Model).

19 Konstanten aus *javax.faces.application.FacesMessage*

Validierung von Eingabewerten

Die Validierung der Eingabewerte mittels Bean Validation funktioniert ohne weitere Konfiguration. Es reicht also, die an die Eingabe-Tags gebundenen Werte mit den gewünschten Bean Validation Constraints zu belegen (Listing 5.45).

```
public class Auto
{
  @NotNull @Size(min = 1)
  private String  name;

  private boolean kombi;

  @Min(2) @Max(5)
  private int     anzahlTueren;

  // … Getter und Setter
}

@Named
public class ValidationModel
{
  private Auto auto = new Auto();

  public Auto getAuto() { return this.auto; }
}

<h:form>
  <h:inputText value="#{validationModel.auto.name}" />
  <h:selectBooleanCheckbox value="#{validationModel.auto.kombi}" />
  <h:inputText value="#{validationModel.auto.anzahlTueren}" />
  …
</h:form>
```

Listing 5.45: Einsatz von Bean Validation zur Validierung von Eingabewerten

Wichtig für eine komplette Validierung ist, dass der Anwendungskontextparameter *javax.faces.INTERPRET_EMPTY_STRING_SUBMITTED_VALUES_AS_NULL* den Wert *true* hat. Ansonsten würde *@NotNull* nicht wie erwartet arbeiten.

Feldübergreifende Validierung

Das bisher Gezeigte berücksichtigt nur Constraints auf einzelnen Eingabewerten. Häufig gibt es aber auch Abhängigkeiten zwischen mehreren Feldern, die eine übergreifende Prüfung verlangen. Im Kapitel über Bean Validation wurde beschrieben, wie solche Validierungsregeln als Constraints auf Klassenebene definiert werden. In Listing 5.46 ist dies nochmals angedeutet.

```
@ValidAuto
public class Auto
{
  …
}

@Constraint(validatedBy = AutoValidator.class)
@Target({ ElementType.FIELD, ElementType.METHOD, ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidAuto
{
  …
}

public class AutoValidator
  implements ConstraintValidator<ValidAuto, Auto>
{
  public boolean isValid(Auto auto, …)
  {
    // Feldübergreifende Prüfung
    return …;
  }
  …
}
```

Listing 5.46: Bean Validation Constraint für eine Prüfung auf Klassenebene

Die Einbindung der Validierung auf Klassenebene stellt sich in JSF leider als nicht ganz so einfach dar. Die Ursache dafür ist, dass durch die Eingabefelder in der Seitenbeschreibung zwar die einzelnen Properties der einzugebenden Objekte referenziert werden, dem System die kompletten Objekte aber nicht bekannt gemacht werden. So sind im Beispiel des vorigen Abschnitts zwar Bindungen zu *name*, *kombi* und *anzahlTueren* eines Autos vorhanden, das komplette *Auto*-Objekt wird jedoch nirgends referenziert. Schlimmer noch – im Sinne der Validierung: Das *Auto*-Objekt der Beispielanwendung wird erst nach Ablauf der Validierungsphase mit den eingegebenen Werten befüllt. Somit ist es prinzipbedingt nicht möglich, dieses Objekt innerhalb der Validierungsphase zu überprüfen.

Als Lösungen stehen im Wesentlichen zwei Vorgehensweisen zur Verfügung: Prüfung des eingegebenen Geschäftsobjekts in Phase 5 (*Invoke Application*) oder Validierung eines temporär erstellten Objekts in Phase 3 (*Process Validations*).

Die erste Variante ist zweifellos die einfachere, da das betroffene Geschäftsobjekt bereits die zu prüfenden Werte enthält. Die Prüfung kann somit bspw. in der Aktionsmethode des auslösenden Buttons durchgeführt werden, indem das Objekt der Methode *Validator.validate* übergeben wird. Im Fall von Validierungsfehlern enthält das Ergebnis des Methodenaufrufs die einzelnen Meldungen, die man recht einfach in *FacesMessages* umsetzen und dem aktuellen Request-Kontext hinzufügen kann (Listing 5.47).

```
@Named
public class ValidationModel
{
  private Auto auto = new Auto();

  @Inject
  private FacesValidationHelper facesValidationHelper;
  …
  public void save()
  {
    // Cross Component Validation vor der Verarbeitung
    if (!this.facesValidationHelper.validate(this.auto))
    {
      // Falls nicht valide, ...
      // ... eintragen (kann von anderen wieder abgefragt werden)
      FacesContext facesContext = FacesContext.getCurrentInstance();
      facesContext.validationFailed();

      // ... weitere Verarbeitung abbrechen
      return;
    }
    …
  }
}

@ApplicationScoped
public class FacesValidationHelper
{
  @Inject
  private Validator validator;

  /**
   * Objekt mit BV validieren und Meldungen als FacesMessages im
   * FacesContext eintragen.
   * @param object zu validierendes Objekt
   * @return true, falls das Objekt valide ist
   */
  public boolean validate(Object object)
  {
    Set<ConstraintViolation<?>> constraintViolations
      = (Set) this.validator.validate(object);
    return convertToFacesMessages(constraintViolations) == 0;
  }

  /**
   * BV-Validierungsmeldungen im FacesContext eintragen.
   * @param constraintViolations Validierungsmeldungen
   * @return Anzahl eingetragenen Meldungen
   */
  private int convertToFacesMessages(
              Set<ConstraintViolation<?>> constraintViolations)
  {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    for (ConstraintViolation<?> cv : constraintViolations)
```

```
    {
      FacesMessage msg = new FacesMessage(cv.getMessage());
      msg.setSeverity(FacesMessage.SEVERITY_ERROR);
      facesContext.addMessage(null, msg);
    }
    return constraintViolations.size();
  }
…
```

Listing 5.47: Feldübergreifende Validierung in einer Aktionsmethode

Eine Abwandlung dieser Vorgehensweise ist es, darauf zu vertrauen, dass die von der Aktionsmethode aufgerufene Geschäftslogik die Validität der verarbeiteten Daten sicherstellt und im Fall eines Regelbruchs eine entsprechende Exception auswirft. Das ist bspw. der Fall, wenn die erfassten Objekte von Java Persistence als Entities in eine Datenbank gespeichert werden. Dabei wird die Gültigkeit der Objekte automatisch mittels Bean Validation geprüft und ggf. eine entsprechende Exception ausgeworfen. Die Validierungsmeldungen können dann daraus entnommen werden (Listing 5.48).

```
@Named
public class ValidationModel
{
  private Auto auto = new Auto();

  @Inject
  private FacesValidationHelper facesValidationHelper;
  …
  public void save()
  {
    try
    {
      // Geschäftslogik aufrufen
    }
    catch (Exception e)
    {
      // Validierungsmeldungen aus Exception holen
      if (this.facesValidationHelper.convertToFacesMessages(e) != 0)
      {
        // Falls Meldungen erzeugt, ...
        // ... eintragen (kann von anderen wieder abgefragt werden)
        FacesContext facesContext = FacesContext.getCurrentInstance();
        facesContext.validationFailed();

        // ... weitere Verarbeitung abbrechen
        return;
      }
      else
      {
        // Falls andere Exception, Aufrufer damit betrauen
        throw e;
      }
```

```
    }
  }
}

@ApplicationScoped
public class FacesValidationHelper
{
  /**
   * Ggf. in der Exception enthaltenen BV Constraint Violations
   * als FacesMessages im FacesContext eintragen.
   *
   * Es wird die gesamte Exception Chain durchsucht, d. h. eine
   * ConstraintViolationException wird auch dann gefunden, wenn sie
   * nur (mittelbare) Cause der übergebenen Exception ist.
   *
   * @param throwable Exception
   * @return Anzahl eingetragenen Meldungen
   */
  public int convertToFacesMessages(Throwable throwable)
  {
    while (throwable != null)
    {
      if (throwable instanceof ConstraintViolationException)
      {
        return convertToFacesMessages(
 ((ConstraintViolationException) throwable).getConstraintViolations());
      }
      throwable = throwable.getCause();
    }

    return 0;
  }

  /**
   * BV-Validierungsmeldungen im FacesContext eintragen.
   * @param constraintViolations Validierungsmeldungen
   * @return Anzahl eingetragenen Meldungen
   */
  private int convertToFacesMessages(
              Set<ConstraintViolation<?>> constraintViolations)
  {
    FacesContext facesContext = FacesContext.getCurrentInstance();
    for (ConstraintViolation<?> cv : constraintViolations)
    {
      FacesMessage msg = new FacesMessage(cv.getMessage());
      msg.setSeverity(FacesMessage.SEVERITY_ERROR);
      facesContext.addMessage(null, msg);
    }
    return constraintViolations.size();
  }
…
```

Listing 5.48: Feldübergreifende Validierung innerhalb der Geschäftslogik

Der Nachteil der bislang gezeigten Vorgehensweisen ist, dass eventuell inkonsistente Daten im Geschäftsobjekt landen können. Die weitere Verarbeitung der Daten muss darauf also Rücksicht nehmen. Möchte man das vermeiden, so muss die Prüfung vor dem Eintrag der Daten in das Objekt geschehen.

Eine elegante Möglichkeit, eine entsprechende Prüfung zum richtigen Zeitpunkt, nämlich am Ende der Validierungsphase, aufrufen zu lassen, ist die Registrierung eines Listeners für den *postValidate*-Event des Formularelements, das die zu prüfenden Eingabefelder enthält (Listing 5.49).

```
<h:form>
  <f:event listener="#{validationModel.validateCrossComponents}"
          type="postValidate" />
  <h:inputText value="#{validationModel.auto.name}" />
  <h:selectBooleanCheckbox value="#{validationModel.auto.kombi}" />
  <h:inputText value="#{validationModel.auto.anzahlTueren}" />
  …
</h:form>
```

Listing 5.49: Registrierung einer Listener-Methode zur feldübergreifenden Validierung

Die referenzierte Methode wird aufgerufen, nachdem die Einzelfeldvalidierungen bereits durchgeführt sind. Ob dabei bereits Fehler festgestellt wurden, wird mit der Methode *FacesContext.isValidationFailed()* ermittelt. In diesem Fall macht die feldübergreifende Prüfung meist keinen Sinn und die Validierungsmethode wird verlassen.

Anschließend werden die betroffenen Eingabekomponenten gesucht, ihre aktuellen Werte ausgelesen und daraus ein temporäres Objekt für die Prüfung hergestellt. Die Validierung des Objekts inklusive der Verarbeitung der ggf. entstandenen Meldungen verläuft analog zum oben Gezeigten. Im Fall eines Validierungsfehlers wird durch Aufruf von *FacesContext.renderResponse* erreicht, dass die Phasen 4 und 5 (*Update Model Values* und *Invoke Application*) übersprungen werden (Listing 5.50).

```
@Named
public class ValidationModel
{
  @Inject
  private FacesValidationHelper facesValidationHelper;
  …
  /**
   * Feldübergreifende Validierung der Eingabewerte.
   *
   * @param componentSystemEvent Event
   */
  public void validateCrossComponents(
                      ComponentSystemEvent componentSystemEvent)
  {
    // Falls schon Validierungfehler, nichts mehr prüfen
    FacesContext facesContext = FacesContext.getCurrentInstance();
```

```
    if (facesContext.isValidationFailed())
      return;

    // Eingabewerte aus den Komponenten holen
    UIComponent form = componentSystemEvent.getComponent();
    String name = getInputValue(form, "name");
    boolean kombi = getInputValue(form, "kombi");
    int anzahlTueren = getInputValue(form, "anzahlTueren");

    // Komplettobjekt daraus erstellen und prüfen
    Auto auto4validation = new Auto(name, kombi, anzahlTueren);
    if (!this.facesValidationHelper.validate(auto4validation))
    {
      // Falls nicht valide, ...
      // ... eintragen (kann von anderen wieder abgefragt werden)
      facesContext.validationFailed();

      // ... nach Validierungsphase direkt zur Renderphase
      facesContext.renderResponse();
    }
  }

  /**
   * Hilfsmethode zum Auslesen eines Komponentenwertes.
   *
   * @param <T> Erwarteter Ziel-Typ
   * @param anchorComponent Anker-Komponente, i. d. R. h:form
   * @param componentId Id der Eingabekomponente
   * @return aktueller Wert der Komponente
   */
  private <T> T getInputValue(UIComponent anchorComponent,
                              String componentId)
  {
    UIInput component
      = (UIInput) anchorComponent.findComponent(componentId);
    return (T) component.getValue();
  }
…
```

Listing 5.50: Listener-Methode für die feldübergreifende Validierung

Bei dieser Vorgehensweise ist vorteilhaft, dass bei nicht validen Daten weder ein Update der Model-Objekte noch ein Aufruf der Geschäftslogik geschehen. Dieser Vorteil wird aber teuer erkauft: Die Zuordnung der Eingabekomponenten zu den Properties der Geschäftsobjekte befindet sich hier redundant in der Seitenbeschreibung und in der Validierungsmethode. Änderungen der Objektstruktur müssen somit an zwei Stellen nachgepflegt werden und eine Designänderung zieht ggf. veränderte Komponenten-IDs nach sich und damit eine gleichlautende Modifikation der Komponentenzugriffe in der Validierungsmethode. Das klingt nach Schwierigkeiten bei der Weiterentwicklung von Anwendungen. Von dieser Validierungsmöglichkeit ist daher abzuraten.

5.18 Immediate-Komponenten

Der Request Processing Lifecycle läuft in der eingangs beschriebenen Reihenfolge ab, wobei die Komponenten der aktuellen Seite in jeder Phase in der Reihenfolge ihres Auftretens im Komponentenbaum behandelt werden. Mit dem Attribut *immediate* kann auf die Abarbeitungsreihenfolge Einfluss genommen werden. Dieses Attribut ist für Eingabe- und Aktionskomponenten vorhanden und hat den Vorgabewert *false*.

„immediate" für Eingabekomponenten

Setzt man *immediate* für eine Eingabekomponente auf *true*, werden deren Konvertierung und Validierung vorgezogen. Die aktuellen Eingabewerte der Komponenten mit *immediate="true"* sind somit bereits validiert und in die entsprechenden Komponentenobjekte als Values eingetragen, wenn die Konvertierung und Validierung der restlichen Komponenten abläuft. Nutzbar ist diese Änderung der Verarbeitungsreihenfolge bspw. für eine feldübergreifende Validierung, allerdings sind die oben beschriebenen Verfahren hier günstiger.

„immediate" für Aktionskomponenten

Wird das Attribut *immediate* für eine Aktionskomponente auf *true* gesetzt, wird bei deren Betätigung die entsprechende Aktionsmethode ausgeführt und anschließend die Render-Phase durchlaufen. Die früheren Phasen werden für nur für solche Komponenten durchlaufen, die ebenfalls *immediate="true"* haben. Der Standardanwendungsfall hierfür ist der Abbrechen- oder Cancel-Button eines Eingabeformulars. Er führt i. d. R. zum Verlassen des Eingabedialogs. Die aktuellen Eingabewerte sollen also nicht validiert und übernommen werden (Listing 5.51).

```
<h:form>
  <h:inputText … />
  <h:inputText … />
  <h:commandButton value="ok" … />
  <h:commandButton value="cancel" immediate="true" …/>
</h:form>
```

Listing 5.51: Typische Konfiguration des Cancel-Buttons mit immediate="true"

5.19 Ajax

Webanwendungen arbeiten in einem Request-Response-Modell, d. h. die Kommunikation zwischen Browser und Server ist nicht dauerhaft, sondern findet nur statt, wenn sie durch eine Benutzeraktion ausgelöst wird. Der Browser sendet dann einen Request zum Server, der dort in der beschriebenen Weise verarbeitet wird. Die schließlich zurückgesen-

dete Response führt zur Anzeige einer neuen Seite im Browser, womit die Verarbeitung endet.

Bei der Entwicklung hochinteraktiver Anwendungen benötigt man mehr als diesen „Aktion-Neue-Seite"-Zyklus. Es entsteht der Bedarf nach einer feingranularen Arbeitsweise, wie wir sie aus dem Bereich der Desktopanwendungen kennen:

- Neben Aktionselementen sollen auch andere Elemente eine Verarbeitung auslösen
- Als Reaktion soll nur ein Teil der Anzeige aktualisiert werden
- Die Verarbeitung soll asynchron ablaufen, um eine flüssige Bedienung der Anwendung zu ermöglichen

Das Gewünschte kann mit Ajax – Asynchronous JavaScript and XML – erreicht werden. Die zentrale Technologie darin ist die Sprache JavaScript, mit der browserseitige Logik implementiert werden kann. Diverse Events der HTML-Elemente im Browser können so Requests auslösen, die parallel zur restlichen clientseitigen Anzeigelogik – asynchron – abgearbeitet werden. Die Request-Ergebnisse führen schließlich zur Modifikation einiger Anzeigeelemente im Browser. Die Übertragung der Requests geschieht mithilfe eines *XmlHttpRequest*-Objekts, was für die weitere Betrachtung aber unerheblich ist.

Im Gegensatz zu den herkömmlichen Requests sprechen wir bei Ajax von Partial Requests, da in beiden Richtungen nur partiell Daten ausgetauscht werden: Im Request sind solche Eingabewerte enthalten, die explizit benannt werden. Umgekehrt wird durch die Response nur ein Teil der Anzeigeseite beeinflusst.

Ajax-Unterstützung finden wir seit JSF 2 im Standard. Damit hat man einerseits die Möglichkeit, UI-Komponenten mit Ajax-Funktionalität anzureichern. Andererseits steht eine JavaScript-Bibliothek zur Verfügung, die zur clientseitigen Programmierung genutzt werden kann. Der Einsatz einer weiteren Programmiersprache erhöht die Gesamtkomplexität der Anwendung allerdings nicht unerheblich. Im Folgenden wird daher hauptsächlich die erste Möglichkeit beschrieben, zumal sie in den meisten Fällen vollkommen ausreichend ist.

Ajax für Aktionselemente

Aktionselemente wie *h:commandButton* können in ihrem Body mit dem Element *f:ajax* auf die Nutzung von Partial Requests umgestellt werden. Die Attribute *execute* und *render* geben dabei eine durch Leerzeichen getrennte Liste von IDs an. IDs ohne einleitenden Doppelpunkt werden relativ zur Anwendungsstelle aufgelöst. Beginnt die ID dagegen mit *:*, wird sie relativ zur gesamten View interpretiert. Die *execute*-IDs adressieren die Komponenten, deren Werte in den ersten Phasen des Partial Requests verarbeitet werden sollen. Nur die hier angegebenen Werte werden somit in den Managed Bean Properties eingetragen. Analog bestimmt das *render*-Attribut, welche Komponenten in der Render-Phase des Partial Requests neu dargestellt werden sollen (Listing 5.52).

```
<h:form id="a">
  <h:outputText id="a1" … />
  <h:outputText id="a2" … />
  <h:commandButton … >
    <f:ajax render="@form" />
  </h:commandButton>
</h:form>
<h:form id="b">
  <h:inputText id="b1" … />
  <h:commandButton … >
    <f:ajax execute="b1" render="b1 :a:a1" />
  </h:commandButton>
</h:form>
```

Listing 5.52: Konfiguration von Buttons für Partial Requests

Für die IDs in den Attributen *execute* und *render* sind neben der relativen und absoluten Adressierung von UI-Elementen einige spezielle Werte erlaubt, die Gruppen von Elementen referenzieren. Tabelle 5.12 fasst die Möglichkeiten zusammen.

***execute*- bzw. *render*-Werte**	**Bedeutung**
id1	relativ adressierte Komponente *id1*
:id2	absolut adressierte Komponente *id2*
@all, *@none*	alle bzw. keine Komponenten
@form	alle Komponenten der Form
@this	auslösende Komponente (Default)

Tabelle 5.12: Erlaubte Elemente in den Attributen „execute“ und „render“ von „f:ajax“

Ajax-Events

Die Nutzung von *f:ajax* ist nicht auf Aktionselemente beschränkt. Vielmehr können auch andere Elemente damit versehen werden. Insofern ist die zuvor dargestellte Nutzung von Ajax bei Aktionselementen nur ein häufig vorkommender Spezialfall.

Mit dem Attribut *event* kann der Auslöser des Partial Requests bestimmt werden. Neben den Standard-Events *action* und *valueChange* sind auch HTML-Events nutzbar. Wird *event* nicht angegeben, gilt für Aktionselemente *action*, für *h:inputXyz* und *h:selectXyz valueChange*.

Mit dem Attribut *listener* kann eine Methode angegeben werden, die während der Bearbeitung des Requests aufgerufen wird. Sie kann parameterlos sein oder einen Parameter vom Typ *AjaxBehaviorEvent* annehmen (Listing 5.53).

```
<h:commandButton … >
  <f:ajax event="action" …
        listener="#{ajaxDemoModel.logAjaxBehaviorEvent}" />
```

```
</h:commandButton>
<h:inputText … >
  <f:ajax event="valueChange" …
        listener="#{ajaxDemoModel.logAjaxBehaviorEvent}" />
</h:inputText>
<h:inputText … >
  <f:ajax event="mouseover" …
        listener="#{ajaxDemoModel.logAjaxBehaviorEvent}" />
</h:inputText>

@Named
public class AjaxDemoModel
{
  public void logAjaxBehaviorEvent(AjaxBehaviorEvent ajaxBehaviorEvent)
  {
    …
  }
  …
}
```

Listing 5.53: Konfiguration diverser UI-Komponenten zur Auslösung von Partial Requests

Bei der Ausführung der Partial Requests werden natürlich auch die bisher besprochenen Event Listener aufgerufen, sofern solche konfiguriert sind und das betroffene Element im execute-Attribut aufgeführt ist.

Ajax Callbacks

Bislang wurde die Verarbeitung der Partial Requests auf der Serverseite beschrieben. Es ist aber mithilfe von Attributen von *f:ajax* möglich, zusätzliche JavaScript-Methoden im Browser aufrufen zu lassen: Mit *onevent* und *onerror* können JavaScript-Methoden mitgegeben werden, die im Verlauf der Verarbeitung des Requests aufgerufen werden. Die Methoden erhalten einen Parameter, dessen Property *status* den Grund des Aufrufs angibt (Listing 5.54).

```
<script type="text/javascript">
  function showAjaxEvent(data) { alert("Status: " + data.status); };
  function showAjaxError(data) { alert("Fehler: " + data.status); };
</script>
…
<h:commandButton … >
  <f:ajax onevent="showAjaxEvent" onerror="showAjaxError" />
</h:commandButton>
```

Listing 5.54: Registrierung von Callback-Funktionen für Ajax Requests

onevent-Methoden werden dreimal für jeden Request aufgerufen: Vor Absenden des Requests zum Server, nach Erhalt der Antwort und nach der Aktualisierung der GUI-Elemente. *status* enthält dabei die Werte *begin*, *complete* bzw. *success*.

onerror-Methoden werden aufgerufen, wenn die Bearbeitung des Requests fehlerhaft endete. stat*us* gibt mit den Werten *httpError*, *serverError*, *malformedXML* oder *emptyResponse* einen Hinweis auf die Fehlerursache.

JavaScript API

Zur Nutzung der Ajax-Funktionalität bietet JSF 2 einige JavaScript-Funktionen an. Diese stehen automatisch zur Verfügung, wenn *f:ajax* in einer Seite genutzt wird. Sie können aber auch explizit geladen werden. Interessant sind hier ggf die beiden Funktionen *jsf.ajax.addOnEvent* und *jsf.ajax.addOnError*, die die globale Registrierung von Callback-Funktionen erlauben.

Tabelle 5.13 fasst die Funktionen zusammen. Details dazu finden Sie in der JSF-Spezifikation (Abschnitt 14, JavaScript API).

Funktion	Beschreibung
jsf.ajax.request (source, event, options)	Ajax Request an Server senden
jsf.ajax.response (request, context)	Serverantwort verarbeiten
jsf.ajax.addOnError(callback)	*onerror*-Methode registrieren
jsf.ajax.addOnEvent(callback)	onevent-Methode registrieren

Tabelle 5.13: JavaScript-Funktionen zur Ajax-Unterstützung

5.20 Templating mit Facelets

Ein wesentlicher Vorteil von Facelets gegenüber JavaServer Pages als Seitenbeschreibungssprache ist die Möglichkeit, Seiten schablonenartig zusammenzusetzen, also beispielsweise ein allgemeines Grundgerüst aller Seiten einer Anwendung in einer Datei zu definieren und diese in die Beschreibung der Einzelseiten einzubeziehen. Mit dieser Vorgehensweise lässt sich z. B. ein einheitliches Erscheinungsbild der Views einer Anwendung erreichen – sicher ein wichtiger Faktor für den Erfolg einer Anwendung.

Facelet Templates sind XHTML-Seiten mit Parametern und Platzhaltern, die von konkreten Seiten – den sog. Template Clients – mit Werten und Inhalten bestückt werden können. Die schließlich zur Anzeige kommende Seite ist somit eine Mischung der Elemente aus Template und Template Client (Abb. 5.10).

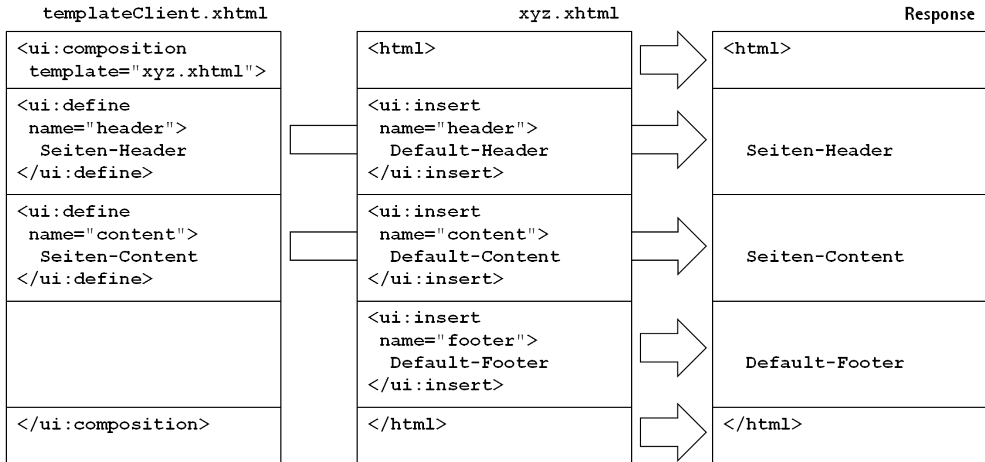


Abbildung 5.10: Mischung der Ergebnisseite aus Template Client und Template

Template

Jedes Facelet kann als Template dienen. Die Besonderheit liegt darin, dass in einem Template beliebig viele Elemente *ui:insert* aufgenommen werden können, die als Platzhalter für Inhalt dienen, der später vom Template Client beigesteuert wird. Das Tag-Präfix *ui* verweist dabei auf den Namespace *http://java.sun.com/jsf/facelets*, in dem die Template Tags zusammengefasst sind. Ein Template kann zudem neue EL-Variablen als Parameter verwenden, die ebenfalls vom Template Client mit Werten versorgt werden können. Templates können an einem beliebigen Platz innerhalb der Webanwendung abgelegt werden. Häufig wählt man dafür Verzeichnisse unterhalb von *WEB-INF*, weil diese vom Browser nicht direkt adressiert werden können. Ein einfaches Template mit den Platzhaltern *header*, *content* und *footer* sowie dem Parameter *title* zeigt Listing 5.55.

```
<html … xmlns:ui="http://java.sun.com/jsf/facelets">
<h:head>
  <title><h:outputText value="#{title}" /></title>
</h:head>

<h:body>
  <h:panelGroup layout="block" style="background-color: blue; …">
    <ui:insert name="header">
      <h:outputText value="#{title}" />
    </ui:insert>
  </h:panelGroup>

  <h:panelGroup layout="block">
    <ui:insert name="content">Default Content</ui:insert>
  </h:panelGroup>
```

```
  <h:panelGroup layout="block" style="font-size: 12px; …">
    <ui:insert name="footer">
      <h:outputText value="#{facesContext.viewRoot.viewId}" />
    </ui:insert>
  </h:panelGroup>
</h:body>
</html>
```

Listing 5.55: Beispielhaftes Template

Template-Client

Eine Seite, die ein Template verwendet, wird wiederum durch eine XHTML-Datei beschrieben. Der wesentliche Teil darin ist das Element *ui:composition*, das mit seinem Attribut *template* das Template referenziert und den zum Seitenaufbau genutzten Teil umschließt. Die Angabe des Templates in *ui:composition* kann relativ zur aktuellen Seite geschehen oder – beginnend mit einem / – relativ zum Context der Webanwendung.

Innerhalb von *ui:composition* werden mit *ui:param*-Elementen Parameter übergeben. Die Attribute *name* und *value* definieren den Namen bzw. Wert des Parameters. Werden im Template verwendete Parametervariablen nicht übergeben, erscheinen sie in der Auswertung der EL-Ausdrücke als leer.

ui:define-Elemente füllen je einen Platzhalter des Templates mit dem Inhalt des Tags. Auf diese Weise nicht gefüllte Platzhalter haben den im Template definierten Inhalt.

Für die Anzeige spielt nur der Inhalt von *ui:composition* eine Rolle, der Rest des Template-Client-Files wird im Rendering ignoriert (Listing 5.56).

```
<html … xmlns:ui="http://java.sun.com/jsf/facelets">
<ui:composition template="/WEB-INF/templates/standard.xhtml">
  <ui:param name="title" value="Seite 2" />
  <ui:define name="content">
    <h:form>
      <h:panelGrid columns="2">
        <h:outputLabel value="Name: " for="name" />
        <h:inputText id="name" />
        …
      </h:panelGrid>
    </h:form>
  </ui:define>
</ui:composition>
</html>
```

Listing 5.56: Seitendefinition auf Basis des Templates aus Listing 5.55

Mehrstufige Templates

Templates können sich ihrerseits auf Templates beziehen. Damit ist eine beliebig tiefe Schachtelung von Templates möglich (Listing 5.57).

Bei der Nutzung eines Sub-Templates können sowohl dessen Platzhalter und Parameter mit Werten versehen werden als auch diejenigen der höheren Templates.

```
<!-- WEB-INF/templates/demo/standard.xhtml -->
<html …>
  …
  <ui:insert name="content">Default Content</ui:insert>
  …
</html>


<!-- WEB-INF/templates/demo/borderLayout.xhtml -->
<ui:composition template="/WEB-INF/templates/demo/standard.xhtml">
  <ui:define name="content">
    …
    <ui:insert name="west" />
    …
  </ui:define>
  …
</ui:composition>


<!-- templatedPage3.xhtml -->
<ui:composition template="/WEB-INF/templates/demo/borderLayout.xhtml">
  …
  <ui:define name="west">…</ui:define>
  …
</ui:composition>
```

Listing 5.57: Mehrstufige Template-Nutzung

Mehrere Templates pro Seite

ui:composition kann pro Seite nur einmal vorkommen, da der umliegende Bereich für die Anzeige ausgeblendet wird. Dementsprechend kann so auch nur jeweils ein Template referenziert werden. Für den Fall, dass in einem Template-Client ein Seitenbereich auf Basis eines weiteren Templates aufgebaut werden soll, bietet die Tag-Bibliothek das Element *ui:decorate* an. Es referenziert wie *ui:composition* ein Template, allerdings wird der Rest der Seite in der Anzeige nicht unterdrückt.

ui:decorate kann auf einer Seite mehrfach verwendet werden – auch zusätzlich zu *ui:composition*. Somit ist es möglich, die gesamte Seite auf Basis eines Templates zu definieren und beliebig viele weitere Templates für bestimmte Bereiche der Seite zu nutzen.

In Dekorations-Templates findet man häufig *ui:insert* ohne *name*, womit der gesamte Inhalt von *ui:decorate* zur Anzeige gebracht wird. Hier wäre aber ebenso eine gezielte Adressierung von benannten Platzhaltern möglich (Listing 5.58).

```
<!-- /WEB-INF/templates/messageBox.xhtml -->
<html …>
  <h:panelGroup style="…">
    <ui:insert/>
  </h:panelGroup>
</html>


<!-- templatedPage4.xhtml -->
<ui:composition template="/WEB-INF/templates/borderLayout.xhtml">
  <ui:param name="title" value="Seite 4" />
  <ui:define name="west">
    <ui:decorate template="/WEB-INF/templates/messageBox.xhtml">
      Jetzt auch mit mehreren Templates!
    </ui:decorate>
  </ui:define>
  …
</ui:composition>
```

Listing 5.58: Nutzung mehrerer Templates in einer Seite

5.21 Eigene JSF-Komponenten

Die Grundausstattung mit Anzeigekomponenten ist in den Standardbibliotheken sehr überschaubar: Elemente mit mehr Funktionalität oder einem ansprechenderen Design als die im HTML-Umfang befindlichen sucht man hier vergebens. Das ist durchaus so gewollt: Dieses Feld wird bewusst Komponentenbibliotheken überlassen, um den Standard nicht zu überladen.

Es ist aber auch möglich, selbst Komponenten zu entwickeln. War das früher eine eher komplexe Tätigkeit, bei der mehrere Klassen und Deskriptoren zu schreiben waren, bietet JSF seit der Version 2 die Möglichkeit, sehr elegant und einfach sog. Composite Components in Form von XHTML-Beschreibungen zu entwerfen, ggf. ergänzt um je eine Komponentenklasse.

Composite Components decken nahezu alle Anforderungen der Anwendungsentwicklung ab, klassische Komponenten findet man seitdem eher in Komponentenbibliotheken anderer Hersteller vor. Die folgenden Abschnitte beschränken sich daher auf Composite Components. Weitere Details zur Komponentenentwicklung finden Sie in der JSF-Spezifikation (Abschnitt 3.6, Composite User Interface Components).

Composite Components

Wie der Name bereits vermuten lässt, können Composite Components aus anderen Komponenten zusammengesetzt werden, und zwar mit einem deklarativen Ansatz durch XHTML-Dokumente in einer Ressourcenbibliothek. Java-Klassen und XML-Deskriptoren wie im klassischen Ansatz sind hier nicht nötig.

Die Beschreibungsdateien werden in einem Ressourcenordner abgelegt, d. h. im Ordner *resources* der Webanwendung bzw. im Ordner *META-INF/resources* eines Classpath-Verzeichnisses.

Der Name einer Beschreibungsdatei ohne die Endung *.xhtml* bestimmt den Namen des damit definierten Tags. Der Bibliotheksname – also der Name des Unterverzeichnisses von *resources* – geht in den Namespace-Namen nach *http://java.sun.com/jsf/composite/* ein. So definiert bspw. die Datei r*esources/components/ee-demos/helloComponentWorld.xhtml* ein Tag namens *helloComponentWorld* im Namespace *http://java.sun.com/jsf/composite/components/ee-demos*.

Die Definition einer Composite Component gliedert sich in zwei Bereiche. Das Element *interface* enthält alle Attribute, die von der nutzenden Seite verwendet werden bzw. mit Werten gefüllt werden können. *implementation* beinhaltet die eigentliche Implementierung der Komponente, die Bestandteil des Komponentenbaums wird.

Die Datei enthält zwar ein komplettes XHTML-Dokument. Zur Komponentendefinition werden aber nur die beiden genannten Teile genutzt, der Rest wird ignoriert. Das Tag *html* wird allerdings i. d. R. zur Deklaration der benötigten Namespaces verwendet. Die Tags zur Definition von Composite Components liegen im Namespace *http://java.sun.com/jsf/composite*, der üblicherweise mit dem Alias *cc* importiert wird (Listing 5.59).

```
<html … xmlns:cc="http://java.sun.com/jsf/composite">
<cc:interface>
  …
</cc:interface>
<cc:implementation>
  …
</cc:implementation>
</html>
```

Listing 5.59: Grundsätzlicher Aufbau einer Composite Component

Im *cc:interface*-Teil einer Komponente werden Parameter deklariert, die bei der Verwendung der Komponente übergeben bzw. referenziert werden können. Das geschieht mithilfe der in Tabelle 5.14 aufgeführten Tags.

Tag in *cc:interface*	Bedeutung
cc:attribute	Deklariert Tag-Attribute, deren Werte im Implementierungsteil mithilfe von EL-Ausdrücken der Form *#{cc.attrs.Name}* genutzt werden können.
cc:facet	Deklariert Facets, die der Komponente zugeordnet werden können. Neben EL-Ausdrücken der Form *#{cc.facets.Name}* sind im Implementierungsteil auch zwei spezielle Tags zur Verwendung von Facets vorgesehen: *cc:renderFacet* zur direkten Ausgabe des Facet-Inhalts und *cc:insertFacet* zum Anfügen des Facets an eine Subkomponente.
cc:valueHolder	Benennt eine Subkomponente, die einen Wert enthält. Dies kann bei der Benutzung zur Zuordnung eines Konverters genutzt werden.
cc:editableValueHolder	Benennt eine Subkomponente mit einem editierbaren Wert. Dies umfasst die Wirkung von *cc:valueHolder* und kann zudem bei der Benutzung verwendet werden, um einen Change Listener anzufügen.
cc:actionSource	Benennt Aktionselemente innerhalb der Komponente. Dies kann bei der Benutzung zur Zuordnung von Action Listenern verwendet werden.

Tabelle 5.14: Im Interface der Komponente nutzbare Tags

Das Tag *cc:implementation* fasst die Bestandteile der neuen Komponente zusammen. Diese können HTML-Texte, Standard-Tags oder auch andere Komponenten sein. Zudem können die inTabelle 5.15 enthaltenen Tags sowie die inTabelle 5.16 dargestellten EL-Ausdrücke genutzt werden.

Tag in *cc:implementation*	Bedeutung
cc:insertChildren	Fügt den Body des Tags ein
cc:insertFacet	Fügt ein Facet der umschließenden Subkomponente hinzu
cc:renderFacet	Fügt den Facet-Inhalt ein

Tabelle 5.15: Tags zur Nutzung im Implementierungsteil einer Komponente

EL-Ausdruck in *cc:implementation*	Beschreibung	Beispiel
#{cc}	Referenz auf die Composite Component	<h:outputText value="#{cc.clientId}"/>
#{cc.attrs}	Zugriff auf Attribute des Interfaces	<cc:attribute name="name1" /> … <h:inputText value="#{cc.attrs.name1}"/>

EL-Ausdruck in *cc:implementation*	Beschreibung	Beispiel
#{cc.facets}	Zugriff auf Facets des Interfaces	<cc:facet name="facet1" /> … <c:if test="#{!empty cc.facets.facet1}">

Tabelle 5.16: Im Implementierungsteil einer Komponente verwendbare EL-Ausdrücke

Einige Beispiele sollen die Zusammenhänge verdeutlichen. In Listing 5.60 ist eine Komponente gezeigt, der zwei Parameter – *label* und *value* – übergeben werden. Sie werden in der Implementierung mithilfe der Ausdrücke *#{cc.attrs.…}* verwendet. Das Eingabefeld *input* innerhalb der Komponente wird zudem mit *cc:editableValueHolder* veröffentlicht, sodass das nutzende Facelet dafür einen Konverter einsetzen kann.

```
<!-- resources/components/ee-demos/labeledInputText.xhtml -->
<html …>
<cc:interface>
  <cc:attribute name="label" required="false" default="" />
  <cc:attribute name="value" required="true" />
  <cc:editableValueHolder name="input"/>
</cc:interface>
<cc:implementation>
  <h:outputLabel value="#{cc.attrs.label}" for="input" />
  <h:inputText value="#{cc.attrs.value}" id="input" />
</cc:implementation>
</html>

<!-- Nutzung der Komponente in einem Facelet -->
<ee-demos:labeledInputText
  label="Eingabe: "
  value="#{componentDemoModel.someDouble}">
  <f:convertNumber … for="input"/>
</ee-demos:labeledInputText>
```

Listing 5.60: Übergabe von Parametern an eine Composite Component

In Listing 5.61 nutzt die Komponente das Tag *cc:insertChildren*, um den vom Nutzer angegebenen Tag Body auszugeben.

```
<!-- resources/components/ee-demos/titledBox.xhtml -->
<html …>
<cc:interface>
  <cc:attribute name="title" />
</cc:interface>
<cc:implementation>
  …
  <h:outputText value="#{cc.attrs.title}" />
  <cc:insertChildren />
```

```
</cc:implementation>
</html>

<!-- Nutzung der Komponente in einem Facelet -->
<ee-demos:titledBox title="Block 2">
  <h:form>
    <h:outputText … />
    <h:inputText … />
  </h:form>
</ee-demos:titledBox>
```

Listing 5.61: Nutzung von Kindelementen in einer Composite Component

Listing 5.62 zeigt mehrere Möglichkeiten, wie die in einer Composite Component befindlichen Aktionselemente – hier Buttons – mit der Logik eines Facelets verknüpft werden können: Für den ersten Button namens *ok* definiert der Interfaceteil der Komponente einen Parameter *action*, dem eine Methodenbindung des angegebenen Typs übergeben werden muss. Das Attribut *targets* ordnet den Parameter dem Button *ok* zu. Für den zweiten Button, *cancel*, wird analog ein Methodenparameter definiert, allerdings ohne das Attribut *targets*. Stattdessen wird der Parameter in der Definition des Buttons explizit mit dem Attribut *action* referenziert. Bei der Benutzung der Komponente sind beide Varianten gleichwertig. Im ersten Fall könnten allerdings mehrere Kommandoelemente in *targets* genannt werden, die dann alle gleichartig verknüpft würden.

Durch *cc:actionSource*-Elemente im Interface werden die Aktionselemente der Komponente veröffentlicht, was bei der Verwendung der Komponente zur Registrierung von Action Listenern genutzt werden kann.

```
<!-- resources/components/ee-demos/okCancel.xhtml -->
<html …>
<cc:interface>
  <cc:attribute name="label" required="false" default="ok" />
  <cc:attribute name="action" required="true"
             method-signature="java.lang.String f()" targets="ok"/>
  <cc:actionSource name="ok" />

  <cc:attribute name="cancelLabel" required="false" default="cancel" />
  <cc:attribute name="cancelAction" required="false"
             method-signature="java.lang.String f()" />
  <cc:actionSource name="cancel" />

  <cc:actionSource name="all" targets="ok cancel" />
</cc:interface>
<cc:implementation>
  <h:commandButton id="ok" value="${cc.attrs.label}" />
  <h:commandButton id="cancel" value="${cc.attrs.cancelLabel}"
                 action="${cc.attrs.cancelAction}" />
</cc:implementation>
```

```
<!-- Nutzung der Komponente in einem Facelet -->
<ee-demos:okCancel action="#{componentDemoModel.doOk}"
                 cancelAction="#{componentDemoModel.doCancel}">
  <f:actionListener type="de.gedoplan.….DemoActionListener" for="all"/>
</ee-demos:okCancel>
```

Listing 5.62: Verknüpfung von Aktionselementen in einer Composite Component

Schließlich zeigt Listing 5.63, wie ein Facet zur Übergabe einer benannten Sequenz von JSF-Tags o. ä. an eine Komponente verwendet werden kann: *cc:facet* deklariert das Facet im Interface der Komponente. Das nutzende Facelet ist dadurch in der Lage, ein entsprechend benanntes Facet als Unterelement der Komponente anzugeben. In der Implementierung der Komponente wird der Facet-Inhalt mit *cc:renderFacet* ausgegeben. Dort ist auch zu sehen, dass der EL-Ausdruck *cc.facets*…. verwendet werden kann, um auf übergebene Facets zuzugreifen.

```
<!-- resources/components/ee-demos/headerFooterBox.xhtml -->
<html …>
<cc:interface>
  <cc:attribute name="headerStyle" … />
  <cc:facet name="header" required="false" />
  …
</cc:interface>
<cc:implementation>
  <h:panelGroup style="${cc.attrs.headerStyle}" layout="block"
               rendered="#{!empty cc.facets.header}">
    <cc:renderFacet name="header" />
  </h:panelGroup>
  …
</cc:implementation>

<!-- Nutzung der Komponente in einem Facelet -->
<ee-demos:headerFooterBox>
  <f:facet name="header">
    <h:outputText value="Box-Header" />
  </f:facet>
  …
```

Listing 5.63: Composite Component mit einem Facet

Composite Components mit Backing Bean

Zusätzlich zur Beschreibung einer neuen Komponente durch ein XHTML-File kann eine Java-Klasse referenziert werden. Damit wird es möglich, Composite Components über die reine Komposition aus anderen Komponenten hinaus mit einer eigenen Logik auszustatten.

Die Verbindung zwischen XHTML und Klasse geschieht durch einen sog. Component Type, der im *cc:interface*-Element als Attribut *componentType* angegeben wird. Die referenzierte Klasse muss mit *@FacesComponent* annotiert sein, wobei der Component Type als Parameter genutzt wird. Im *cc:implementation*-Teil der Komponente kann dann mithilfe der EL-Variablen *cc* auf die Properties und Methoden der Klasse zugegriffen werden (Listing 5.64).

```
<!-- resources/components/ee-demos/monthNavigator.xhtml -->
<html …>
<cc:interface componentType="de.gedoplan.MonthNavigator">
  …
</cc:interface>
<cc:implementation>
  …
  <h:commandButton … actionListener="#{cc.incMonth}">
  …
  …
</cc:implementation>

/**
 * Backing Bean zur Composite Component monthNavigator.
 */
@FacesComponent("de.gedoplan.MonthNavigator")
public class MonthNavigator extends UIInput implements NamingContainer
{
  public void incMonth() { … }
  …
  public String getFamily()
  {
    return "javax.faces.NamingContainer";
  }
  …
}
```

Listing 5.64: Composite Component mit Backing Bean

Die Klasse muss zumindest indirekt *UIComponent*[20] als Basisklasse besitzen, das Interface *NamingContainer*[21] implementieren und als Family *javax.faces.NamingContainer* haben. Die Family wird von der Methode *getFamily* als Ergebnis geliefert und dient zur Auswahl des richtigen Renderers. Für Details dazu sei auf die Spezifikation verwiesen.

Am einfachsten ist es, die Klasse von einer der in Tabelle 5.17 angegebenen Basisklassen abzuleiten, da damit ein Großteil der Funktionalität bereits implementiert wird. Dabei existieren naturgemäß Überschneidungen. So wird bspw. eine neue Eingabekomponente für einen Wert in vielen Fällen auch mehrere Teilkomponenten enthalten. Als Basisklasse kämen hier also *UIInput* oder *UINamingContainer* in Frage.

20 *javax.faces.component.UIComponent*
21 *javax.faces.component.NamingContainer*

Sinnvolle Basisklasse	für Komponenten, die
UIOutput	(genau) einen Wert ausgeben
UIInput	die Eingabe (genau) eines Wertes ermöglichen
UISelectOne / UISelectMany	eine Werteauswahl ermöglichen
UICommand	ein Kommandoelement darstellen
UIMessage	eine Meldung ausgeben
UINamingContainer	mehrere Teilkomponenten gruppieren

Tabelle 5.17: Übliche Basisklassen für Backing Beans von Komponenten

Den kompletten Code der Komponente *monthNavigator* hier abzudrucken, würde den Rahmen sprengen. Sie ist aber im Begleitprojekt zu diesem Kapitel mit ausführlichen Kommentaren enthalten.

5.22 Komponentenbibliotheken

Die JSF-Standardbibliothek enthält neben der algorithmischen Basis des Frameworks nur einen begrenzten Vorrat von UI-Komponenten. Für eine ansprechende Gestaltung von Anwendungen ist daher in aller Regel der Einsatz weiterer Bibliotheken unverzichtbar. Vor der JSF-Version 2 waren die Komponentenbibliotheken sehr proprietär implementiert und verlangten vielfach eine spezielle Konfiguration der gesamten Anwendung. Durch die Standardisierung insbesondere der Ajax-Integration und der Ressourcenverwaltung in JSF 2 ist der Einsatz der Zusatzbibliotheken einfach geworden. Technisch ist nun auch die gleichzeitige Verwendung verschiedener Bibliotheken machbar, wobei hier stilistisch Grenzen gesetzt sind.

Es würde den Rahmen des Buches sprengen, alle verfügbaren Komponentenbibliotheken aufzählen oder annähernd detailliert beschreiben zu wollen. Daher sei hier nur eine (kurze und unvollständige) Liste von populären Bibliotheken mit Links zu den Anbieterseiten aufgeführt:

- Apache MyFaces (*http://myfaces.apache.org*)
- ICEfaces (*http://www.icefaces.org*)
- PrimeFaces (*http://www.primefaces.org*)
- RichFaces (*http://www.jboss.org/richfaces*)

Bevor Sie fragen: Es gibt keine „beste“ Komponentenbibliothek. Alle bieten eine große Menge von UI-Komponenten an, mit denen sich optisch und technisch gute Anwendungen verwirklichen lassen. Die Eignung für ein Projekt lässt sich nur auf Basis der speziellen Anforderungen ermitteln. Auf den Beispielseiten der Anbieter können Sie sich einen ersten Eindruck verschaffen.

5.23 Security

Der Begriff Security umfasst für Anwendungen mehrere Bereiche, von denen hier nur die beiden betrachtet werden sollen, die die Berechtigungssteuerung betreffen: Login-Konfiguration, Security-Rollen und Zugriffsregeln für Webseiten. Genau genommen ist dies natürlich kein JSF-eigenes Thema. Vielmehr betrifft es Webanwendungen generell.

Login-Konfiguration

Für die Zugriffskontrolle ist es zunächst notwendig, den Benutzer zu authentisieren. Dazu nutzt man i. d. R. einen Login-Dialog, der dem Seitenzugriff vorangestellt wird, wenn der Benutzter sich noch nicht angemeldet hat. Diesen Dialog konfiguriert man mit dem Element *login-config* im Deployment Descriptor der Webanwendung (Listing 5.65).

```
<login-config>
  <auth-method>FORM</auth-method>
  <form-login-config>
    <form-login-page>/login.xhtml</form-login-page>
    <form-error-page>/login_error.xhtml</form-error-page>
  </form-login-config>
</login-config>
```

Listing 5.65: Login-Konfiguration im Descriptor „WEB-INF/web.xml“

Darin bestimmt das Element *auth-method* die Art der Authentisierung:

- *BASIC* oder *DIGEST:* Abfrage von Username und Passwort durch den Browser.

Hier verwaltet der Browser den Anmeldestatus. Der Unterschied zwischen den beiden Verfahren liegt in der Art der Übermittlung des Passworts an den Server: *BASIC* sendet das Passwort im Klartext, während es bei *DIGEST* verschlüsselt wird. Eine Abmeldung ist im Allgemeinen nur durch ein Schließen des Browsers möglich.

- *FORM:* Nutzung eines selbsterstellten Formulars zur Anmeldung.

Im Element *form-login-config* und seinen Unterelementen *form-login-pag*e und *form-error-page* werden zwei Seiten für das Login-Formular und die Anzeige von Login-Fehlern angegeben. Eine Abmeldung kann hier durch das Beenden der Session (mittels ihrer Methode *invalidate* oder duch Timeout) erreicht werden.

- *CLIENT-CERT:* Übernahme der Userinformationen aus einem Public Key Certificate.

Dieses Verfahren setzt voraus, dass die Kommunikation über HTTPS geschieht und der Benutzer ein entsprechendes Zertifikat nutzt.

Üblich ist die Nutzung des formularbasierten Verfahrens, weil dabei der Login-Dialog im Stil der restlichen Anwendung gestaltet werden kann und eine programmgesteuerte Abmeldung des Users möglich ist. Das Eingabeformular für den Login-Dialog ist dabei so zu

schreiben, dass die Eingabefelder für Usernamen und Passwort *j_username* und *j_password* heißen und der Request mittels *POST* an *j_security_check* gesendet wird (Listing 5.66).

```
<form method="POST" action="j_security_check">
  <input type="text" name="j_username"/>
  <input type="password" name="j_password"/>
  <input type="submit" value="anmelden"/>
</form>
```

Listing 5.66: Eingabeelemente im Login-Formular

Security-Rollen

Die Berechtigunssteuerung aus Sicht der Anwendung stützt sich auf sog. Security Roles ab. Sie stellen virtuelle Gruppierungen von Usern dar, wobei in der Anwendung nur der Rollenname bekannt ist, während die Zuordnung von Usern zu den Rollen Aufgabe des eingesetzten Application Servers ist. Dadurch wird vermieden, das konkrete User- oder Gruppennamen Teil der Anwendungskonfiguration werden.

Die Spezifikation regelt nicht, auf welche Weise ein Application Server die Zuordnung durchführt. Die in der Praxis anzutreffenden Verfahren reichen von einfachen Properties-Dateien über Datenbankeinträge bis zur Nutzung von Directory Services wie Active Directory, LDAP o. ä. Details finden Sie wie üblich in der Dokumentation Ihres Servers. Im Begleitprojekt zu diesem Kapitel sind Kurzanleitungen für GlassFish 3 sowie JBoss 6 und 7 enthalten.

In der Anwendung werden die von ihr genutzten Rollen deklariert, und zwar in entsprechend vielen *security-role*-Elementen im Deployment Descriptor (Listing 5.67).

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

Listing 5.67: Deklaration einer Security Role in „WEB-INF/web.xml“

Zugriffsregeln

Die Seiten einer Webanwendung sind zunächst öffentlich erreichbar. Möchte man dies einschränken, sind dazu *security-constraint*-Elemente im Deployment Descriptor der Anwendung einzutragen. Diese haben jeweils diese Unterlemente:

- *web-resource-collection:* Definition einer Teilmenge der Seiten der Anwendung.

Hier wird mittels *web-resource-name* ein Name für die Teilmenge vergeben und es werden ein oder mehrere *url-pattern*-Elemente angegeben, die bestimmen, welche Seiten Teil der Menge sind. Die hier nutzbaren Patterns zeigt Tabelle 5.18.

- *auth-constraint:* Angabe des berechtigten Benutzerkreises

Dies geschieht durch die Angabe einer oder mehrerer zuvor deklarierter Security Roles in jeweils einem *role-name*-Element.

Listing 5.68 zeigt als Beispiel ein Security Constraint, der nur Nutzern mit der Rolle *demoRole* Zugriff zu den Seiten in und unterhalb des Verzeichnisses *view/private* der Anwendung erlaubt.

```
<security-constraint>
  <web-resource-collection>
    <web-resource-name>secretPages</web-resource-name>
    <url-pattern>/view/private/*</url-pattern>
  </web-resource-collection>
  <auth-constraint>
    <role-name>demoRole</role-name>
  </auth-constraint>
</security-constraint>
```

Listing 5.68: Beispiel für ein Security Constraint

Pattern	Bedeutung
/path	Einzelseite, relativ zum Kontext der Anwendung adressiert
*/path/**	Verzeichnis inkl. seiner Unterverzeichnisse
**.extension*	URLs mit der angegebenen Endung

Tabelle 5.18: Erlaubte Werte im Element „url-pattern"

Die Aussage eines so definierten Security Constraints ist die folgende: Fordert der Browser eine Seite an, deren URL zu einem der angegeben Patterns passt, so muss der Benutzer angemeldet sein und mindestens eine der angegebenen Rollen besitzen, um die Seite angezeigt zu bekommen. Ist er noch nicht angemeldet, wird der zuvor konfigurierte Login-Prozess durchgeführt. Hat der Benutzer keine ausreichende Rollenzuordnung, wird der HTTP-Fehler *403* erzeugt, was zur Anzeige einer entsprechenden Fehlerseite führt. Diese lässt sich mit dem Element *error-page* im Deployment Descriptor konfigurieren (Listing 5.69).

```
<error-page>
  <error-code>403</error-code>
  <location>/login_insufficient.xhtml</location>
</error-page>
```

Listing 5.69: Deklaration einer Fehlerseite für einen HTTP-Fehlercode

In einem Security Constraint sind noch weitere Einstellungen möglich. So kann die Regel z. B. auf bestimmte HTTP-Methoden eingeschränkt oder eine abgesicherte Übertragung verlangt werden. Informationen dazu finden Sie in der Servlet-Spezifikation im Abschnitt 13, Security.

6 Enterprise JavaBeans

6.1 Aufgabenstellung

EJBs gehören zu den ältesten Bestandteilen der Java-EE-Plattform. Ihre Aufgabe ist – grob umrissen – die Bereitstellung von Geschäftslogik in Form von Komponenten mit definierten Schnittstellen, die sich flexibel kombinieren lassen. Sie bilden damit ein wesentliches Fundament für Enterprise-Anwendungen – zumindest bis zur Version 5 der Plattform.

Im aktuellen Release der Java EE können die Aufgaben von Enterprise JavaBeans weitgehend von CDI Beans übernommen werden. Die Integration der beiden Standards untereinander ist sehr hoch: EJBs sind Managed Beans im Sinne von CDI. Eine Injektion von EJB-Instanzen in CDI Beans und umgekehrt ist problemlos möglich.

Es ist vorstellbar, dass die Spezifikationen Enterprise JavaBeans und CDI in Zukunft weiter zusammenwachsen, bis hin zu einer vollständigen Integration von EJB in CDI. Derzeit haben EJBs allerdings noch einige Eigenschaften, die CDI Beans nicht mitbringen. Dieses Buchkapitel konzentriert sich daher nach einer Beschreibung des Aufbaus von EJBs auf ihren Mehrwert gegenüber den bereits beschriebenen CDI Beans:

- Remote-Zugriff
- Transaktionssteuerung
- Asynchrone Methoden
- Timer
- Security

Weitere Details finden Sie wie üblich in der Spezifikation[1].

6.2 Aufbau von Enterprise JavaBeans

EJBs werden durch einfache Klassen – POJOs – implementiert. Eine allgemeine Vorschrift zur Implementierung bestimmter Interfaces oder zur Ableitung von vordefinierten Basisklassen existiert nicht. Insofern entsprechen EJBs im Aufbau exakt CDI Beans.

1 JSR 318: Enterprise JavaBeansTM,Version 3.1, EJB Core Contracts and Requirements, EJB 3.1 Expert Group, 05.11.2009, *http://jcp.org* –> Search JSR 318 –> Final Release Download –> ejb-3_1-fr-spec.pdf

EJB-Methoden werden allerdings nicht direkt aufgerufen, sondern immer über ein Hüllobjekt ähnlich den dynamischen Proxies der CDI Beans in den normalen Scopes. Diese Hüllobjekte werden automatisch erzeugt, sind für den Aufrufer transparent und implementieren einen Großteil der weiter unten beschriebenen Funktionalitäten.

Durch Auszeichnung mit einer der folgenden Annotationen wird eine Klasse zu einer EJB des angegebenen Typs:

- *@Stateless*[2]: Stateless Session Bean

Objekte dieses Typs haben – wie der Name schon sagt – keinen für den Aufrufer sichtbaren Status. Sie dürfen zwar Instanzvariablen enthalten, aber es ist nicht garantiert, dass diese ihre Inhalte von einer Benutzung des Objekts zur nächsten beibehalten. Das liegt daran, dass der EJB-Container für Stateless Session Beans die Freiheit hat, die Zuordnung der EJB-Objekte zu den Hüllobjekten zwischen den Aufrufen beliebig zu wechseln. Ein Methodenaufruf eines Benutzers kann also ein Objekt benutzen, das zuvor von einem anderen Aufrufer verwendet wurde. Durch die Statuslosigkeit der EJBs ist diese mögliche Optimierung der Ressourcennutzung durch den Container für den Aufrufer transparent.

Stateless Session Beans sind implizit dem CDI-Pseudo-Scope Dependent zugeordnet. Eine andere Zuordnung ist nicht erlaubt.

Listing 6.1 zeigt eine Stateless Session Bean. Man erkennt in diesem Ausschnitt mit Ausnahme der Annotation keinen Unterschied zu einer CDI Bean. Die Session Bean profitiert allerdings implizit von der später noch beschriebenen automatischen Transaktionssteuerung, sodass bspw. bei Ausführung der Methode *insert* garantiert eine Transaktion aktiv ist, an die der Entity Manager gebunden ist.

```
@Stateless
public class ShopItemRepository
{
  @PersistenceContext(unitName = "ee_demos")
  private EntityManager entityManager;

  public void insert(ShopItem shopItem)
  {
    this.entityManager.persist(shopItem);
  }
  …
```

Listing 6.1: Beispiel für eine Stateless Session Bean[3]

2 *javax.ejb.Stateless*
3 Den in diesem Kapitel gezeigten Beispielcode finden Sie im Begleitprojekt *ee-demos-ejb*

- *@Stateful*[4]: Stateful Session Bean

Diese Objekte halten Inhalte für den Aufrufer auch über mehrere Aufrufe hinweg bereit. Die Zuordnung von EJB-Objekten zu Hüllobjekten ist hier starr: Ein Aufrufer referenziert immer das gleiche Objekt, solange seine Sitzung noch aktiv ist. Wird die EJB als Managed Bean im CDI-Sinne genutzt, d. h. wird sie z. B. durch Injektion in eine andere Bean zum Leben erweckt, bestimmt sich die Lebensdauer der Sitzung aus dem CDI-Scope der EJB. Stateful Session Beans dürfen einem beliebigen CDI Scope zugeordnet werden.

EJB-Instanzen können allerdings auch auf Nicht-CDI-Art erzeugt werden. Das ist insbesondere für die weiter unten beschriebenen Remote-Zugriffe so. In diesem Fall bestimmt der aufrufende Programmcode die Sitzungsdauer: Die Instanz wird beim ersten Zugriff erzeugt und bleibt bis zum Aufruf einer mit *@Remove*[5] annotierten Methode aktiv. Zudem kann mit *@StatefulTimeout*[6] für jede Stateful Bean ein Session Timeout angegeben werden. Wird innerhalb dieser Zeit nicht auf eine EJB-Instanz zugegriffen, kann sie vom Container verworfen werden. Listing 6.2 zeigt eine mit beiden angesprochenen Möglichkeiten ausgestattete Stateful Session Bean.

```
@Stateful
@StatefulTimeout(value = 10, unit = TimeUnit.MINUTES)
public class ShopServiceBean
{
  @Remove
  public void close()
  {
  }
  …
```

Listing 6.2: Stateful Session Bean mit Timeout und Remove-Methode

Die EJB-Instanz wird auch verworfen, wenn ein Methodenaufruf eine System Exception auswirft. System Exceptions sind meist Unchecked Exceptions, die schwerwiegende Fehlersituationen kennzeichnen. Eine genauere Definition folgt später (Abschnitt 6.5, Transaktionssteuerung).

- *@Singleton*[7]: Singleton Bean

Diese statusbehafteten Objekte werden pro Java-Prozess und Anwendung nur maximal einmal erzeugt. Dieses EJB-Konzept kann gleichwertig durch Beans im Scope Application ersetzt werden und soll hier nicht weiter betrachtet werden.

4 *javax.ejb.Stateful*
5 *javax.ejb.Remove*
6 *javax.ejb.StatefulTimeout*
7 *javax.ejb.Singleton*

- *@MessageDriven*[8]: Message-driven Bean

Objekte dieses Typs dienen als Empfänger von Meldungen. Sie können bspw. durch eingehende Meldungen aus dem Java Messaging System oder von einem Konnektor aktiviert werden. Da beides nicht Thema dieses Buches ist, sollen auch Message-driven Beans hier nicht betrachtet werden.

Zu der Bean-Klasse kommen ggf. noch Interfaces sowie ein Deployment Descriptor hinzu. Das wird in den folgenden Abschnitten beschrieben.

6.3 EJB Deployment

Das Paketierungsformat für Enterprise Java Beans ist ein EJB-JAR, d. h. ein JAR-File, in dem die Klassen, Interfaces und Deskriptoren einer oder mehrerer Beans eingepackt sind. Es erhält die Dateiendung *.jar*, ist also von außen nicht von einer einfachen Bibliothek zu unterscheiden. Die EJB-JARs einer Anwendung werden ihrerseits in ein EAR, also ein JAR-File mit der Endung *.ear*, verpackt.

Seit Java EE 6 ist es zudem möglich, EJBs als Teil einer Webanwendung zu entwerfen. Dann werden ihre Klassen und Interfaces direkt in das entsprechende WAR verpackt.

EJBs können optional mit einem Deployment Descriptor namens *ejb-jar.xml* beschrieben werden. Er befindet sich im Verzeichnis META-INF des EJB-JARs bzw. im Verzeichnis WEB-INF des WARs. Nahezu alle Einstellungen, die im Deployment Descriptor gemacht werden können, lassen sich auch mithilfe von Annotationen durchführen. Der Descriptor soll daher hier nicht weiter berücksichtigt werden. Bei Bedarf finden Sie weitere Informationen in der EJB-Spezifikation (Kapitel 19, Deployment Descriptor, und 20, Packaging). Zusätzlich zum allgemeinen Descriptor können auch serverspezifische Konfigurationsdateien verwendet werden. Informationen dazu gibt Ihnen die Dokumentation des von Ihnen verwendeten Servers.

Ob Sie sich für das einfache WAR oder das komplexere EAR entscheiden, hängt im Wesentlichen von der Größe und Struktur Ihrer Anwendung ab:

- Das klassische EAR-Format spiegelt sehr gut die grobe Architektur größerer Anwendungen wieder: Einige Webanwendungen nutzen für ihre Geschäftslogik einige EJBs. Dabei sind die Klassen der Anwendungsteile voneinander isoliert: Jede einzelne Webanwendung sowie alle EJBs werden in getrennten Classloadern geladen. Damit haben die Webanwendungen Zugriff auf die EJBs, aber nicht auf die Klassen anderer Webanwendungen. Umgekehrt können die EJBs sich untereinander aufrufen, können die Klassen der Webanwendungen jedoch nicht erreichen.

8 *javax.ejb.MessageDriven*

- Das WAR-Format eignet sich gut für den häufigen Fall einer Webanwendung mit einigen EJBs. Die Klassen der Anwendung werden hier allerdings von nur einem Classloader geladen, können sich also beliebig gegenseitig aufrufen.

Die Art der Paketierung hat keinen Einfluss auf die in diesem Kapitel beschriebenen Eigenschaften von EJBs. Abbildung 6.1 stellt die beiden Formate nochmals einander gegenüber.

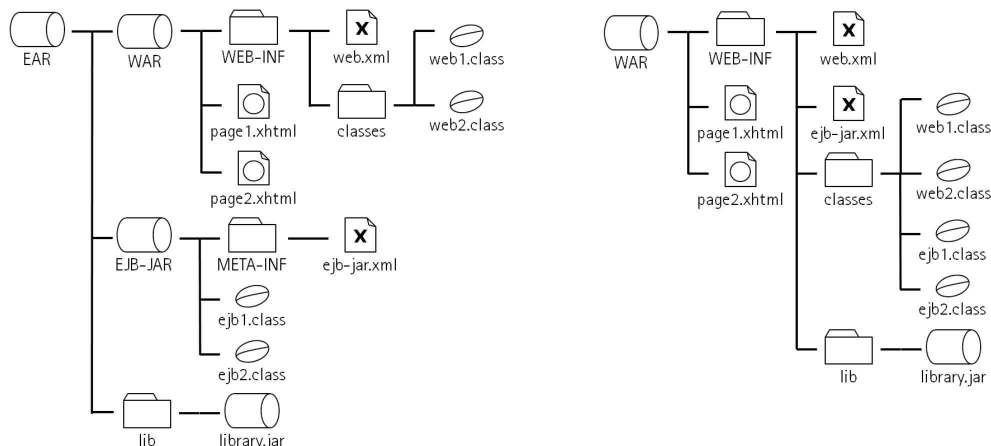


Abbildung 6.1: EAR und WAR als Deployment-Formate für EJBs

6.4 Lokaler Zugriff auf Session Beans

Session Beans sind Managed Beans im Sinne von CDI. Dementsprechend können ihre Instanzen wie gewohnt mittels *@Inject* in andere CDI Beans oder auch Session Beans injiziert werden. Historisch bedingt hält die EJB-Spezifikation zu diesem Zweck auch eine Injektionsannotation bereit: *@EJB*[9] bewirkt das Gleiche wie *@Inject*, allerdings eingeschränkt auf EJBs. Für neue Anwendungen sollte ausschließlich *@Inject* verwendet werden.

Local Interface

Ebenfalls historisch begründet ist die Möglichkeit, für eine EJB ein Local Interface zu definieren. Darunter versteht man ein einfaches Interface, das üblicherweise von der EJB-Klasse implementiert wird und mit *@Local*[10] annotiert ist. Alternativ ist auch die Annotation der Klasse mit *@Local* mit der Interfaceklasse als Parameter möglich (Listing 6.3).

9 *javax.ejb.EJB*
10 *javax.ejb.Local*

```
@Local
public interface ShopServiceLocal
{
  public List<ShopItem> getAllShopItems();
  …
}

@Stateful
// Alternativ hier statt im Interface: @Local(ShopServiceLocal.class)
public class ShopServiceBean implements ShopServiceLocal
{
  public List<ShopItem> getAllShopItems() { … };
  …
}
```

Listing 6.3: EJB mit Local Interface

Das Local Interface umfasst eine ggf. echte Teilmenge der von der Bean angebotenen Methoden. Ist ein Local Interface vorhanden, definiert es mit seinen Vaterinterfaces die CDI Bean Types der EJB. Somit können nur Variablen oder Parameter dieser Interfacetypen Injektionsziele sein. Damit sind dann auch nur die Methoden des jeweiligen Interfaces erreichbar.

No-Interface View

In der Praxis hat das Local Interface seinen Wert verloren, seit es mit EJB 3.1 und CDI möglich ist, die Bean-Klasse direkt zu verwenden, d. h. als Typ des Injektionsziels zu benutzen. Zudem enthielten auch schon früher Local Interfaces meist nicht nur einen Teil, sondern alle öffentlichen Methoden der Bean. Viele Softwareentwickler haben sie daher zu Recht als überflüssigen Ballast im Programmcode empfunden.

Als Alternative kann der direkte Zugriff auf die EJB über die sog. No-Interface View verwendet werden. Abgesehen von dem auch hier transparent genutzten Hüllobjekt verbirgt sich hinter der No-Interface View einfach nur die Klasse selbst. Die View muss allerdings eingeschaltet werden, was durch Annotation der Klasse mit *@LocalBean*[11] geschieht. Für Beans ohne Local und Remote Interfaces ist *@LocalBean* implizit vorhanden.

Listing 6.4 zeigt eine Bean mit No-Interface View sowie den Zugriff mittels Injektion in eine beliebige andere Bean. Die beiden aufgeführten Alternativen – Zugriff über No-Interface View bzw. Local Interface – sind gleichwertig, wenn man voraussetzt, dass das Local Interface alle *public*-Methoden der Bean umfasst.

```
@Stateful
@LocalBean
public class ShopServiceBean implements ShopServiceLocal
{
```

11 *javax.ejb.LocalBean*

entwickler.press

```
  …
}

public class SomeOtherBean
{
  @Inject
  private ShopServiceBean shopServiceBean; // Für Zugriff via No-IF View

  @Inject
  private ShopServiceLocal shopServiceLocal; // Für Zugriff via Local IF
  …
}
```

Listing 6.4: Bean mit No-Interface View und beispielhafter Zugriff

6.5 Remote-Zugriff

Anders als CDI Beans können Session Beans auch aus einem anderen Java-Prozess heraus aufgerufen werden. Dazu ist zwingend die Definition eines Remote Interfaces notwendig, das einerseits die Menge der aufrufbaren Methoden bestimmt und andererseits auf der Clientseite die Erstellung eines Stubs, d. h. eines dynamischen Proxies zum Aufruf der serverseitigen Methoden ermöglicht. Dieses Stub-Objekt implementiert wie die Bean das Remote Interface, leitet aber alle Methodenaufrufe serialisiert zur echten Ausführung auf den Server weiter.

Remote Interface

Das Remote Interface ist wieder ein einfaches Interface der Bean-Klasse, das diesmal mit *@Remote*[12] annotiert wird. Auch hier ist alternativ die Annotation der Bean-Klasse mit *@Remote* und der Interfaceklasse als Parameter erlaubt.

Bei der Gestaltung des Remote Interfaces muss man berücksichtigen, dass alle Methodenaufrufe über eine Netzwerkverbindung zwischen Client und Server übertragen werden müssen (das sog. Marshalling). Dadurch bedingt müssen alle verwendeten Typen für Parameter, Return-Werte und Exceptions serialisierbar sein (Listing 6.5).

```
@Remote
public interface ShopService
{
  public List<ShopItem> getAllShopItems();
  …
}

@Stateful
```

12 *javax.ejb.Remote*

```
@LocalBean
// Alternativ hier statt im Interface: @Remote(ShopService.class)
public class ShopServiceBean implements ShopService
{
  public List<ShopItem> getAllShopItems() { … };
  …
}

public class ShopItem implements Serializable
{
  …
}
```

Listing 6.5: EJB mit Remote Interface

Eintrag von EJBs im Namensdienst des Servers

Wird eine Anwendung auf einem Application Server deployt, so werden die enthaltenen EJBs – genauer: ihre Proxy-Objekte – im Namensdienst des Servers eingetragen. Dieser JNDI-basierte Dienst speichert Objekte unter einem hierarchischen Namen ab, ähnlich den Dateien in einem Dateisystem. EJBs werden unter einem Namen der folgenden Form abgelegt: *java:global/applicationName/moduleName/beanName!interfaceName*

Die Bestandteile darin sind:

- *applicationName*: Name der Gesamtapplikation.

Der Name der Enterprise-Anwendung ohne die Endung *.ear* bzw. ein im Deployment Descriptor *application.xml* angegebener Name. Ist die EJB nicht Teil eines EAR-Files, entfällt dieser Teil des Namens inkl. des darauf folgenden Trennzeichens /.

- *moduleName*: Name des Anwendungsmoduls.

Die EJB kann Bestandteil eines EJB-JAR-Files oder eines WAR-Files sein, entweder separat deployt oder als Teil einer Enterprise-Anwendung. Der Name ergibt sich aus dem Dateinamen ohne Endung *.jar* bzw. *.war*, kann aber auch im Descriptor *ejb-jar.xml* bzw. *web.xml* explizit angegeben werden.

- *beanName*: Name der EJB.

Das ist normalerweise der einfache Klassenname der Bean. Er kann allerdings auch mithilfe des Parameters name der Annotationen *@Stateless, @Stateful* etc. angegeben werden.

- *interfaceName*: Name des implementierten Interfaces.

Letzter Bestandteil des JNDI-Namens ist der voll qualifizierte Name des vom Proxy-Objekt implementierten Interfaces, bzw. im Fall der No-Interface View der Bean-Klasse.

Wenn die EJB nur eine Sicht implementiert, d. h. entweder nur die No-Interface View oder nur genau ein Local bzw. Remote Interface vorhanden ist, wird zusätzlich zum beschrie-

benen JNDI-Eintrag ein weiterer angelegt, der nur die Teile bis einschließlich *beanName* enthält. Die EJBs aus Listing 6.1 und Listing 6.5 hätten somit die folgenden JNDI-Einträge, wenn sie in einer Webanwendung namens *ee-demos-ejb.war* deployt würden:

- *java:global/ee-demos-ejb/ShopItemRepository!de.gedoplan.….ShopItemRepository*
- *java:global/ee-demos-ejb/ShopItemRepository*
- *java:global/ee-demos-ejb/ShopServiceBean!de.gedoplan.….ShopServiceBean*
- *java:global/ee-demos-ejb/ShopServiceBean!de.gedoplan.….ShopService*

Die ersten drei nur lokal wirkenden Einträge sind nur dann relevant, wenn nicht die beschriebenen Injektionsmöglichkeiten genutzt werden sollen oder können. Im Folgenden wird nur der letzte Eintrag betrachtet, da er sich auf ein Remote Interface bezieht.

Remote Lookup und clientseitige Nutzung von EJBs

Für den Zugriff auf eine serverseitige EJB muss das Clientprogramm zunächst eine Verbindung zum JNDI-Dienst des Servers öffnen. Das Verfahren dazu ist abhängig vom eingesetzten Application Server, umfasst aber zumeist die Erzeugung eines Objekts des Typs InitialContext, wobei einige Verbindungsparameter in einer Classpath-Ressource namens jndi.properties übergeben werden. Details entnehmen Sie bitte der Dokumentation des von Ihnen genutzten Servers. Das Begleitprojekt zu diesem Buchkapitel enthält beispielhafte Verbindungsdaten für GlassFish 3 sowie JBoss 6 und 7.

Mithilfe der Methode lookup kann das Proxy-Objekt zur gewünschten Bean unter Angabe ihres Lookup-Namens zum Client transferiert werden. Mit ihm kann schließlich auf die serverseitigen Methoden zugegriffen werden (Listing 6.6).

```
public class ShopServiceTest
{
  …
  // Verbindung zum JNDI öffnen (nutzt Ressource jndi.properties)
  Context jndiCtx = new InitialContext();

  // Lookup der gewünschten Bean
  String lookupName
  = "java:global/ee-demos-ejb/ShopServiceBean!de.gedoplan.….ShopService";
  ShopService shopService = (ShopService) jndiCtx.lookup(lookupName);

  // Remote-Aufruf der Methoden der Bean
  List<ShopItem> shopItems = shopService.getAllShopItems();
  …
}

# jndi.properties für GlassFish 3.1.1
java.naming.factory.initial=\
  com.sun.enterprise.naming.SerialInitContextFactory
```

Listing 6.6: Clientseitige Programmsequenz zum Zugriff auf eine EJB

Für Stateful Session Beans wird durch den Lookup eine Session für das betroffene Objekt gestartet, die so lange andauert, bis für das Objekt eine mit *@Remove* annotierte Methode aufgerufen wird. Um danach eine neue Session zu starten, wird ein erneuter Lookup benötigt (Listing 6.7).

```
ShopService shopService = (ShopService) jndiCtx.lookup(lookupName);
… // erste Session
shopService.close();
…
shopService = (ShopService) jndiCtx.lookup(lookupName);
… // zweite Session
shopService.close();
```

Listing 6.7: Sitzungssteuerung durch Lookup und Remove-Methoden

6.6 Transaktionssteuerung

Im Kapitel über CDI wurde gezeigt, das CDI Beans zunächst keine Transaktionssteuerung vornehmen, diese aber mithilfe eines Interceptors hinzugefügt werden kann. Bei EJBs findet man die umgekehrte Situation vor: EJBs übernehmen die Steuerung von Transaktionen, wenn dies nicht durch explizite Konfiguration deaktiviert wird.

Transaction Management und Transaction Attribute

Die Art der Transaktionssteuerung wird durch Annotationen bestimmt. Zunächst kann mit *@TransactionManagement*[13] auf Klassenebene zwischen deklarativem und programmatischem Transaktionsmanagement gewählt werden:

- *@TransactionManagement(TransactionManagementType.CONTAINER)* ist die Vorgabeeinstellung. Hier werden Transaktionen durch den Container gesteuert, wobei das Verfahren durch die weiter unten beschriebene Annotation *@TransactionAttribute* deklariert wird.
- *@TransactionManagement(TransactionManagementType.BEAN)* wählt dagegen den programmatischen Weg: in den Bean-Methoden ist es hier erlaubt, direkte Transaktionssteuerung auszuüben (mithilfe einers Objekts vom Typ *UserTransaction*).

Nur mit dem ersten Verfahren bieten EJBs gegenüber CDI Beans im Hinblick auf die Transaktionssteuerung einen Mehrwert. Daher soll im Folgenden nur dieser Ansatz betrachtet werden. Hier kann mit einer weiteren Annotation – *@TransactionAttribute*[14] – auf Klassen- oder Methodenebene gewählt werden, wann eine Transaktion begonnen und abgeschlossen werden soll:

13 *javax.ejb.TransactionManagement*
14 *javax.ejb.TransactionAttribute*

- *@TransactionAttribute(TransactionAttributeType.REQUIRED)*

Das ist die Voreinstellung. Wird eine so annotierte Methode aufgerufen und ist noch keine Transaktion aktiv, so wird eine neue gestartet. Endet die Methode mit *return*, wird die Transaktion mit einem Commit abgeschlossen. Bei Auswurf einer System Exception wird dagegen ein Rollback durchgeführt. Für Application Exceptions kann der Transaktionsausgang gewählt werden. Das und auch die Einordnung der Exceptions in System und Application Exceptions wird weiter unten erläutert.

- *@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)*

Bei diesem Modus wird beim Methodenaufruf stets eine neue Transaktion gestartet. Die weitere Vorgehensweise entspricht dem *REQUIRED*-Modus. Sollte bei Aufruf bereits eine Transaktion aktiv sein, wird sie für die Dauer des Aufrufs suspendiert. Die neue Transaktion wird dadurch nicht eingeschachtelt. Vielmehr sind äußere und neue Transaktion in ihren Ergebnissen nicht voreinander abhängig.

- *@TransactionAttribute(TransactionAttributeType.MANDATORY)*

Eine so annotierte Methode darf nur aufgerufen werden, wenn eine aktive Transaktion vorhanden ist. Andernfalls wird eine Exception[15] ausgeworfen.

- *@TransactionAttribute(TransactionAttributeType.NEVER)*

Das Gegenstück zu *MANDATORY*: Die Methode darf nur ohne aktive Transaktion aufgerufen werden, sonst wird wiederum eine Exception[16] ausgeworfen.

- *@TransactionAttribute(TransactionAttributeType.SUPPORTS)*

Eine so annotierte Methode darf ohne und mit aktiver Transaktion aufgerufen werden.

- *@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)*

Hier wird eine beim Aufruf aktive Transaktion für die Dauer der Methodenausführung suspendiert.

Tabelle 6.1 fasst die Auswirkungen der Transaktionsattribute auf die bei Methodenausführung genutzte Transaktion zusammen.

Transaction Attribute	Transaktion bei Aufruf	Transaktion während Methodenausführung
REQUIRED	keine T1	T2 T1
REQUIRES_NEW	keine T1	T2 T2

15 *javax.ejb.EJBTransactionRequiredException*, *javax.ejb.EJBTransactionRequiredLocalException* oder *javax.transaction.TransactionRequiredException* je nach Aufrufkontext. Details siehe Spezifikation.

16 *javax.ejb.EJBException* oder *java.rmi.RemoteException* je nach Aufrufkontext. Details siehe Spezifikation.

Transaction Attribute	Transaktion bei Aufruf	Transaktion während Methodenausführung
MANDATORY	keine T1	- (Exception) T1
NEVER	keine T1	keine - (Exception)
SUPPORTS	keine T1	keine T1
NOT_SUPPORTED	keine T1	keine keine

Tabelle 6.1: Auswirkung der Transaktionsattribute auf die Transaktionspropagierung

Application und System Exceptions

Endet ein Methodenaufruf in einem der beiden Automatikmodi *REQUIRED* und *REQUIRES_NEW* mit einer Exception, hängt es von deren Typ ab, ob der Container ein Commit oder Rollback durchführt.

- Application Exceptions dienen dazu, solche Fehler aus der Anwendungslogik an den Aufrufer zu melden, die er korrigieren kann, wie bspw. die Übergabe einer unbekannten Artikelnummer oder eines zu hohen Betrags. Hierbei handelt es sich um Checked Exceptions[17] oder aber um Unchecked Exceptions, die mit *@ApplicationException*[18] annotiert sind.
- System Exceptions melden i. d. R. schwerwiegende Fehler aus der Anwendungslogik, die meistens auf einen Fehler eines Systemteils zurückgehen, z. B. auf das Fehlen einer Datenbanktabelle. Der Aufrufer kann hier bis auf eine Fehleranzeige oder Protokollierung keine Fehlerbehandlung durchführen. System Exceptions sind alle mit Ausnahme der Application Exceptions.

Endet die betreffende Methode mit einer System Exception, wird ein Rollback durchgeführt. Bei einer Application Exception kommt es dagegen zum Commit, wenn die Exception nicht mit *@ApplicationException(rollback=true)* annotiert ist.

Der Vollständigkeit halber sei noch erwähnt, dass ein Rollback auch durch Aufruf der Methode *EJBContext.setRollbackOnly* erzwungen werden kann. Ein Objekt des Typs *EJBContext*[19] lässt sich mithilfe von *@Resource* bspw. in eine Instanzvariable der EJB injizieren.

17 mit Ausnahme von *java.rmi.RemoteException* oder davon abgeleiteten Klassen
18 *javax.ejb.ApplicationException*
19 *javax.ejb.EJBContext*

6.7 Asynchrone Methoden

Geschäftslogik kann so umfangreich sein, dass ihre Abarbeitung mehr Zeit benötigt als man den Aufrufer normalerweise blockieren möchte. Da Methodenaufrufe in Java synchron ausgeführt werden, könnte in solchen Fällen auf asynchrone Kommunikation mit Java Messaging o. ä. ausgewichen werden. Enterprise JavaBeans bieten ab der Version 3.1 eine elegante Alternative. Langlaufende Methoden können hier mit @*Asynchronous*[20] annotiert werden (Listing 6.8).

```
@Asynchronous
public void archiviereBelege(int jahr) { … }
```

Listing 6.8: Definition einer Methode mit asynchroner Ausführung

Der Aufrufer der Methode erhält sofort die Kontrolle zurück, während die Geschäftslogik der Bean parallel abgearbeitet wird. Die Annotation kann für die gewünschte Methode der EJB-Klasse verwendet werden, oder aber bei der Deklaration der Methode im Remote oder Local Interface. Somit kann eine Methode bspw. für einen Remote-Aufruf asynchron ablaufen, während sie lokal synchron aufgerufen wird.

Meistens liefern asynchrone Methoden wie im Beispiel keinen Ergebniswert, da sich der Aufrufer nach dem Anstoß der Methode nicht mehr für ihre Ausführung interessiert. Es ist jedoch auch möglich, einen Wert zurückzuliefern, und zwar in Form eines *Future*[21]-Objekts. Für diesen Zweck enthält der Standard die Klasse *AsyncResult*[22] als konkrete Implementierung des Interfaces *Future*. Der Aufrufer der EJB-Methode erhält wiederum sofort die Kontrolle zurück und kann das Ergebnis der parallelen Berechnung zu einem späteren Zeitpunkt mithilfe der Methode *get* des *Future*-Werts abholen (Listing 6.9).

```
@Stateless
public class DeepThoughtServiceBean implements DeepThoughtService
{
  @Asynchronous
  public Future<String>
          getAnswerToQuestionAboutLifeUniverseAndEverything()
  {
    … // Time passes (7.5 million years …?)
    return new AsyncResult<String>("Zweiundvierzig");
  }
}

public class DeepThougthServiceTest
{
  public void testGetAnswerToQuestionAboutLifeUniverseAndEverything()
```

20 *javax.ejb.Asynchronous*
21 *java.util.concurrent.Future*
22 *javax.ejb.AsyncResult*

```
    throws Exception
  {
    …
    DeepThoughtService deepThoughtService
      = (DeepThoughtService) jndiCtx.lookup(lookupName);

    Future<String> futureAnswer = deepThoughtService
                  .getAnswerToQuestionAboutLifeUniverseAndEverything();
    … // Berechnung läuft parallel
    String answer = futureAnswer.get();
    …
  }
}
```

Listing 6.9: Asynchroner Methodenaufruf mit Ergebnis

6.8 Timer

In vielen Anwendungen gibt es einen Bedarf für zeitgesteuerte Tätigkeiten, sei es zur periodischen Überwachung von externen Ereignissen oder für regelmäßige, termingesteuerte Aufgaben. Dafür existiert für EJBs mit Ausnahme der Stateful Beans ein Timer-Service mit sehr umfangreichen Scheduling-Möglichkeiten. Eine Bean kann eine oder mehrere Methoden enthalten, die mit *@Schedule*[23] annotiert sind. Über die Parameter der Annotation können die Aufruftermine sehr flexibel spezifiziert werden, wie die Beispiele in Listing 6.10 zeigen.

```
@Stateless
public class SchedulerServiceBean implements SchedulerService
{
  @Schedule(second = "0/10", minute = "*", hour = "*",
           persistent = false)
  private void alle10Sekunden() { … }

  @Schedule(dayOfWeek = "Mon")
  private void montags() { … }

  @Schedule(minute = "15", hour = "3", dayOfWeek = "Mon-Fri")
  private void wochentagsUm0315() { … }

  @Schedule(minute = "*/5", hour = "*")
  private void alle5Minuten() { … }

  @Schedule(dayOfMonth = "Last")
  private void ultimo() { … }
```

23 *javax.ejb.Schedule*

```
  @Schedule(dayOfMonth = "2nd Fri", hour = "20")
  private void jedenZweitenFreitagUm2000() { … }
  …
}
```

Listing 6.10: Periodische Methodenausführung mittels „@Schedule“

Die Methoden dürfen parameterlos sein oder einen Parameter vom Typ *Timer*[24] annehmen. Der übergebene Wert enthält Daten über den ausgelösten Timer. Details dazu finden Sie in der Dokumentation der Klasse.

Die Timer sind standardmäßig persistent, können aber mithilfe des Annotationsparameters *persistent* auch nichtpersistent gestaltet werden. Persistente Timer bleiben über einen Shutdown der Applikation hinaus gültig. Während der Downtime verpasste Timer-Events werden automatisch nachgeholt, wenn die Anwendung wieder gestartet wird.

Zusätzlich zur Deklaration durch Annotationen können Timer auch programmgesteuert gestartet und gestoppt werden. Den Einstieg dazu bildet ein Objekt des Typs *TimerService*[25], das mittels *@Resource* bspw. in eine Instanzvariable der Bean injiziert werden kann. Hierüber sind diverse Methoden zum Erzeugen von Timern erreichbar sowie eine Methode zum Abfragen aller Timer der Bean. Jeder einzelne davon kann mithilfe seiner Methode *cancel* gestoppt werden.

Die mittels TimerService erzeugten Timer rufen zu den gewünschten Zeitpunkten eine Methode der Bean auf, die mit *@Timeout*[26] annotiert ist.

Listing 6.11 zeigt beispielhaft die Nutzung des Timer Service.

```
@Stateless
public class SchedulerServiceBean implements SchedulerService
{
  @Resource
  private TimerService timerService;

  public void startTimers()
  {
    …
    this.timerService.createSingleActionTimer(
      3000, new TimerConfig("einmalig nach 3 Sekunden", false));

    this.timerService.createIntervalTimer(
      0, 1000, new TimerConfig("sekuendlich", false));

    this.timerService.createCalendarTimer(
      new ScheduleExpression().second("0/2").minute("*").hour("*"),
```

24 *javax.ejb.Timer*
25 *javax.ejb.TimerService*
26 *javax.ejb.Timeout*

```
        new TimerConfig("alle 2 Sekunden", false));
    }

    public void stopTimers()
    {
      Collection<Timer> allTimers = this.timerService.getTimers();
      for (Timer timer : allTimers)
      {
        timer.cancel();
      }
    }

    @Timeout
    public void tick(Timer timer) { … }
    …
  }
```

Listing 6.11: Nutzung von „TimerService“ zum Erzeugen und Stoppen von Timern

6.9 Security

Für den Aufruf von EJB-Methoden wird ein ähnliches Security-Konzept genutzt, wie es im JSF-Kapitel für die Views einer Webanwendung beschrieben wurde: Mithilfe deklarativer Security kann der Aufruf von EJB-Methoden nur solchen Usern gestattet werden, die bestimmten Security-Rollen zugeordnet sind. Zudem ist es möglich, die Rollenzugehörigkeit im Programm abzufragen.

Deklarative Security

Um den Aufruf einer EJB-Methode nur bestimmten Benutzern zu erlauben, muss die Methode mit einer der folgenden Annotationen versehen werden:

- *@RolesAllowed*[27]: Zugriffserlaubnis nur für bestimmte Security-Rollen.

Der Parameter der Annotation bestimmt, welche Rollen Erlaubnis erhalten. Der aufrufende User muss mindestens einer der Rollen zugeordnet sein.

- *@PermitAll*[28]: Zugriffserlaubnis für alle Benutzer.

Dies ist die Voreinstellung.

- *@DenyAll*[29]: Zugriffsverbot für alle Benutzer.

Die Annotationen können auch für die Bean-Klasse genutzt werden. Sie gelten dann für die Methoden ohne eigene Security-Annotation. Listing 6.12 zeigt ein Beispiel.

27 *javax.annotation.security.RolesAllowed*
28 *javax.annotation.security.PermitAll*
29 *javax.annotation.security.DenyAll*

entwickler.press

```
@Stateless
@PermitAll
public class SecurityDemoServiceBean
{
  …
  @RolesAllowed("demoRole")
  public void restrictedMethod()
  {
    …
```

Listing 6.12: Beschränkung der Zugriffe auf Methoden für bestimmte Nutzer

Wird eine Methode von einem User aufgerufen, der nicht die benötigte Rollenzuordnung hat, wird der Aufruf vom Container durch Auswerfen einer Exception des Typs *EJBAccessException*[30] abgelehnt.

Statt der Annotationen können auch Einträge im Deployment Descriptor *ejb-jar.xml* gemacht werden. Details dazu finden Sie in der EJB-Spezifikation im Abschnitt 17.3.2.2, Specification of Method Permissions in the Deployment Descriptor.

Für einige Application Server sind Einträge in serverspezifischen Deskriptoren nötig, um die gezeigten Security-Mechanismen nutzen zu können. So benötigt bspw. JBoss 7 im Deskriptor *jboss-ejb3.xml* für jede EJB eine Zuordnung zu der genutzten Security Realm, d. h. der serverseitigen Security-Konfiguration. Details dazu finden Sie in der Dokumentation des von Ihnen genutzten Servers. Das Begleitprojekt dieses Kapitels ist für die Nutzung der Security von GlassFish 3 und JBoss 7 vorbereitet.

Programmgestützte Security

In EJB-Methoden kann auf den aktuell aufrufenden User und seine Rollenzuordnung zugegriffen werden, und zwar mithilfe eines Objekts vom Typ *EJBContext*[31], das mittels *@Resource* bspw. in eine Instanzvariable der Bean injiziert werden kann.

Mit der Methode *EJBContext.getCallerPrincipal* kann der aktuell aufrufende User abgefragt werden. Das zurückgegebene Objekt vom Typ *Principal* enthält u. a. den Anmeldenamen des Benutzers (*Principal.getName()*). Der Wert ist nicht für eine Autorisierung des Users im Programm gedacht – dazu ist das Rollenkonzept besser geeignet. Man kann den Wert aber zur Protokollierung oder für benutzerabhängige Datenzugriffe verwenden.

Mit *EJBContext.isCallerInRole* kann abgefragt werden, ob der aufrufende User einer bestimmten Rolle zugeordnet ist. In der Kombination mit weiteren Anwendungsdaten kann so eine umfangreichere, kontextabhängige Autorisierung durchgeführt werden, die alleine mit den deklarativen Möglichkeiten nicht erreichbar wäre.

30 *javax.ejb.EJBAccessException*
31 *javax.ejb.EJBContext*

Listing 6.13 zeigt einen Ausschnitt aus einer EJB, die die genannten Methoden nutzt. Darin deutet die Methode *partlyRestrictedMethod* eine kontextabhängige Berechtigungssteuerung an.

```
@Stateless
public class SecurityDemoServiceBean
{
  @Resource
  private EJBContext ejbCtx;

  public String getCallerName()
  {
    Principal caller = this.ejbCtx.getCallerPrincipal();
    return caller == null ? "unauthenticated" : caller.getName();
  }

  public boolean isCallerInDemoRole()
  {
    return this.ejbCtx.isCallerInRole("demoRole");
  }

  @RolesAllowed("eeDemoUser")
  public void partlyRestrictedMethod(double amount)
  {
    if (amount >= 10000)
      if (!this.ejbCtx.isCallerInRole("demoRole"))
        throw new IllegalArgumentException(
         "Big amounts need role demoRole");
  }
  …
```

Listing 6.13: Nutzung der Security-Informationen im Code einer EJB

Die Security von EJBs kann noch weiter konfiguriert bzw. beeinflusst werden. So können bspw. Methoden mit anderen Rollen ausgeführt oder auch die in einer EJB genutzten Rollen anderen Rollen zugeordnet werden. Sie finden Informationen dazu in Kapitel 17, Security Management, der EJB-Spezifikation.

7 Ein „Real World"-Projekt

Nachdem die bisherigen Kapitel des Buches wesentliche Bausteine der Plattform Java EE 6 beschrieben haben, soll nun ein Projekt beschrieben werden, das sich dieser Bausteine bedient. Es handelt sich dabei um eine Anwendung, die für den Einsatz in der Verwaltung der GEDOPLAN GmbH gedacht ist, die also neben dem Beispielcharakter auch dem „Real World"-Anspruch gerecht werden soll. Aus Gründen der Übersichtlichkeit wurde die Anwendung allerdings auf einen Teil reduziert, der dem Ziel des Buchs entspricht.

7.1 Aufgabenstellung

Die Anwendung „ProVS" – „Projektverwaltungssystem" – realisiert eine Projektverwaltung, d. h. eine Verwaltung von Beratungs- oder Entwicklungsprojekten inklusive der dazu nötigen Stammdaten sowie der Erfassung von Projektzeiten.

Projekthintergrund

Das Geschäft von GEDOPLAN ist projektgetrieben. Im Normalfall beauftragen uns unsere Kunden mit der Unterstützung in Projekten im Hause des Kunden oder auch bei uns. Die dafür eingesetzten Mitarbeiter erfassen mit der Anwendung ihre Projektzeiten und Reisekosten. Daraus betreibt das Projektbüro die Fakturierung sowie die Erstellung monatlicher Übersichten.

Ein Projekt wird recht früh, ggf. schon vor dem ersten Kundenkontakt, vom Vertrieb angelegt. Das kann insbesondere noch vor der Unterbreitung eines Angebots oder der Beauftragung durch den Kunden erfolgen.

Kommt es zum Auftrag, werden dem Projekt Aufgaben und Mitarbeiter zugeordnet. Während der Durchführungszeit buchen die Mitarbeiter Projektzeiten auf ihre Projekte. Im Projektbüro werden die erfassten Daten monatlich zur Fakturierung genutzt.

Anwendungsdomänen

Die zugrunde liegenden Daten lassen sich grob in zwei Domänen einteilen:

- Domäne *Firma*: Stammdaten der Auftraggeber, Auftragnehmer und Mitarbeiter.
  - *Firma*, *Standort*: Stammdaten von Firmen. Diese können mehrere Standorte haben.
  - *Person*, *Mitarbeiter*: Stammdaten der zugehörigen Personen. *Person* bildet dabei den privaten Aspekt ab, während *Mitarbeiter* die Verknüpfung zu einer Firma herstellt.
  - *Adresse*, *Land*: Adressdaten.

- Domäne *Projekt*: Stammdaten der Projekte sowie erfasste Projektzeiten.
  - *Projekt*, *Aufgabe*: Stammdaten der Projekte. Sie stellen jeweils eine Aufgabe dar, die beliebige Unteraufgaben haben kann.
  - *MitarbeiterAufgabe*: Zuordnung von Mitarbeitern zu Aufgaben.
  - *ProjektZeit*: Erfasste Projektzeiten.

Die Domäne *Projekt* nutzt Elemente der Domäne *Firma*, jedoch nicht umgekehrt. Dadurch wird eine Wiederverwendung von *Firma* in einem anderen Zusammenhang – bspw. einem CRM-System – ermöglicht. Abbildung 7.1 und Abbildung 7.2 zeigen die Klassendiagramme der beiden Domänen.

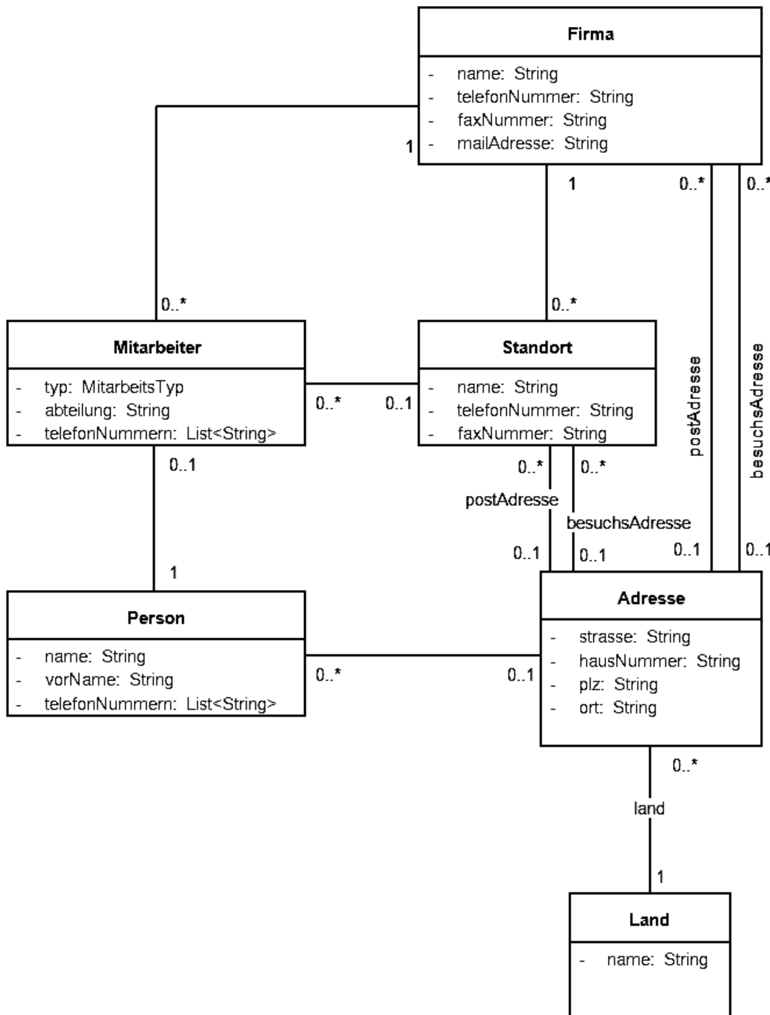


Abbildung 7.1: Klassendiagramm der Domäne „Firma“

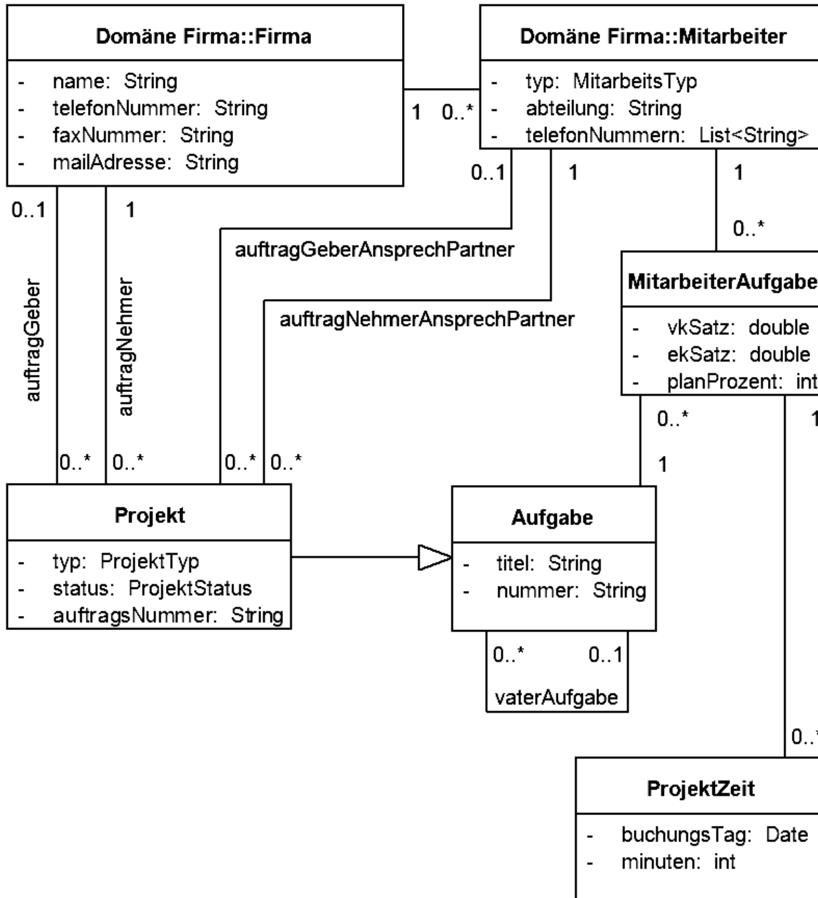


Abbildung 7.2: Klassendiagramm der Domäne „Projekt"

7.2 Anwendungsarchitektur

Die Anwendung ist als Webanwendung mit JSF und Facelets für die Präsentation aufgebaut. Die Logik der Anwendung befindet sich in CDI Beans, wobei eine saubere Schichtentrennung eingehalten wird:

- Die Präsentationslogik befindet sich in Model-Objekten, d. h. CDI Beans mit dem Stereotype *@Model*. Hier ist nur die Aufbereitung der Daten für die Präsentation und die Statusverwaltung der Benutzeroberfläche enthalten.
- Für die Geschäftslogik sind CDI Beans mit dem Stereotype *@DomainService* vorgesehen. Hier sind über den reinen Datenzugriff hinausgehende Geschäftsprozessteile ent-

halten, z. B. umfangreichere Datenzusammenstellungen oder auch Zugriffe auf technische Dienste der Serverinfrastruktur.

- Der Zugriff auf die Datenbank geschieht mit Repository-Objekten. Dahinter verbergen sich CDI Beans mit dem Stereotype *@DataRepository.* Sie implementieren die CRUD-Funktionalitäten sowie die benötigten Find-Operationen auf Basis von Java Persistence.

Zwischen diesen Anwendungsschichten erfolgen die Zugriffe nur in einer Richtung: Model –> Service –> Repository. Anders als in manchen anderen Beispielen ist hier aber ein direkter Zugriff der Models auf die Repositories erlaubt. Andernfalls müssten die Repository-Methoden überwiegend in der Serviceschicht dupliziert werden, ohne dass damit ein struktureller Vorteil verbunden wäre.

Zugriffe der Klassen der Projektdomäne auf die Firmendomäne sind mit den gleichen Einschränkungen erlaubt, nicht aber umgekehrt (Abb. 7.3).

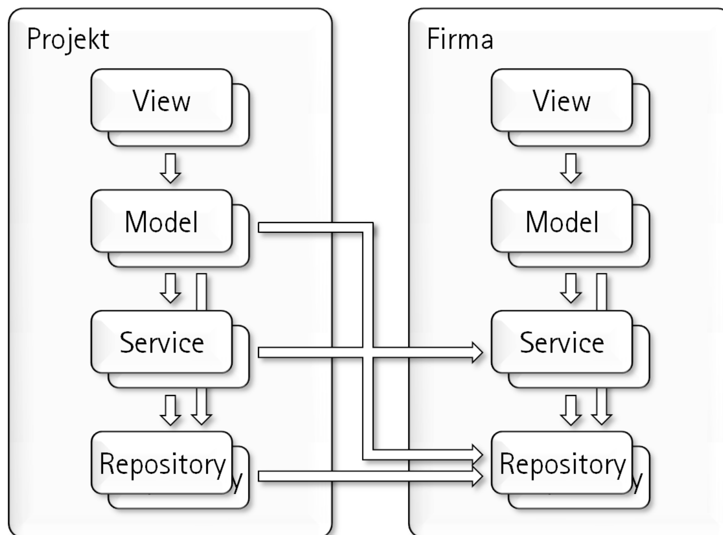


Abbildung 7.3: Erlaubte Zugriffspfade

Bei der in Abbildung 7.3 gezeigten Abhängigkeit der Repositories der beiden Komponenten untereinander müssen die Relationen der verwendeten Entity-Klassen mit berücksichtigt werden: Eine Entity der Domäne *Projekt* darf also eine unidirektionale Relation zu einer *Firma*-Klasse besitzen, während bidirektionale oder umgekehrt gerichtete Relationen nicht erlaubt sind. Eine weitergehende Abhängigkeit der Repository-Klassen untereinander tritt in der Praxis kaum auf, da die Klassen über den reinen DB-Zugriffscode hinaus keinerlei Logik enthalten.

Die skizzierte Struktur der Anwendung spiegelt sich in der Paketstruktur wider: Die beiden Komponenten befinden sich in den Paketen *de.gedoplan.buch.eedemos.provs.firma* bzw. *de.gedoplan.buch.eedemos.provs.projekt*. Darin sind jeweils Unterpakete *model*, *service* und

repository enthalten. Die Anwendung ist als einzelne Webanwendung aufgebaut, sodass eine wirkliche Trennung der Komponenten untereinander nicht stattfindet. Es wäre aber auch möglich gewesen, die beiden Komponenten als eigenständige Jar-Files zu paketieren und die Gesamtanwendung als Enterprise-Applikation aufzubauen.

7.3 Persistenz

Entities

Wie im Kapitel über Java Persistence dargestellt, müssen Entity-Klassen einige wenige Voraussetzungen erfüllen, damit sie mithilfe eines Persistence Providers verarbeitet werden können, u. a.:

- Annotation als persistente Klasse (*@Entity* oder *@MappedSuperclass*)
- Vorhandensein eines identifizierenden Attributs

In der Praxis stellt es sich als vorteilhaft heraus, wenn einige weitere Eigenschaften vorhanden sind:

- Verzicht auf mehrere ID-Attribute[1]
- Implementierung von *equals* und *hashCod*e auf Basis des ID-Attributs
- Vorhandensein eines Versionsattributs
- Implementierung von *toString*
- Serialisierbarkeit

Die genannten Eigenschaften lassen sich gut in eine generische Basisklasse auslagern (Listing 7.1[2]).

```
@MappedSuperclass
@Access(AccessType.FIELD)
public abstract class SingleIdEntity<K> implements Serializable
{
  @Version
  private long updateCount;

  public abstract K getId();

  public int hashCode()
  {
```

1 Zusammengesetzte IDs lassen sich als Embedded ID formulieren

2 Den in diesem Kapitel gezeigten Beispielcode finden Sie in den Begleitprojekten *ee-demos-baselibs* und *ee-demos-provs*. *ee-demos-baselibs* enthält dabei die Klassen, die allgemeiner Natur sind, also nicht nur in der Projektverwaltung Anwendung finden können.

```
    K thisId = getId();
    return thisId != null ? thisId.hashCode() : 0;
  }

  public boolean equals(Object obj) { … }

  public String toString() { … }
}
```

Listing 7.1: Abstrakte Basisklasse für alle Entity-Klassen

Von dieser sehr allgemeinen Entity-Basisklasse können spezialisierte abgeleitet werden, um bestimmte Typen der IDs oder bestimmte ID-Generatoren einzusetzen. ProVS nutzt in dieser Weise je eine Basisklasse für Entities mit *String*-basierter ID und für Entities mit generierter *Integer*-ID (Listing 7.2).

```
@MappedSuperclass
@Access(AccessType.FIELD)
public abstract class StringIdEntity extends SingleIdEntity<String>
{
  @Id
  protected String id;

  protected StringIdEntity() { }
  protected StringIdEntity(String id) { this.id = id; }

  public String getId() { return this.id; }
}

@MappedSuperclass
@Access(AccessType.FIELD)
public abstract class GeneratedIntegerIdEntity
  extends SingleIdEntity<Integer>
{
  @Id
  @GeneratedValue(strategy = GenerationType.TABLE,
                  generator = "IntegerIdGenerator")
  @TableGenerator(name = "IntegerIdGenerator", allocationSize = 1000)
  protected Integer id;

  public Integer getId() { return this.id; }
}
```

Listing 7.2: Entity-Basisklassen für String-IDs und generierte Integer-IDs

Der Gewinn der Bereitstellung solcher allgemeiner Entity-Basisklassen zeigt sich zunächst bei der Implementierung konkreter Entity-Klassen. Hier müssen nur noch die jeweiligen fachlichen Attribute berücksichtigt werden (Listing 7.3).

```
@Entity(name = "Land")
@Access(AccessType.FIELD)
@Table(name = Land.TABLE_NAME)
public class Land extends StringIdEntity
{
  public static final String TABLE_NAME = "PROVS_LAND";

  @NotEmpty
  private String name;
  …
}

@Entity(name="Firma")
@Access(AccessType.FIELD)
@Table(name = Firma.TABLE_NAME)
public class Firma extends GeneratedIntegerIdEntity
{
  public static final String TABLE_NAME = "PROVS_FIRMA";

  @NotEmpty
  private String name;
  …
}
```

Listing 7.3: Implementierung konkreter Entity-Klassen auf Basis der allgemeinen Basisklassen.

Listing 7.4: In Listing 7.3 sind zwei weitere Details zu erkennen, die sich in der Praxis bewährt haben:

- Bereitstellung des Tabellennamens mithilfe einer öffentlichen Konstante. Damit wird es leicht möglich, in Native Queries den korrekten Tabellennamen zu nutzen.
- Nutzung von Bean Validiation für Werteeinschränkungen. Das im Beispiel genutzte Constraint *@NotEmpty* ist eine Kombination von *@NotNull* und *@Size*(min=1).

Repositories

Anwendungen benötigen in der Regel viele einfache Zugriffe auf die Datenbank – neben den CRUD-Operationen sind das meist einige Finder, d. h. Methoden, die bestimmte Suchanfragen abwickeln. Diese Operationen lassen sich großenteils wiederum sehr gut in eine allgemein verwendbare Basisklasse auslagern, wobei sich hier die Vereinheitlichung der Entity-Klassen erneut positiv auswirkt.

```
public abstract class SingleIdEntityRepository
                      <K, E extends SingleIdEntity<K>>
  implements Serializable
{
  protected EntityManager entityManager;

  @Inject
```

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

  public void persist(E entity) { this.entityManager.persist(entity); }
  public E merge(E entity) { return this.entityManager.merge(entity); }
  …
  public E findById(K id) { return entityManager.find(getEntityClass(),
                                                     id); }
  public List<E> findAll() { … }
  …
```

Listing 7.5: Basisklasse für Repositories für Entity-Klassen wie in Listing 7.3

Neben den gezeigten einfachen Methoden lassen sich in einer solchen Repository-Basisklasse einige unterstützende Methoden unterbringen, die die Implementierung von häufig benötigten Find-Methoden in konkreten Repository-Klassen erleichtert. Es würde den Rahmen sprengen, diese Methoden hier abzudrucken. Sie finden die Klasse mit entsprechenden Kommentaren im Begleitprojekt.

Der benötigte Entity Manager wird mittels Injektion in der Repository-Klasse bereitgestellt. Wie in den Kapiteln über CDI und Java Persistence bereits angesprochen, ist dazu ein entsprechender Producer nötig. Darauf geht der nächste Abschnitt ein.

Nach dieser Vorarbeit ist die Bereitstellung konkreter Repository-Klassen mit wenig Aufwand möglich (Listing 7.5).

```
@DataRepository
public class LandRepository
  extends SingleIdEntityRepository<String, Land>
{
}

@DataRepository
public class FirmaRepository
  extends SingleIdEntityRepository<Integer, Firma>
{
  public Firma findByName(String name)
  {
    return findSingleByProperty(Firma_.name, name);
  }
  …
```

Listing 7.6: Implementierung konkreter Repository-Klassen

Es hat sich in der Praxis bewährt, für jede Entity-Klasse eine Repository-Klasse bereitzustellen, soweit dafür überhaupt DB-Operationen benötigt werden. Aufgrund der Trivialität der Repository-Klassen lohnt es sich nicht, mehrere Entity-Klassen gemeinsam in einem Repository zu behandeln.

Eine Anmerkung des Autors: Das von mir vorgestellt Repository-Konzept entspricht in seinen Grundzügen dem DAO[3] Pattern, allerdings angereichert um die Injektionsmöglichkeiten von CDI. Es wird in der Java Community häufig die Meinung vertreten, dass eine Kapselung des Datenbankzugriffscodes in separate Klassen nicht mehr angebracht sei, da sie im Wesentlichen ein Objekt vom Typ *EntityManager* umhüllen und ihre Methoden mittels direkter Delegation daran implementieren. In der gezeigten Form halte ich das Konzept aber weiter für tragfähig, da sich der Aufwand in sehr begrenztem Rahmen hält, die Anwendung der Klassen recht komfortabel ist und eine angenehme Trennung des technischen Datenbankzugriffscodes vom fachlichen Code der aufrufenden Geschäftslogik erzeugt.

Transaktionsgebundener Entity Manager

Wie im Kapitel über Java Persistence gezeigt, kann man mit der Annotation *@PersistenceContext* einen transaktionsgebundenen Entity Manager in CDI Beans injizieren lassen (Listing 7.6). Dieses Verfahren hat aber den Nachteil, dass der Name der Persistence Unit – hier *ee_demos* – redundant an allen Injektionsstellen im Programm vorkommt. Abhilfe schafft da die Bereitstellung einer Producers, der den Entity Manager zentral zur Injektion im CDI-Stil anbietet.

```
public class SomeRepository
{
  @PersistenceContext(name = "ee_demos")
  private EntityManager entityManager;
  …
```

Listing 7.7: Bereitstellung einer Entity-Manager-Instanz per Injektion

```
@ApplicationScoped
public class EntityManagerProducer
{
  @PersistenceContext(name = "ee_demos")
  @Produces
  private EntityManager entityManager;
}

public class SomeRepository
{
  @Inject
  private EntityManager entityManager;
  …
```

Listing 7.8: Producer für einen transaktionsgebundenen Entity Manager

Auf diese Weise lassen sich auf bequeme Weise applikationsweit alle Repository-Klassen mit einem Entity Manager versorgen. Da er transaktionsgebunden ist, werden die von ihm verwalteten Entity-Objekte am Transaktionsende automatisch in die Datenbank ge-

3 Data Access Object

schrieben. Eine übliche Stelle zur Steuerung von Transaktionen ist die Serviceschicht der Anwendung. Hier würde man die entsprechenden Methoden bspw. mit einem Transaktions-Interceptor versehen, wie er im CDI-Kapitel gezeigt wurde.

```
public class SomeService
{
  @Inject
  private SomeRepository someRepository;

  @TransactionRequired
  public void doSomething()
  {
    …
    someRepository.persist(someObject);
    …
  }
  …
```

Listing 7.9: Nutzung der Transaktionsbindung des Entity Managers.

Listing 7.10: Listing 7.8 deutet eine solche Vorgehensweise an. Die Transaktionssteuerung könnte analog auch durch eine EJB erfolgen. Dem Vorteil der automatischen Verknüpfung des Entity Managers – und damit der von ihm durchgeführten Operationen – steht der Nachteil entgegen, dass die Entity-Objekte nach dem Verlassen der Servicemethoden detacht werden, da zu dem Zeitpunkt der Entity Manager geschlossen wird. Die darauf aufbauende Präsentation kann also noch nicht geladene Lazy-Attribute nicht mehr nachladen. Andererseits müssen die Objekte später wieder mittels *merge* attacht werden, wenn sie im Persistenzkontext weiterverarbeitet werden sollen, wodurch i. d. R. zusätzliche DB-Zugriffe nötig werden.

Conversation Scoped Entity Manager

ProVS nutzt eine andere Möglichkeit, die sich insbesondere durch den Conversation Scope von CDI ergibt: Zu Beginn eines Geschäftsprozesses wird ein Entity Manager erzeugt und bis zum Ende des Geschäftsprozesses offen gehalten. Am Ende des Java-Persistence-Kapitels wurde schon angedeutet, wie dieses Verfahren implementiert werden kann.

In ProVS wird allerdings statt einer Eigenimplementierung eine Portable Extension für CDI verwendet: JBoss Seam Persistence bietet u. a. den sog. Seam Managed Persistence Context an, mit dessen Hilfe konversationsbasierte Entity Manager erzeugt werden können.

Seam Persistence wird – wie alle CDI-Erweiterungen – dadurch aktiviert, dass die entsprechenden Bibliotheken in den Classpath integriert werden. In den Maven-Projektdateien des Beispielprojekts befinden sich dazu die folgenden Listing 7.9 gezeigten Dependencies. Der erste Teil – die sog. Bill Of Material, kurz BOM – deklariert nur die Versionen der zu einem Release gehörenden Teilmodule und weiteren Abhängigkeiten. Seam Persistence

benötigt anschließend zwei Dependencies für die Programmschnittstelle und die Laufzeitimplementierung.

```
<project …>
  …
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>org.jboss.seam</groupId>
        <artifactId>seam-bom</artifactId>
        <version>3.1.0.Final</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>

  <dependencies>
    <dependency>
      <groupId>org.jboss.seam.persistence</groupId>
      <artifactId>seam-persistence-api</artifactId>
    </dependency>

    <dependency>
      <groupId>org.jboss.seam.persistence</groupId>
      <artifactId>seam-persistence</artifactId>
      <scope>runtime</scope>
    </dependency>
  …
```

Listing 7.11: Maven Dependencies für JBoss Seam Persistence

Der Seam Managed Persistence Context – kurz SMPC – wird durch den in Listing 7.10 gezeigten Producer bereit gestellt. Dabei fällt auf, dass das gelieferte Objekt nicht vom Typ *EntityManager* ist, sondern vom Typ *EntityManagerFactory*. Dennoch wird dadurch die Injektion eines Entity Managers in andere CDI Bean ermöglicht, wobei dieser im Conversation Scope platziert wird.

```
public class SeamManagedPersistenceContextProducer
{
  @ExtensionManaged
  @PersistenceUnit(unitName = "ee-demos")
  @Produces @ConversationScoped
  EntityManagerFactory entityManagerFactory;
}
```

Listing 7.12: Producer für den Seam Managed Persistence Context

Im Kern liefert der SMPC einen Application Managed Entity Manager, dessen Lebenszeit durch die Conversation bestimmt wird. Applikationsgesteuerte Entity Manager verhalten sich im Hinblick auf Transaktionen anders als die zuvor beschriebene transaktionsgebundene Variante: Hier sind die schreibenden Operationen *persist*, *merge* und *remove* auch außerhalb einer aktiven Transaktion erlaubt. Die von ihnen erzeugten Änderungen wie auch Modifikationen an den vom Entity Manager verwalteten Entity-Objekten werden allerdings erst dann in die Datenbank geschrieben, wenn eine Transaktion schließlich begonnen wird und mit einem Commit endet.

Transaktionssteuerung

Zur Transaktionssteuerung kann ein weiteres Feature von JBoss Seam verwendet werden: Das von Seam Persistence referenzierte Modul Seam Transaction bietet einen Transaktions-Interceptor an, der mithilfe der aus der EJB-Spezifikation bekannten Annotation *@TransactionAttribute*[4] an Methoden gebunden werden kann. Er wird mit dem in Listing 7.11 gezeigten Eintrag in *beans.xml* aktiviert.

```
<beans …>
  <interceptors>
    <class>org.jboss.seam.transaction.TransactionInterceptor</class>
  …
```

Listing 7.13: Aktivierung des Transaktions-Interceptors aus Seam Transaction

Mit der Kombination dieses Transaktions-Interceptors und dem zuvor beschriebenen SMPC lässt sich die in Listing 7.12 beschriebene Methode formulieren, die zum Speichern aller bislang gemachten Änderungen aufgerufen werden kann.

```
@DataRepository
public class RepositoryMaster implements Serializable
{
  @Inject
  EntityManager entityManager;

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void saveAll()
  {
    this.entityManager.flush();
  }
}
```

Listing 7.14: Methode zum Speichern aller Änderungen

Nach diesen Vorarbeiten können Geschäftsprozesse nach dem folgenden Muster programmiert werden:

4 *javax.ejb.TransactionAttribute*

1. Start einer Long-running Conversation bei Beginn des Geschäftsprozesses
2. Aufruf beliebiger EntityManager-Operationen, auch über mehrere Requests erstreckt
3. Speichern der Änderungen durch Aufruf von *RepositoryMaster.saveAll()* und Beenden der Konversation am Ende des Geschäftsprozesses.

Listing 7.13 zeigt einen Ausschnitt aus dem Teil von ProVS, der die Personenstammdaten bearbeitet. Er wurde aus Gründen der Übersichtlichkeit ein wenig verkürzt. Im Begleitprojekt finden Sie den kompletten Code.

```
@ConversationScoped @Model
public class PersonenVerwaltungModel implements Serializable
{
  @Inject
  private SessionService sessionService;

  public String editPerson(Person person)
  {
    this.sessionService.beginConversation();
    setCurrentPerson(this.personRepository.findById(person.getId()));
    return "editPerson";
  }

  public String save()
  {
    this.repositoryMaster.saveAll();
    this.sessionService.endConversation();
    return "saved";
  }

  public String cancel()
  {
    this.sessionService.endConversation();
    return "cancelled";
  }
  …
}

public class SessionService implements Serializable
{
  @Inject
  private Conversation conversation;

  public void beginConversation()
  {
    if (this.conversation.isTransient())
      this.conversation.begin();
  }

  public void endConversation()
```

```
  {
    if (!this.conversation.isTransient())
      this.conversation.end();
  }
  …
}
```

Listing 7.15: Nutzung von Konversationen zur Implementierung von Geschäftsprozessen

Entity-Zombies

Zum Abschluss des Themas Persistenz in diesem Kapitel soll noch auf ein Detail der Implementierung einer persistenten Klasse hingewiesen werden: Die Klasse *Mitarbeiter* stellt die Verbindung zwischen Personen und Firmen her. Sie besitzt dazu zu beiden Klassen jeweils eine bidirektionale Relation (Listing 7.14).

```
@Entity(name = "Mitarbeiter")
@Access(AccessType.FIELD)
public class Mitarbeiter extends GeneratedIntegerIdEntity
{
  @NotNull
  @ManyToOne
  private Person person;

  @NotNull
  @ManyToOne
  private Firma  firma;
  …
}

@Entity(name = "Person")
@Access(AccessType.FIELD)
public class Person extends GeneratedIntegerIdEntity
{
  @OneToMany(mappedBy = "person",
           cascade = CascadeType.ALL, orphanRemoval = true)
  private List<Mitarbeiter> mitarbeiter;
  …
}

@Entity(name = "Firma")
@Access(AccessType.FIELD)
public class Firma extends GeneratedIntegerIdEntity
{
  @OneToMany(mappedBy = "firma",
           cascade = CascadeType.ALL, orphanRemoval = true)
  private List<Mitarbeiter>  mitarbeiter;
  …
}
```

Listing 7.16: Bidirektionale Relationen zwischen Mitarbeiter und Person bzw. Firma

An der Parametrierung der *@OneToMany*-Annotationen mit *cascade=CascadeType.ALL* und *orphanRemoval=true* kann man erkennen, dass *Mitarbeiter* aus Sicht von *Person* und *Firma* als abhängige Entity geführt wird.

Soll in einem Geschäftsprozess ein Objekt vom Typ *Mitarbeiter* gelöscht werden, muss man darauf achten, dass zuvor alle Verbindungen von den betroffenen Objekten der Typen *Person* bzw. *Firma* gelöst werden, indem das zu löschende Objekt aus den entsprechenden Listen entfernt wird. Wird dies nur auf einer Seite getan, bspw. nur im verbundenen Firmenobjekt, so kann es passieren, dass das gelöschte Mitarbeiterobjekt unmittelbar nach dem Löschen wieder erzeugt wird. Auslöser ist der Parameter *cascade=CascadeType.ALL* im Personenobjekt, der u. a. die Operationen *persist* und *merge* umschließt und somit eine Neuanlage des Mitarbeiterobjekts auslösen kann.

Zur Vermeidung dieses Zombieverhaltens wird in *Firma.removeMitarbeiter* die Verbindung des zu löschenden Mitarbeiters von Person gelöst, bevor die eigentliche Löschoperation durchgeführt wird (Listing 7.15).

```
@Entity(name = "Firma")
@Access(AccessType.FIELD)
public class Firma extends GeneratedIntegerIdEntity
{
    public void removeMitarbeiter(Mitarbeiter mitarbeiter)
    {
        // Mitarbeiter von Person lösen,
        // um Neuanlage durch Kaskadierung nach dem Löschen zu verhindern
        mitarbeiter.setPerson(null);
        this.mitarbeiter.remove(mitarbeiter);
    }
…}

@Entity(name = "Mitarbeiter")
@Access(AccessType.FIELD)
public class Mitarbeiter extends GeneratedIntegerIdEntity

{
  public void setPerson(Person person)
     {
       if (this.person != null)
       this.person.removeMitarbeiter(this);    this.person = person;

       if (this.person != null)      this.person.addMitarbeiter(this);
    }
…}
```

Listing 7.17: Lösen der Objektverbindungen vor dem Löschen

7.4 Views

Die Benutzeroberfläche der Anwendung besteht aus JSF-Facelets. Da die Bestandteile der Standardbibliothek nur eine Grundfunktionalität anbieten, angelehnt an die HTML-Elemente, wird für ProVS als zusätzlich Komponentenbibliothek PrimeFaces eingesetzt. Dazu enthält die Maven-Projektdatei die in Listing 7.16 gezeigten Abhängigkeiten.

```
<project …>
  …
  <dependencies>
    <dependency>
      <groupId>org.primefaces</groupId>
      <artifactId>primefaces</artifactId>
      <version>3.1.1</version>
    </dependency>

    <dependency>
      <groupId>org.primefaces.themes</groupId>
      <artifactId>sam</artifactId>
      <version>1.0.3</version>
    </dependency>
  …
```

Listing 7.18: Maven Dependencies für PrimeFaces

Die erste Dependency betrifft die Komponentenbibliothek, die zweite eine sog. Skinning-Bibliothek, mit der die Gestaltung der Elemente modifiziert wird. Sie wird durch einen Eintrag im Deployment Descriptor der Anwendung aktiviert (Listing 7.17).

```
<web-app …>
  <context-param>
    <param-name>primefaces.THEME</param-name>
    <param-value>sam</param-value>
  </context-param>
  …
```

Listing 7.19: Aktivierung eines Skins für PrimeFaces

PrimeFaces enthält eine Menge schöner Oberflächenelemente, von denen in ProVS einige eingesetzt werden (Abb. 7.4,Abb. 7.5 und Abb. 7.6). Da das Oberflächendesign nicht Kernthema dieses Buches ist, wurde ihm für die Beispielanwendung keine wesentliche Beachtung geschenkt.

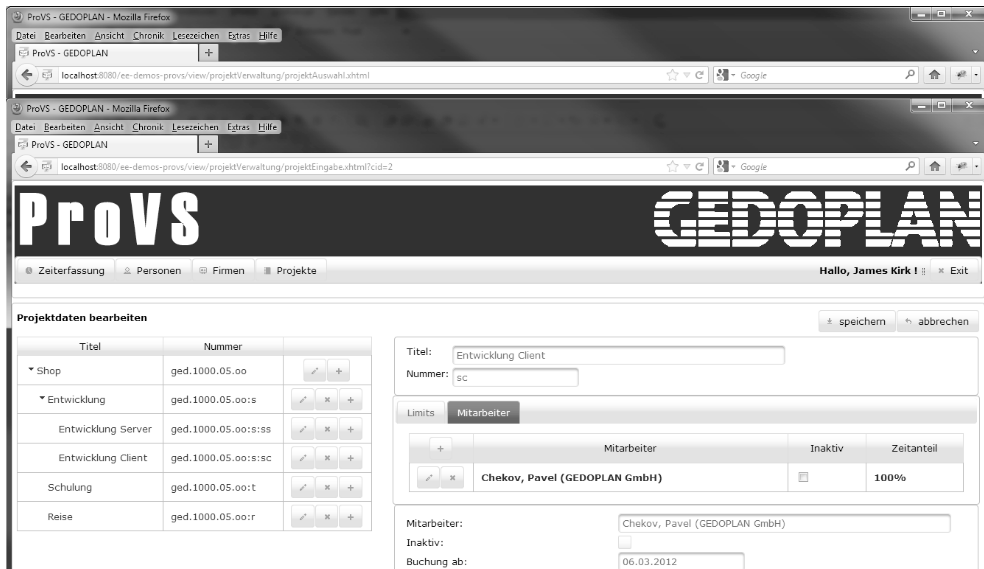


Abbildung 7.5: Baumartige Darstellung von Daten, diverse Layoutelemente

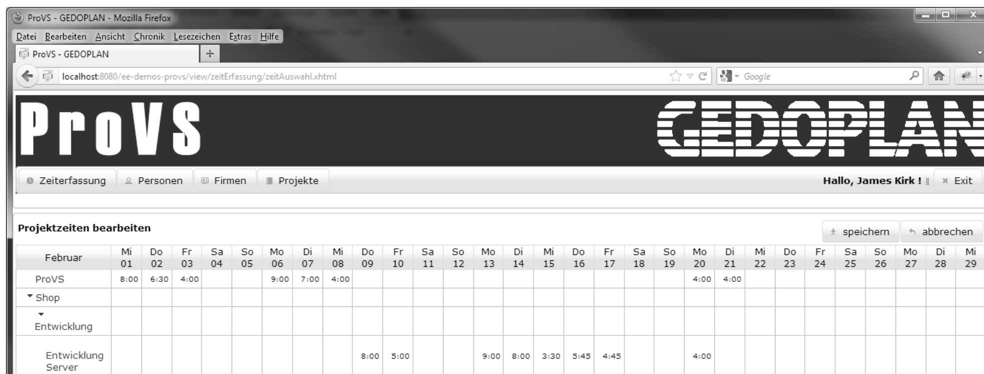


Abbildung 7.6: Tabelle mit Eingabeelementen

Templates

Die Views nutzen ein gemeinsames Template, um eine einheitliche Gestaltung der Oberfläche zu erreichen. Hier sind die Navigationselemente enthalten, ein Platzhalter für den Seiteninhalt sowie ein Anzeigeelement für globale Meldungen (Listing 7.18).

```
<html...>
<h:head>
  ...
  <h:outputStylesheet library="css" name="provs.css" />
</h:head>
```

```
<h:body>
  <p:layout fullPage="true">
    <p:layoutUnit position="north">
      ...
      <h:form>
        <p:toolbar rendered="#{navigationModel.loggedIn}">
          <p:toolbarGroup align="left">
            <p:commandButton type="push" value="Zeiterfassung" .../>
          ...
        </p:toolbar>
      </h:form>
    </p:layoutUnit>

    <ui:insert name="content">
      <p:layoutUnit position="center">???</p:layoutUnit>
    </ui:insert>

    <ui:insert name="messages">
      <p:layoutUnit position="south">
        <p:messages globalOnly="true" display="text" />
      </p:layoutUnit>
    </ui:insert>
  </p:layout>
</h:body>
</html>
```

Listing 7.20: Template „WEB-INF/templates/provs_template.xhtml“

Die konkreten Views referenzieren das Template in der im JSF-Kapitel gezeigten Weise (Listing 7.19).

```
<html …>
<h:body>
  <ui:composition template="/WEB-INF/templates/provs_template.xhtml">
    <ui:define name="content">
      <prime:layoutUnit position="center">
        …
```

Listing 7.21: Nutzung des Templates in einer View

Converter mit Datenbankzugriff

Für die Eingabe von fachlichen Objekten wird in aller Regel ein passender Converter benötigt. So ist bspw. für die Auswahl eines Landes in einer Drop-Down-Box ein Converter notwendig, der Objekte des Typs *Land* in eine String-Repräsentation zur Verarbeitung in der HTML-Oberfläche wandelt und umgekehrt. Für die Konvertierung eines Strings zurück in ein Land wird ein Datenbankzugriff benötigt, um die verfügbaren bzw. gültigen *Land*-Werte zu ermitteln. Das kann auf einfache Weise mithilfe eines entsprechen-

den Repository-Objekts gelöst werden. Im Standard ist es allerdings nicht möglich, in ein Converter-Objekt eine CDI Bean injizieren zu lassen. Abhilfe schafft hier erneut eine CDI-Erweiterung: JBoss Seam Faces bietet u. a. Injektionsmöglichkeiten für Faces Converter und Validators an. Damit lässt sich der Converter wie gewünscht implementieren (Listing 7.20).

```
@FacesConverter(forClass = Land.class)
public class LandConverter implements Converter
{
  @Inject
  private LandRepository landRepository;

  public Object getAsObject(FacesContext context, UIComponent component,
                            String value)
  {
    if (value == null) return "";
    return this.landRepository.findById(value);
  }

  public String getAsString(FacesContext context, UIComponent component,
                            Object value)
  {
    if ("".equals(value)) return null;
    Land land = (Land) value;
    return land.getId();
  }
}
```

Listing 7.22: Implementierung eines Converters mithilfe einer injizierten CDI Bean

Seam Faces hat allerdings eine Eigenschaft, die im Zusammenhang mit der zuvor beschriebenen Nutzung eines konversationsgesteuerten Entity Managers zu einer Fehlfunktion führt: Seam Faces installiert einen Servlet Listener, der für jeden Web-Request eine Transaktion beginnt und diese mit Ende der Request-Verarbeitung wieder schließt. Dadurch werden Datenänderungen ungewollt und eventuell fehlerhaft mit jedem Request in die Datenbank geschrieben. Diesen Listener muss man somit bei Nutzung des SMPC-basierten Verfahrens abschalten, und zwar durch einen Context-Parameter im Descriptor *web.xml* (Listing 7.21).

```
<web-app …>
  <context-param>
    <param-name>org.jboss.seam.transaction.disableListener</param-name>
    <param-value>true</param-value>
  </context-param>
  …
```

Listing 7.23: Abschaltung des Servlet Listeners zur Transaktionssteuerung

Leider ist es damit noch nicht getan: Seam Faces installiert einen weiteren Listener zur Transaktionssteuerung, diesmal als Phase Listener: Er startet eine Transaktion vor Phase 2 (Apply Request Values) und beendet sie wieder nach Phase 6 (Render Response). Er lässt sich über das Verfahren der View-Konfiguration mithilfe der Annotation *@ViewConfig* abschalten, das ebenfalls Teil von JBoss Seam ist. Dazu muss das in Listing 7.22 gezeigte Interface in das Projekt integriert werden. Mithilfe von *@ViewConfig* können noch weitere Eigenschaften der Views konfiguriert werden. Bei Interesse schauen Sie bitte in die JBoss-Seam-Dokumentation.

```
@ViewConfig
public interface ViewConfiguration
{
  static enum ViewConfigurationParameter
  {
    @ViewPattern("/*")
    @SeamManagedTransaction(SeamManagedTransactionType.DISABLED)
    ALL;
  }
}
```

Listing 7.24: Abschaltung des Phase Listeners zur Transaktionssteuerung

7.5 Fachliche Injektion

Die Injektionsmöglichkeiten von CDI werden zu einem großen Teil für den Aufbau der Struktur der Anwendung verwendet: Models injizieren sich Services, diese injizieren sich Repositories usw. Die Producer und Qualifier von CDI eröffnen aber auch noch den anderen Weg der fachlichen Injektion. Hierbei werden Geschäftsobjekte mithilfe von Producern so bereitgestellt, dass andere CDI Beans sie mittels Injektion benutzen können.

Listing 7.23 zeigt zwei Models aus dem Zeiterfassungsteil von ProVS, zwischen denen durch eine fachliche Injektion eine Datenübergabe stattfindet. Das obere Model – *ZeiterfassungAuswahlModel* – ist für die Auswahl der Person zuständig, für die Projektzeiten erfasst werden sollen. Sie stellt diese Person mithilfe des gezeigten Producers zur Verfügung. Die zweite Bean – *ZeiterfassungEingabeModel* – dient dem Eingabeformular als Model. Sie übernimmt die ausgewählte Person durch einfache Injektion.

```
@Model
public class ZeiterfassungAuswahlModel implements Serializable
{
  …
  @Produces @Current @Zeiterfassung
  public Person getBuchungsPerson() { return this.buchungsPerson; }
  …
}
```


entwickler.press

```
@ConversationScoped @Model
public class ZeiterfassungEingabeModel implements Serializable
{
  …
  @Inject @Current @Zeiterfassung
  private Person  buchungsPerson;
  …
}
```

Listing 7.25: Fachliche Injektion

Diese Verfahren ermöglicht eine recht elegante Zusammenarbeit verschiedener Komponenten ohne direkte Kopplung trotz gemeinsamer Daten. Dadurch wird häufig eine übersichtliche Aufteilung der Model-Klassen erleichtert. Vergleichen Sie dazu einmal die angeführten Models der Zeiterfassung mit dem *FirmenVerwaltungModel*, das als Model für die Verwaltung der Firmendaten dient und in herkömmlicher Weise ohne fachliche Injektion programmiert ist.

Ein weiteres Beispiel für eine fachliche Injektion in ProVS ist die Bereitstellung des aktuell angemeldeten Users in der Klasse *SessionService (Listing 7.*24).

```
@SessionScoped @DomainService
public class SessionService implements Serializable
{
  @Produces @Current @Angemeldet
  public Principal getCurrentUser() { … }
  …
```

Listing 7.26: Bereitstellung des aktuellen Benutzers

Jede andere Bean der Anwendung kann so auf den aktuellen Nutzer zugreifen, ohne Kenntnis davon haben zu müssen, wie er ermittelt wird.

Bedenken Sie bei dieser Vorgehensweise, dass die Injektion nur einmal beim Erstellen des empfangenden Objekts ausgeführt wird.

Stichwortverzeichnis

Symbole

@Access 62
@Alternative 29, 46
@Any 27, 32, 34, 44
@ApplicationScoped 38, 193
@AroundInvoke 41
@AssertFalse 156
@AssertTrue 156
@Asynchronous 249
@AttributeOverride 73
@Basic 96
  fetch 96
@Cacheable 142
@CollectionTable 75
@Column 71, 128
@Constraint 157
  validatedBy 159
@ConversationScoped 38, 193
@DecimalMax 156
@DecimalMin 156
@Decorator 44
@Default 26, 32, 33
@Delegate 44
@DenyAll 252
@Dependent 39
@Digits 156
@DiscriminatorColumn 124
@DiscriminatorValue 124
@Disposes 33
@EJB 241
@ElementCollection 75
  fetch 95
@Embeddable 73
@EmbeddedId 130
@Entity 62, 70
@EntityListeners 138
@Enumerated 72
@FacesComponent 232
@FacesConverter 208, 209
@Future 156
@GeneratedValue 76
@Id 62
@IdClass 130
@Inheritance 123
@Inject 21, 23, 51, 241
@Interceptor 41
@InterceptorBinding 42
@JoinColumn 84, 86, 88, 91
@JoinTable 84, 88, 91, 93
@Lob 73
@Local 241
@LocalBean 242
@ManagedBean 184
@ManyToMany 93
  cascade 96
  fetch 94
@ManyToOne 83, 131
  cascade 96
  fetch 94
@MapKeyColumn 76
@MappedSuperclass 127
@MapsId 132
@Max 156
@MessageDriven 240
@Min 156
@Model 45, 257
@Named 24, 28, 32, 46, 184
@NamedNativeQuery 113

@NamedQuery 111
@New 40
@Nonbinding 43
@NonBinding 28
@NotNull 156
@Null 156
@Observes 48
@OneToMany 86
  cascade 96
  fetch 94
  orphanRemoval 98
@OneToOne 91, 131
  cascade 96
  fetch 94
  orphanRemoval 98
@OrderBy 99
@OrderColumn 99
@Past 156
@Pattern 156
@PermitAll 252
@PersistenceContext 66, 144
@PersistenceUnit 146
@PostConstruct 25
@PreDestroy 26
@PrePersist 164
@PreRemove 164
@PreUpdate 164
@PrimaryKeyJoinColumn 92
@Produces 32, 33
@Qualifier 26
@Remote 243
@Remove 239, 246
@ReportAsSingleViolation 158
@RequestScoped 36, 193
@Resource 30
@RolesAllowed 252
@Schedule 250
@SecondaryTable 128
@SessionScoped 37, 193
@Singleton 239
@Size 156
@SqlResultSetMapping 113
@Stateful 239
@StatefulTimeout 239
@Stateless 238
@StaticMetamodel 117
@Stereotype 45
@Table 70
@Temporal 73
@Timeout 251
@TransactionAttribute 246, 266
@TransactionManagement 246
@Transient 73
@Typed 22
@UniqueConstraint 71
@Valid 157
@Version 134
@WebServlet 169

A

ActionEvent 203
ActionListener 203
Ajax 218
  Callbacks 221
  Events 220
  für Aktionselemente 219
  JavaScript API 222
AjaxBehaviorEvent 220
Aktivierung einer Bean 37
Aktivierung eines Decorators 45
Aktivierung eines Interceptors 43
Allocation Size 78
Alternatives 28
andauernde Konversation 38
AnnotationLiteral<T> 50
Annotation Processor 117
Anordnung von Relationselementen 99
Apache MyFaces CODI 53
Application Exceptions 248

Application Managed
Entity Manager 145
Application Message Bundle 199
Application Scope 37
Architekturmodell 11
aspektorientierte Programmierung 40
asynchrone Methoden 249
AsyncResult 249
Ausgabe von Meldungen 209
Autodeploy-Ordner 14

B

Bean Manager 51
BeanManager 51
Bean Name 24
Bean Proxies 39
beans.xml 20, 29, 43, 45, 46
Bean Type 21
Bean Validation 153
Bean Validation in SE-Umgebungen 166
Built-in Beans 31
Bulk Delete 139
Bulk Update 139

C

Cache 142
Caching 140
Callback-Methoden 137
CascadeType 96
cc:actionSource 228, 230
cc: attribute 228
cc: editableValueHolder 228, 229
cc: facet 228, 231
cc: implementation 228
cc: insertChildren 228, 229
cc: insertFacet 228
cc: interface 227
cc: renderFacet 228
cc: valueHolder 228
CDI 13, 17
CDI Bean 20
Component Type 232
Composite Components 227
mit Backing Bean 231
Constraint Composition 157
Constraint Programming 159
ConstraintValidator 159
ConstraintValidatorContext 160
ConstraintViolation 161
ConstraintViolationException 165
Construktor Injection 23
Container Managed Persistence 58
Context Parameters 176
Contexts 35
Conversation 38
Conversation Scope 38
Converter 208
CriteriaBuilder 114
Criteria Queries 114
Aggregationsfunktionen 121
Attributzugriffe 116
Bedingungen 118
Constructor Expressions 120
Fetch Joins 122
Fluent API 115
Funktionen 121
Gruppierung 121
Joins 119
Parameter 120
Query Roots 115
Selektion 115
Sortierung 122
statisches Metamodell 116
Tupe-Selektion 120
CriteriaQuery 115
CRUD-Methoden 66

D

DAO 263
Dead Lock 136
Decorator-Aktivierung 45
Decorator Class 44
Decorators 43
Default Listener 139
Delegate 44
Dependency Injection 19
Dependent IDs 130
Dependent Scope 39
Deployment-Formate 14
Detached 69
Detached Objects 68
Direktionalität 83
Diskriminator 123
Disposer 31
Disposer Methods 33

E

Eager Loading 94, 95
EAR-Deployment 15
EAR-Format 240
Eclipselink 61
EJBAccessException 253
EJBContext 248
EJB Deployment 240
Enterprise-Anwendung 13
Enterprise JavaBeans 13, 237
Entity Beans 58
Entity-Klassen 62
Entity Manager
  Application Managed 266
  konversationsgesteuert 264
  transaktionsgebunden 263
EntityManager 66
EntityManagerFactory 151
Entity Manager injizieren 66
EntityTransaction 152
EnumType 72
erweiterte Entity Manager 143
Events 47
Events erzeugen 47
Events feuern 48
Event Source 47
Events verarbeiten 48
Event<T> 47
Eventverarbeitung 47
Extended Entity Manager 144

F

f: actionListener 187, 203
f: ajax 219, 220, 221, 222
f: attribute 183
f: convertDateTime 207
f: converter 208
f: convertNumber 207
f: event 206
f: facet 182, 196
f: loadBundle 199
f: param 182, 183
f: phaseListener 205
f: selectItems 188
f: subview 183
f: validateBean 183
f: validateDoubleRange 183
f: validateLength 183
f: validateLongRange 183
f: validateRegex 183
f: validateRequired 183
f: validator 183
f: valueChangeListener 204
f: verbatim 183
f: view 183
Facelets 173
faces-config.xml 176, 184, 199
  locale-config 197
  message-bundle 199
  navigation-rule 191
  phase-listener 205
  resource-bundle 198
Faces Events 203

Faces Servlet 172
Faces Servlet registrieren 175
fachliche Identität 76
Fat Session Problem 37
FetchType 94
Feuern von Events 48
Field Access 63
Field Injection 21
First Level Cache 140
FlushModeType 110
Flush-Modus 110
Full Profile 15

G

GenerationType 76
generierte IDs 76
GET Support 201

H

h:body 181
h:button 182
h:column 182, 194
h:commandButton 181
h:commandLink 181
h:dataTable 182, 194
h:form 182
h:graphicImage 179
h:head 181
h:inputHidden 181
h:inputSecret 181
h:inputText 181
h:inputTextarea 181
h:link 182
h:message 182, 209
h:messages 182, 209
h:outputFormat 179
h:outputLabel 179
h:outputLink 179
h:outputScript 182
h:outputStylesheet 182
h:outputText 179
h:panelGrid 180
h:panelGroup 180
h:selectBooleanCheckbox 181
h:selectManyCheckbox 181
h:selectManyListbox 181
h:selectManyMenu 181
h:selectOneListbox 181
h:selectOneMenu 181
h:selectOneRadio 181
Hibernate 61
HttpServlet 169
HttpServletRequest 169
HttpServletResponse 169

I

ID-Klasse 129
immediate
  bei Aktionskomponenten 218
  bei Eingabekomponenten 218
InheritanceType 123
InitialContext 245
Initializer Methods 23
InjectionPoint 32, 34
Injektion eines Entity Managers 66
Injektion von Bean-Instanzen 49
Instance<T> 49
Integration von Bean Validation
    in JPA 164
    in JSF 166
Interceptor-Aktivierung 43
Interceptor Binding 42
Interceptor Class 41
Interceptors 40
Internationalisierung 196
  Locale 197
  Programmgesteuerter Zugriff
    auf Texte 199
  Resource Bundles 198
Introspektion des Injektionsziels 34
Inversion of Control 19
InvocationContext 41

J

Java Data Objects 59
Java EE Resources 30
Java Persistence 57
Java Persistence in SE-Anwendungen 149
  Erzeugung eines Entity Managers 150
  Konfiguration 149
  Transaktionssteuerung 152
JavaServer Faces 167
JavaServer Pages 170
JBoss Seam 54
  Faces 273
  Persistence 264
JDO 59
JNDI-Namen von EJBs 244
jndi.properties 245
JoinType 119
JPA 57
JPQL 100
  Aggregationsfunktionen 106
  Constructor Expressions 108
  Fetch Joins 111
  from 102
  Funktionen 107
  group by 108
  having 108
  Named Queries 111
  Operatoren 104
  order by 108
  Paging 109
  Parameter 106
  Query Hints 110
  select 102
  where 104
jsf.ajax.addOnError 222
jsf.ajax.addOnEvent 222
JSF-EL 187
JSF Expression Language 186
JSF Tag Libraries 178
  HTML-Bibliothek 178
JSF-Tag-Libraries
  Core-Bibliothek 182
JSP-EL 187

K

Kardinalität 83
Kaskadieren 96
Komponentenbibliotheken 233
Kontexte 35
Konversation 38
Konvertierung 206
  Custom Converter 208
  vordefinierte Konverter 207

L

Lazy Load Exception 95
Lazy Loading 94
Lifecycle Callbacks 25
Listener-Klasse 138
Locale 197
Local Interface 241
Locking 133
Locking Hints 136
LockModeType 133
LockTimeoutException 136
long-running Conversation 38

M

Managed Beans 184
mappedBy 88, 91
Marshalling 243
Message Bundle 162
Message-driven Bean 240
Method Injection 23
Model 257
Modellgenerator 117
Model View Controller 171
MVC 171

N

NamingContainer 232
Native Queries 112
  Named Queries 113
Navigation 191
  inline 192
  programmgesteuert 193
  regelbasiert 191
NavigationHandler 193
No-Interface View 242

O

Objektgleichheit 79
Objektprüfung 161
Objektrelationen 82
Observer Method 48
OpenJPA 61
OptimisticLockException 135
Optimistic Locking 133, 134
O/R-Mapper 59
orm.xml 65
Orphan Removal 98
Outcome 191
Ownership 82

P

Partial Requests 219
Passivierung einer Bean 37
Persistence 151
Persistence Unit 64
PersistenceUnitUtil 81
persistence.xml 64, 65, 141, 164
Persistent 69
Persistente Ordnung 99
PessimisticLockException 136
Pessimistic Locking 133, 135
Phase Events 204
PhaseListener 205
Plattformen 16
Portable Extensions 53
PrimeFaces 270
Producer 31
Producer Fields 33
Producer Methods 32
Profile 15
programmgesteuerter Zugriff auf CDI Beans 49
Property Access 63
ProVS 255

Q

Qualifier 26
Qualifier mit Parametern 27
Queries 100
Query Cache 143

R

Redirect 192
Relation 82
Remote Interface 243
Remote-Zugriff 243
Repository 258, 261
Request Processing Lifecycle 173
  1 - Restore View 174
  2 - Apply Request Values 174
  3 - Process Validations 174
  4 - Update Model Values 175
  5 - Invoke Application 175
  6 - Render Response 175
Request Scope 36
ResourceBundle 198
Ressourcenverwaltung 199
  Internationalisierung 201

S

Scheduling 250
Scopes 35, 193
Seam Managed Persistence Context 264
Secondary Tables 128
Second Level Cache 141
Second Level Cache API 142
Security 252
- deklarativ 252
- Login-Konfiguration 234
- programmgestützt 253
- Security-Rollen 235
- Zugriffsregeln 235

SelectItem 188
Serializable 37
Service 258
Servlets 168
Session Scope 37
Setter Injection 23
Singleton Bean 239
Stateful Session Bean 239
Stateless Session Bean 238
Stereotypes 45
System Events 205
System Exceptions 248

T

technische Identität 76
Template 222, 223, 271
- mehrere pro Seite 225
- mehrstufig 225

Template Client 222
Template-Client 224
Templating 222
TemporalType 73
Timer 250
- nichtpersistent 251
- persistent 251

TimerService 251
Transaction Attribute 246
Transaction Management 246
Transaktions-Interceptor 42, 266
Transaktionssteuerung 246
Transient 69
transient Conversation 38
transiente Konversation 38
transitive Gültigkeit 157
Tuple 120

U

ui:composition 224
ui:decorate 225
ui:define 224
ui:insert 223
ui:param 224
UICommand 233
UIComponent 232
UIInput 233
UIMessage 233
UINamingContainer 233
UIOutput 233
UISelectMany 233
UISelectOne 233
Unified Expression Language 186
- arithmetische Ausdrücke 190
- Methodenbindung 187
- vordefinierte Variablen 189
- Wertebindung 188

V

Validation 166
Validation Constraints 155
ValidationMessages 162
validation.xml 155
Validator 166
ValidatorFactory 166

Validatorklasse 159
Validierung 210
  Eingabewerte 211
  feldübergreifend 211
Validierungsgruppen 163
  Default 163
Validierungsmeldungen 162
  Internationalisierung 162
ValueChangeEvent 203
ValueChangeListener 204
Verarbeitung tabellarischer Daten 194
Vererbung 122
  JOINED 126
  Non-Entity-Basisklassen 127
  SINGLE_TABLE 123
  TABLE_PER_CLASS 125
Versionsattribut 134
View ID 191
vordefinierte Constraints 156

W

WAR-Deployment 15
WAR-Format 241
Webanwendung 13
Webanwendungen 167
WebBeans 17
Web Profile 15
web.xml
  ecurity-constraint 235
  error-page 236
  login-config 234

X

XmlHttpRequest 219

Z

zusammengesetzte IDs 129