Java 2 Enterprise Edition - Enterprise Java Beans & Enterprise Information Systems

Inhalt

EINLEITUNG 2

1. ENTERPRISE INFORMATION SYSTEM TIER 3

1.1 DIE BEDEUTUNG VON INFORMATIONSSYSTEMEN IN MODERNEN UNTERNEHMEN 3
1.2 JDBC 3
1.2.1 JDBC Core Package 5
1.2.1.1 Laden des Treibers 6
1.2.1.2 Aufbau einer Verbindung zum DB-Server 6
1.2.1.3 Statements 6
1.2.1.4 Datenmanipulation 6
1.2.1.5 Select-Anfragen und Result-Sets 6
1.2.1.6 *Prepared Statements* 7
1.2.1.7 *Callable Statements* 8
1.2.1.8 Erweiterungen des *Core Package* in JDBC 2.0 8
1.2.2 JDBC Optional Package 9
1.3 SQLJ [2, KAP.9] 10
1.3.1 Einbettung von SQL in Java-Programme 10
1.3.2 Server-sided Java mit SQLJ 11
1.3.3 DB-seitige Verwendung von Java-Datentypen 12
1.4 J2EE-CONNECTOR 12
1.4.1 Motivation 13
1.4.2 Resource Adapter 13
1.4.2.1 *Resource Manager*-spezifisches API 13
1.4.2.2 Common Client Interface 13
1.4.3 Kontrakte 14
1.5 IN ALLER KÜRZE: DAS TRANSAKTIONSPARADIGMA 14
1.6 JAVA TRANSACTION API 15
1.6.1 Motivation 15
1.6.2 Transaktionsattribute 15
1.6.3 Verwendung von JTA-Transaktionen 17

2. ENTERPRISE JAVA BEANS TIER 17

2.1 *BUSINESS OBJECTS* 17
2.1.1 Gemeinsame Anforderungen 17
2.2 *ENTERPRISE JAVA BEANS* 19
2.2.1 Grundlagen 19
2.2.1.1 Container 20
2.2.2.2 Deployment-Deskriptoren 21
2.2.2 Remote Method Invocation (RMI) 21
2.2.3 RMI over IIOP 22
2.2.4 Entity Beans 22
2.2.4.1 Eigenschaften 22
2.2.4.2 Verwendung 23
2.2.4.3 Persistenz 23
2.2.4.4 Interfaces 24
2.2.4.5 EJB-Query Language (EJB-QL) 25
2.2.5 Session Beans 25
2.2.5.1 Stateful Session Beans 25
2.2.5.2 Stateless Session Beans 26
2.2.5.3 Session Beans als *Facade* 26

2.2.6 JMS und Message-driven Beans *26*
2.2.6.1 Java Message Service API 26
2.2.6.2 Message-Driven Beans 28
2.2.7 Allgemeine Entwurfsrichtlinien *28*
2.2.7.1 Access-Objekte 29
2.2.7.2 Value-Objekte 29
2.2.8 Nutzung von EJBs *29*
3. ZUSAMMENFASSUNG UND AUSBLICK 30
ANHANG 31
Quellenverzeichnis *31*

Einleitung

Nachdem die erste Ausarbeitung die Grundkonzepte und das Schichtenmodell der J2EE-Plattform erläutert und die Basistechnologien vorgestellt hat, befaßt sich der folgende Text näher mit zwei der Schichten.
Die unterste Schicht der heute in Unternehmen verbreiteten *Three-Tier*-Informationssysteme bilden Systeme zur persistenten Datenhaltung, in der J2EE-Plattform wird diese als *Enterprise Information Systems Tier* bezeichnet. Hier soll insbesondere auf die Anbindung von Informationssystemen an Anwendungen eingegangen werden.
Anschließend werden Technologien der mittleren Schicht vorgestellt, die i.a. als „Applikationsserver“ (bzw. „J2EE-Server“ in der Sun-Terminologie) bezeichnet wird. Man trennt diese Schicht häufig man in zwei weitere Subschichten auf, von denen hier der *Enterprise Java Beans Tier*, in dem Client-unabhängige Anwendungslogik beheimatet ist, weiter betrachtet wird.

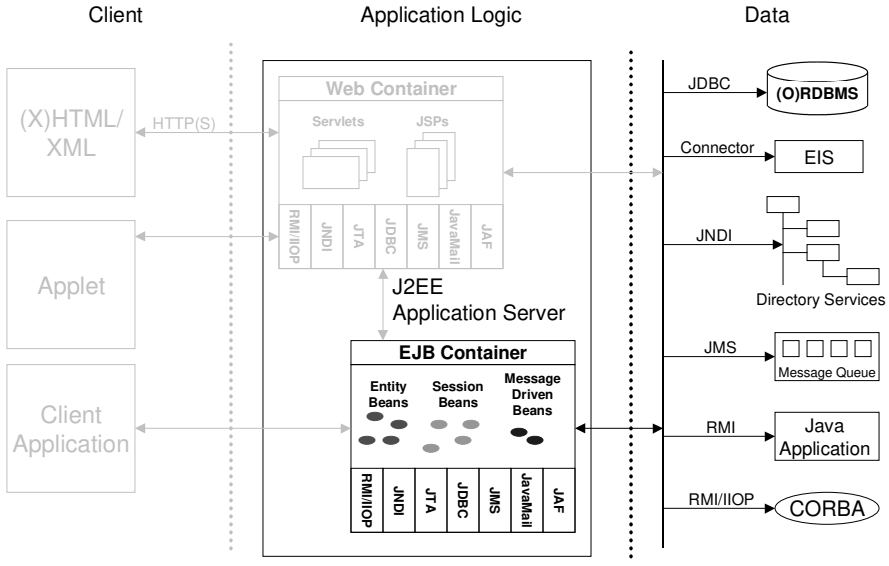


Abb. 1 Übersicht

1. Enterprise Information System Tier

1.1 Die Bedeutung von Informationssystemen in modernen Unternehmen

Zum Einstieg in die Betrachtung der untersten Schicht der J2EE-Plattform soll zunächst auf die Bedeutung hingewiesen werden, welche die Datenhaltung in heutigen Unternehmen hat.
Galten bis vor einigen Jahren Anlagen und andere Investitionsgüter als Grundlage einer erfolgreichen Geschäftstätigkeit, so bildet heute Information die Basis für die Funktionstüchtigkeit nahezu jeder Unternehmung. Ihre ständige Verfügbarkeit und sichere Aufbewahrung ist oft kritisch für den Erfolg und auch das Überleben einer Unternehmung. Inkonsistenzen oder Verlust von Daten sind somit unter allen Umständen zu vermeiden.
Im Zuge der Entwicklung der Informationstechnologie sind so in fast allen modernen Unternehmen Informationssysteme entstanden, die zur Speicherung und Verwaltung von Daten dienen. Sie existieren in vielen unterschiedlichen Ausprägungen. Neben den relationalen Datenbanksystemen, die sich nach anfänglichen Querelen auf der Suche nach „dem besten“ Datenmodell heute auf breiter Front durchgesetzt haben, gibt es neuerdings objektrelationale Datenbanksysteme, welche die Vorteile der objektorientierten Entwicklung und die Robustheit relationaler Datenbanksysteme vereinen sollen. Auf ihnen basieren *Enterprise Resource Planning*-Systeme wie SAP/R3. Aber auch „Altlasten“ wie TP-Monitore auf Mainframe-Basis und andere sogenannte *Legacy*-Systeme aus den Anfangstagen der Informationstechnik sind noch an vielen Orten im Einsatz. In der Terminologie des Java-Erfinders Sun werden alle diese Systeme unter dem Begriff *Enterprise Information Systems* (EIS) zusammengefaßt. Ein weiterer gängiger Begriff für Informationssysteme, die u.U. mehrere Datenquellen verwalten ist *Resource Manager* (RM). Sie bilden in der Drei-Schichten-Architektur der J2EE-Plattform den *Enterprise Information System Tier*.
Bei der Entwicklung von Anwendungen, die auf Daten dieser Informationssysteme zugreifen, müssen die individuellen Fähigkeiten Einschränkungen der Systeme berücksichtigt werden. Besonders die *Legacy*-Systeme weisen hier z.T. erhebliche Restriktionen auf, so unterstützen sie oft keine verteilten Transaktionen, haben nur eingeschränkte Sicherheitsfunktionen oder sonstige Limitierungen. Mit dem Aufkommen des E-Commerce sind zusätzlich viele neue Anwendungsmöglichkeiten entstanden, die jedoch auch auf diesen *Legacy*-Systemen aufsetzen müssen. Für die Entwickler müssen daher Schnittstellen zur Verfügung gestellt werden, die einerseits die Fähigkeiten des Informationssystems voll nutzen, aber gleichzeitig von den z.T. erheblichen Unterschieden zwischen ihnen abstrahieren.

1.2 JDBC

Wie erfolgt nun die Anbindung der *Enterprise Information Systems* an die J2EE-Plattform bzw. die Anwendung?
Der überwiegende Teil dieser Systeme sind relationale und in jüngerer Zeit objektrelationale Datenbanksysteme, auf die üblicherweise mittels SQL zugegriffen wird. Deren aktuellste Variante, SQL:1999 oder auch SQL3 genannt, fließt inzwischen zunehmend in entsprechende Produkte ein. Prinzipiell stehen dem Anwendungsentwickler zwei grundlegende Methoden zur Nutzung von SQL zur Wahl:
Die erste Möglichkeit ist ein *Call Level Interface* oder kurz CLI genannt, das über Methodenaufrufe das Absetzen von SQL-Befehlen an die Datenbank ermöglicht. Diese Aufrufe lassen sich nahtlos in die Programme integrieren, da es sich für den Compiler um ganz normale Methodenaufrufe handelt. Für die klassischen Programmiersprachen wie z.B. C stellt dazu jeder Datenbankhersteller seine eigene Bibliothek mit eigener Schnittstelle zur Verfügung, was die Portabilität der Anwendungen erheblich verschlechtert, da oft

weitreichende Eingriffe in den Code notwendig sind, um auf ein anderes als das ursprünglich vorgesehene Datenbanksystem zuzugreifen.
Die zweite Möglichkeit ist „eingebettetes“ oder *embedded* SQL. Dabei werden SQL-Befehle direkt in den Quelltext der Wirtssprache geschrieben. Ein Präprozessor des DB-Herstellers bildet sie dann auf entsprechende CLI-Aufrufe ab. Anschließend kann die eigentliche Übersetzung mit dem normalen Compiler der Wirtssprache erfolgen.
Wegen der oben genannten Einschränkungen bei der Portabilität kam ein Ansatz mit herstellerspezifischen Schnittstellen für Suns J2EE-Plattform nicht in Frage. Stattdessen wurde mit dem *Java Database Connectivity Data Access API* (JDBC) ein unabhängiges, standardisiertes Call Level Interface geschaffen, mit dem prinzipiell auf jedes (objekt-)relationale Datenbanksystem über eine einheitliche Schnittstelle zugegriffen werden kann. Es ist auf Grundlage der Erfahrungen mit Microsofts ODBC unter Mitarbeit vieler namhafter Datenbankhersteller[2, Kap. 8.1] entstanden. Das einzige was vom Hersteller des DBMS benötigt wird, ist ein passender JDBC-Treiber, der die JDBC-Aufrufe in Aufrufe an die Schnittstelle des Datenbanksystems umsetzt. Die programmseitigen Anpassungen an andere DBMS beschränken sich zumindest theoretisch darauf, einen anderen Treiber für die Verbindung zur jeweiligen Datenbank zu wählen. Die Treiberunterstützung kann dabei auf vier Arten erfolgen [7] (vgl. Abb. 2):

1. Eine JDBC-ODBC-Bridge setzt JDBC-Befehle in ODBC-Aufrufe um. ODBC ist sehr weit verbreitet und wird inzwischen von fast allen Datenbanken unterstützt. Der Nachteil liegt in der Notwendigkeit, ODBC-Code und häufig auch einen DB-spezifischen Client auf jedem Client-Rechner zu installieren. Dadurch werden Installation und Wartung relativ aufwendig, weswegen man die Bridge-Lösung nur als Notlösung betrachten sollte, wenn kein JDBC-Treiber eines anderen Typs vorhanden ist. Zudem ist der Overhead durch zusätzliche Indirektion über ODBC nicht zu vernachlässigen, so daß man für verteilte Systeme mit hoher Transaktionslast auf andere Varianten zurückgreifen sollte.
2. Eine etwas direktere Methode besteht darin, JDBC-Aufrufe in Aufrufe des DB-APIs umzusetzen (*native-API partly Java technology-enabled driver*). Aber auch dazu muß auf jedem Client-Rechner Binärcode des DB-Herstellers installiert werden.
3. Eine dritte Variante übersetzt die JDBC-Aufrufe in ein Datenbank-unabhängiges Netzwerkprotokoll, die dann ein Middleware-Server in API-Aufrufe des jeweiligen DBMS übersetzt (*net-protocol fully Java technology-enabled driver*). Der herstellerspezifische Code muß daher nur auf diesem Server vorhanden sein, wodurch sich die Wartung erheblich vereinfachen läßt. Daher gilt dieser Weg gemeinhin als der flexibelste und findet inzwischen auch recht breite Unterstützung, allerdings ist der Treiberserver natürlich ein potentieller Engpaß für die Systemperfomanz.
4. Als weitere Möglichkeit gibt es Treiber, die JDBC-Aufrufe direkt in das Netzwerkprotokoll des jeweiligen DBMS übersetzen (*native-protocol fully Java technology-enabled driver*). Sie bieten sich besonders für die Verwendung in Intranets an, da hier häufig vergleichsweise homogene Strukturen zu finden sind, die den „Umweg“ von Variante 3 überflüssig machen.

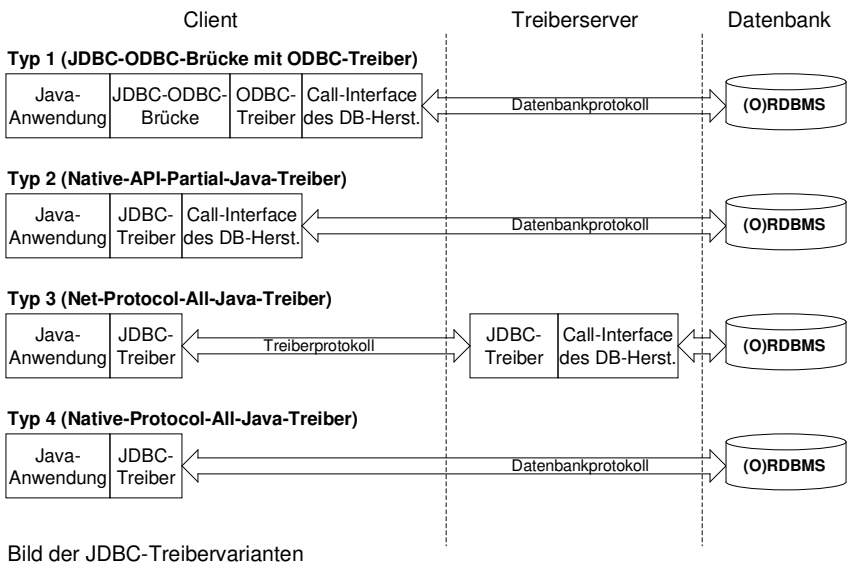


Abb. 2 Treibertypen

Alle großen Hersteller von Datenbanksystemen aber auch Drittfirmen bieten inzwischen JDBC-Treiber an. Zum Zeitpunkt der Entstehung dieses Artikels enthielt die Sun-Datenbank 145 Treiber, wobei Typ 4 momentan am weitesten verbreitet ist. Da bei Typ 1 und 2 DB-abhängiger Code auf dem Client-Rechner benötigt wird, eignen sich diese beiden Varianten nicht für Applets, da diese normalerweise nicht in der Lage sind, Binärdaten vom Server auf den Rechner des Nutzers zu übertragen.

JDBC ist momentan in Version 2.1 erhältlich und besteht aus zwei Teilen: Das *Core Package* (`java.sql.*`) ist bereits in der Standardausgabe des *Java Software Development Kit* (J2SE SDK) enthalten und liefert die Basisfunktionalität für den Zugriff auf Datenbanksysteme. Das sogenannte *Optional Package* ist in der *Enterprise Edition* des SDK (J2EE SDK) enthalten oder als Ergänzung zur Standardausgabe erhältlich. Es ergänzt Mechanismen, die für professionelle Anwendungen unerläßlich sind, wie z.B. *Connection Pooling* oder verteilte Transaktionen. Die Nachfolger-Version 3.0 hat inzwischen den Status *Proposed Final Draft* erreicht und wird in die Version 1.4 des J2SE SDK einfließen.

1.2.1 JDBC Core Package

Zunächst sollen an dieser Stelle die Grundprinzipien von JDBC vorgestellt werden. Als Low-Level-Schnittstelle überläßt JDBC einen erheblichen Teil der Ausnahmebehandlung dem Anwender und nimmt keinerlei Überprüfung der SQL-Statements auf Korrektheit vor. Da aber prinzipiell nur Strings an die Datenbank geschickt werden, ist die Flexibilität bei der Nutzung von DB-spezifischen Features entsprechend groß. Genauso groß ist aber die Gefahr, durch Verwendung von proprietären Funktionen die Portabilität der Anwendung zu verschlechtern. So unterstützen viele DB-Hersteller kein standard-konformes SQL sondern nur einen eigenen Dialekt. Entsprechender JDBC-Code muß also u.U. bei der Portierung auf eine andere Datenbank angepaßt werden. Auf eine mögliche Lösung des Problems wird später im Zusammenhang mit SQLJ hingewiesen.
Die wichtigsten Grundelemente und -klassen von JDBC sollen an einem einfachen Beispiel schrittweise erläutert werden.[8]

1.2.1.1 Laden des Treibers

Als erster Schritt muß der für die Datenbank erforderliche Treiber geladen werden, der natürlich auf dem Client-Rechner vorhanden sein muß. Prinzipiell stehen drei Möglichkeiten zur Verfügung:

1. Explizites Laden mit dem Classloader, der Aufruf

   ```
   Class.forName(driverKlassenName);
   ```

   erzeugt automatisch eine Instanz des Treibers mit dem Namen `driverKlassenName`, der nun dem System zur Verfügung steht.

2. In den System-Properties kann der Datenbanktreiber über das Property

   ```
   jdbc.drivers = driverKlassenName
   ```

   festgelegt werden.

3. Der Treiber wird mit der Methode `registerDriver(driver)` direkt bei der Klasse angemeldet, die den Verbindungsaufbau übernimmt, dem `DriverManager`.

1.2.1.2 Aufbau einer Verbindung zum DB-Server

Als nächstes muß eine Verbindung zum Datenbankserver aufgebaut werden. Auch hier genügt ein Einzeiler, in dem die URL der Datenbank, Benutzername und Paßwort als Strings übergeben werden:

```
Connection conn = DriverManager.getConnection(url, login,
password);
```

1.2.1.3 Statements

Mit dem so erhaltenen *Connection*-Objekt können nun u.a. die Metadaten der Datenbank erfragt, und *Statement*-Objekte erzeugt werden, mit denen man SQL-Befehle an die Datenbank absetzen kann:

```
Statement stmt = conn.createStatement();
```

1.2.1.4 Datenmanipulation

Mit diesem *Statement*-Objekt kann man nun sowohl SQL-Anfragen als auch Änderungen durchführen oder mit DDL-Befehlen das Datenbankschema modifizieren, also Tabellen löschen, ändern oder erzeugen. Zu Durchführung von Änderungen dient

```
Stmt.executeUpdate(updateStatement);
```

mit dem ein beliebiger zur Laufzeit als String generierter SQL-Änderungsbefehl abgesetzt werden kann.

1.2.1.5 Select-Anfragen und Result-Sets

Die Auswertung von Anfragen ist prinzipiell ähnlich einfach, erfordert wegen der mengenorientierten Arbeitsweise von SQL aber weitere Vorkehrungen, um die Ergebnismengen Element für Element abarbeiten zu können. Dazu liefert jede Methode von JDBC, mit der man Anfragen an das DBS stellen kann, `ResultSet`-Objekte als Rückgabewert. Mit dieser Java-Variante des Cursor-Konzepts kann dann die Ergebnismenge Zeile für Zeile in Programmvariablen übertragen und dort verarbeitet werden:

```
ResultSet rs=Stmt.executeQuery(selectStatement);
```

Ein weiteres Problem ist die Konvertierung der Datentypen des Datenbanksystems in passende Java-Datentypen. Im JDBC-Standard ist definiert, auf welche Java-Datentypen die jeweiligen SQL-Datentypen abgebildet werden. So werden beispielsweise SQL- CHARS und -VARCHARs auf Java-Strings umgesetzt. Für komplexere SQL-Typen wie `Date` oder `Timestamp` bringt das JDBC-Package entsprechende Java-Typen mit.
Die Abbildung selbst erfolgt durch den Treiber, der Programmierer kann dann für jede Zeile des *Result Sets* mit `getXXX`-Methoden auf die Spalten über deren Namen oder Index zugreifen, wobei XXX dem Java-Datentyp der Spalte entspricht. Folglich muß er dazu den Java-Typ der jeweiligen Spalte kennen oder durch Zugriff auf die Metadaten herausfinden. Ein kleine Beispielanfrage auf eine Tabelle „Personal“ soll die Vorgehensweise verdeutlichen. „Personal“ verfüge unter anderem über die Attribute „PNR“ (Typ Integer), „Name“, „Vorname“ (beide vom Typ VARCHAR(30)), „Gehalt“ (Typ Money) und „Abteilung“ (VARCHAR (20)):

```
ResultSet rs=Stmt.executeQuery(
          "SELECT NAME, VORNAME, GEHALT
           FROM Personal
           WHERE Abteilung='Verkauf'");

while (rs.next()) {
     String name=rs.getString("NAME");
     String vorname=rs.getString("VORNAME");
     Double gehalt=rs.getDouble("Gehalt");
     System.out.println(name+"   "+vorname+"   "+gehalt);
}
```

Das Beispiel gibt „Name“, „Vorname“ und „Gehalt“ aller Mitarbeiter der Abteilung „Verkauf“ über den *Standard Output Stream* aus. Die Methode `boolean next()` des *Result Sets* liefert so lange *true* zurück, wie noch weitere Tupel in der Ergebnismenge vorliegen und schaltet den Cursor jeweils eine Position weiter.
Der Befehl `execute()` ist allgemeiner gehalten, und kann beliebige SQL-Strings an die Datenbank schicken, `executeBatch()` erlaubt die Ausführung einer Reihe von SQL-Änderungsbefehlen, die vorher mit `addBatch(String sql)` dem Statement übergeben wurden. Wird versucht, ein nicht batchfähiges *Statement* hinzuzufügen (z.B. alle Select-*Statements*), wird eine Ausnahme geworfen.

1.2.1.6 *Prepared Statements*

In vielen Anwendungen werden ähnliche Datenbankanfragen oder Änderungen sehr häufig durchgeführt. Der Parser des DBMS muß diese Befehle jedesmal analysieren, auf Korrektheit prüfen und optimieren, was die Last auf dem DB-Server erhöht und die maximal erreichbare Transaktionsleistung reduziert. Daher gibt es mit den `PreparedStatements` eine

Erweiterung der einfachen *Statements*, mit deren Hilfe parametrisierte SQL-Befehle effizient mehrfach ausgeführt werden können. Wenn der Treiber Precompilation unterstützt, wird das *Statement* beim Erzeugen zur Vorkompilierung an das Datenbanksystem geschickt, was die Abarbeitung unter Umständen erheblich beschleunigen kann, da die Überprüfung des Statements nur einmal ausgeführt werden muß. Bei der Verwendung werden dann nur die jeweils aktuellen Parameter eingesetzt.

1.2.1.7 *Callable Statements*

Um die Netzbelastung niedrig zu halten, ist es oft wünschenswert, die Verarbeitung von Daten möglichst nahe am oder besser noch im DB-Server vorzunehmen. Dazu werden *Stored Procedures* verwendet, die bisher mit einer herstellerspezifischen Sprache erstellt und dann aus der Anwendung aufgerufen werden. Derartige Aufrufe werden in JDBC mit einem `CallableStatement` realisiert.
Mit der Methode `close()` können Verbindungen, Statements und *Result Sets* geschlossen werden. Grundsätzlich können sämtliche Methoden Ausnahmen werfen (`SQLException`), da die Überprüfung der SQL-Anfragen erst zur Laufzeit geschieht. Daher sind sämtliche Methoden in einen `try-catch`-Blöcke zu klammern, auf die in den obigen Beispielen aus Gründen der Übersichtlichkeit verzichtet wurde.

1.2.1.8 Erweiterungen des *Core Package* in JDBC 2.0

Das *Core Package* hat mit JDBC 2.0 einige Erweiterungen erfahren, z.B. gibt es verbesserte *Result Sets*, die eine freie Positionierung des Cursors in der Ergebnismenge gestatten (*Scrollable Result Sets*) (vgl. Abb. 3) und die Modifikation der Daten an der Cursorposition erlauben (*Updateable Result Sets*). Letztere bieten `updateXXX()`-Methoden zur Änderung der jeweiligen Spalte an. Die Umsetzung zwischen Java- und SQL-Typ erfolgt analog zu den `getXXX()`-Methoden. Die Spezifikation des gewünschten Typs von *Result Set* erfolgt durch Parameter bei der Generierung des *Statement*-Objekts.
Neu ist auch die Unterstützung von SQL:1999-Datentypen, mit denen objektrelationale DBMS genutzt werden können:

- BLOBs und CLOBs sind Container für große Binärdatenmengen bzw. Textdaten, wie beispielsweise Bilder oder Audio- und Videodaten. Wegen der Größe dieser Objekte ist ein komplettes Kopieren in das *Result Set* wenig sinnvoll oder wegen Speicherbeschränkungen sogar unmöglich. Daher erhält die Ergebnismenge bei Anfragen auf große Objekte nur Referenzen, sogenannte Lokatoren. Mit dieser Referenz kann dann das Objekt ganz oder in Teilen beim Client materialisiert werden. Eine andere Möglichkeit besteht darin, daß große Objekt als Datenstrom zu lesen. So müssen z.B. bei Videoanwendungen nur die gerade benötigten Teile über das Netzwerk transportiert werden.
- SQL:1999 unterstützt auch mengenwertige Attribute mit dem Array-Typ. Auch diese werden in *Result Sets* nicht materialisiert, Statt dessen erhält man mit der `getArray()`-Methode des *Result Sets* eine Referenz in Form eines Objekts, welches das Array-Interface implementiert. Auf dieses Objekt kann man dann mit `getArray()` die Elemente auslesen.
- Benutzerdefinierte SQL-Datentypen können mit JDBC entweder mit sogenannten Umsetzungsklassen genutzt werden, oder falls keine solchen definiert wurden, über das allgemein gehaltene `Struct`-Interface.
- Wurden in klassischen relationalen Datenbanksystemen Referenzen auf andere Datensätze über die Wertegleichheit eines Fremdschlüssels zum Primärschlüssel des referenzierten

Tupels realisiert, bieten ORDBMS zusätzlich von System vergebene Objekt-IDs auf die Mittels des Datentyps *Reference* verwiesen werden kann. JDBC ermöglicht den Zugriff und die Nutzung dieser Referenz-Typen mit entsprechenden `setRef()` und `getRef()`-Methoden. Mit der so erhaltenen Referenz kann dann z.B. in einer weiteren Anfrage die Ergebnismenge qualifiziert werden.

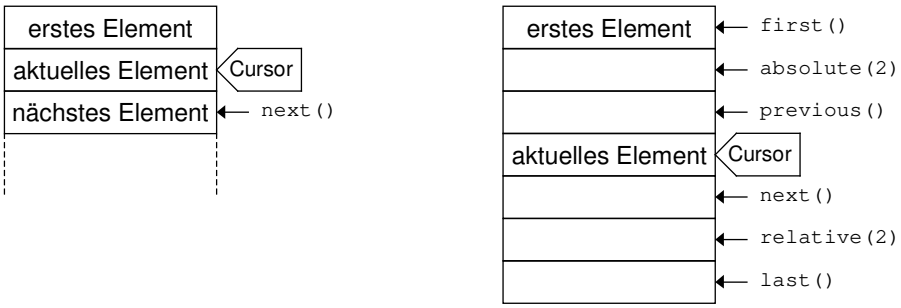


Abb. 3 Result Sets: die ursprüngliche Variante (links) bietet nur minimalen Komfort, während ein Scrollable Result Set ein freies Positionieren des Cursors in der Ergebnismenge erlaubt.

1.2.2 JDBC Optional Package

Das Optional Package von JDBC ergänzt nun weitere Funktionen, die für die Verwendung in Geschäftsanwendungen notwendig sind, bietet aber auch teilweise Ersatz für einige der obigen Mechanismen. Das neue `DataSource`-Interface stellt eine gegenüber dem `DriverManager` verbesserte Methode für die Erstellung einer Verbindung zu einer Datenbank zur Verfügung, mit dem Ziel Portabilität und Wartbarkeit zu verbessern. Ein `DataSource`-Objekt repräsentiert dabei i.a. eine Datenbank, aber eine andere Datenquelle, wie ein Spreadsheet oder auch eine Textdatei in Tabellenform. Der große Vorteil liegt darin, daß von den spezifischen Eigenschaften der Datenquelle abstrahiert wird, die so innerhalb des Programms abstrakt gehandhabt wird.
Das `DataSource`-Objekt wird im allgemeinen von einem separaten Programm erzeugt und bei einem *Naming Service,* wie z.B. das *Java Naming and Directory Interface* (JNDI) angemeldet, wodurch die Anwendung sehr einfach über den logischen Namen auf die Datenquelle zugreifen kann, und keine weiteren Informationen über den benötigten Treibertyp oder URL wissen muß, da alle Informationen über die Datenquelle und ihre Lokalisierung im zugehörigen `DataSource`-Objekt gespeichert sind. Man vermeidet dadurch auch die feste Kodierung des Treibernamens, wodurch die Anwendung unabhängiger vom verwendeten DBMS wird. Wird die Datenquelle auf einen anderen Server verlagert, muß nur die jeweilige Eigenschaft in der `DataSource`-Beschreibung angepasst und dem JNDI bekannt gemacht werden, statt wie bisher alle auf diese Datenquelle zugreifenden Clients zu modifizieren. Der Aufbau einer Verbindung sieht dann folgendermaßen aus:

```
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup("jdbc/BeispielDB");
Connection con = ds.getConnection("Login","1234Passwort");
```

Das so instanziierte `Connection`-Objekt kann dann wie eine `Connection` verwendet werden, die von einem `DriverManager` erzeugt wurde, bietet aber darüber hinaus noch weitere Möglichkeiten. Da der Aufbau einer Verbindung zur Datenbank ein aufwendiger

Vorgang ist, wird mit *Connection Pooling* versucht, die Anzahl der neu zu erzeugenden Verbindungen gering zu halten, indem nicht mehr benötigte Verbindungen wiederverwendet werden. Die Nutzung dieses Mechanismus ist denkbar einfach, denn die Erzeugung einer solchen Verbindung erfolgt über das `DataSource`-Objekt genauso wie mit einfachen Verbindungen. Unterstützt die Implementierung der Datenquelle *Connection Pooling*, wird automatisch eine wiederverwendbare `Connection` zurückgeliefert. Auch deren Nutzung unterscheidet sich nicht von der Verwendung einer gewöhnlichen Verbindung. Lediglich das Schließen der Verbindung sollte in einem `finally`-Block im Anschluß an einen `try-catch`-Block erfolgen, um sicherzustellen, daß die `Connection` auch im Fehlerfall wieder freigegeben wird. Weiterhin erlaubt das `DataSource`-Interface die einfache Nutzung von verteilten Transaktionen. Sie geschieht wie das *Connection Pooling* völlig transparent, wenn die Datenquelle verteilte Transaktionen unterstützt. Allerdings ist bei der Verwendung von verteilten Transaktionen die später noch näher beschriebene anwendungsseitige Kontrolle der Transaktionsgrenzen nicht möglich.
Eine Erweiterungen der JDBC *Result Sets* stellt das `RowSet`-Interface dar. Sie können als *JavaBeans* verwendet und daher mit entsprechenden graphischen Werkzeugen bearbeitet werden. Sie können entweder ständig mit der Datenquelle verbunden sein, oder aber offline die Daten zur Verfügung stellen (`CachedRowSet`).

1.3 SQLJ [2, Kap.9]

Eingangs wurde erwähnt, daß es grundsätzliche zwei Möglichkeiten zum Datenbankzugriff gibt, ein *Call Level Interface* oder *embedded SQL*. *Embedded SQL* kann für den Programmierer eine nicht unerhebliche Arbeitserleichterung sein, da es von der fehlerträchtigen Verwendung des jeweiligen Call Level Interfaces abstrahiert.
Ein Konsortium von führenden DB-Herstellern, u.a. IBM, Oracle und Informix, sowie JavaSoft definierte folglich mit SQLJ einen entsprechenden Standard. Er gestattet nicht nur die Einbettung von SQL in Java-Programme, sondern bietet auch die Möglichkeit, in Java *Stored Procedures* zu implementieren sowie die Ergänzung von SQL durch die Verwendung von Java-Klassen als Datentypen.

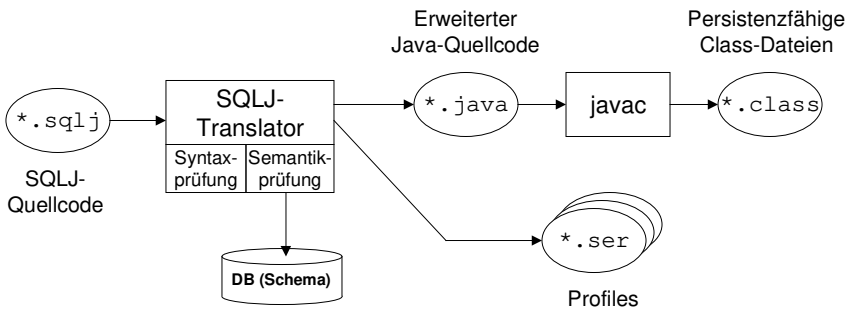
1.3.1 Einbettung von SQL in Java-Programme

Teil 0 der SQLJ-Spezifikation beschreibt die Einbettung von SQL in Java-Programme. Ein Präprozessor, auch Generator genannt, erzeugt dann aus diesen Befehlen entsprechenden Java-Code, der mit dem normalen Java-Compiler übersetzt werden kann. Die Vorteile, welche die Einbettung von SQL für die Entwicklung und auch den späteren Einsatz der Anwendung bringt, sind vielfältig:

- Die kurze und prägnante Schreibweise erleichtert nicht nur die Entwicklung, sie verbessert auch Lesbarkeit und Verständlichkeit, und trägt damit zu besserer Wartbarkeit der Anwendung bei.
- Die Überprüfung der SQL-Syntax kann bereits zur Übersetzungszeit erfolgen, was nicht unerhebliche Performanz-Vorteile zur Laufzeit bringt.
- Auch die Semantik der SQL-Statements kann bereits zur Laufzeit gegen das bestehende Datenbankschema validiert werden.
- Die Anpassung des Codes an verschiedene Datenbanken, die trotz JDBC-Standard noch an vielen Stellen notwendig ist, erfolgt jetzt einfach durch die Verwendung eines anderen Präprozessors.

Natürlich kommt solch ein Gewinn an Komfort und Geschwindigkeit nicht ohne einen Preis. Eingebettetes SQL ist grundsätzlich statisch, die Befehle stehen also zur Übersetzungszeit in

ihrer Grundstruktur fest und können lediglich parametrisiert werden. Oft benötigt man jedoch dynamisch zur Laufzeit erzeugt SQL-Anweisungen, so daß nicht vollständig auf JDBC verzichtet werden kann. Aber auch wenn kein dynamisches SQL benötigt wird, benötigt der Entwickler grundlegendes Wissen über JDBC, insbesondere zur Ausnahmebehandlung. Der Präprozessor kann nicht voraussehen, wie möglicherweise auftretende *Exceptions* gehandhabt werden sollen. Daher müssen SQLJ-Statements i.a. durch `try-catch`-Blöcke geklammert sein.
Die Syntax von SQLJ unterscheidet sich etwas von der von Java. SQLJ-Befehle werden mit einem `#sql` eingeleitet, gefolgt von der SQL-Anweisung in geschweiften Klammern und einem Semikolon. Für Select-Statements müssen zusätzlich sogenannte Iteratoren definiert werden. Aus diesen werden entsprechende Java-Klassen generiert, die den Zugriff auf die Ergebnismenge ermöglichen. Ihre Verwendung erfolgt analog zu der von *JDBC Result Sets*.
Der Austausch von Daten zwischen SQLJ und dem Java-Programm erfolgt über Host-Variablen. Diese werden innerhalb des SQLJ-Befehls durch einen vorangestellten Doppelpunkt gekennzeichnet.
Die Verbindung zur Datenbank erfolgt über sogenannte Verbindungskontexte (`ConnectionContext`). Diese basieren auf JDBC-Verbindungen und werden auch genauso genutzt: Man übergibt dem Kontruktor die Position der Datenbank als URL sowie Nutzername und Passwort. Alternativ kann für eine bereits existierende `Connection` ein Verbindungskontext erstellt werden. Durch Verwendung mehrere Verbindungskontexte wird der parallele Zugriff auf mehrere Datenbanken oder aus mehreren Threads auf die gleiche Datenbank möglich.
Das Vorübersetzen der SQLJ-Programme unterscheidet sich etwas von der Vorgehensweise üblicher Präprozessoren für eingebettetes SQL, weshalb es an dieser Stelle ausführlicher vorgestellt werden soll(vgl. Abb. 4). Zunächst kann der Präprozessor eine Überprüfung der SQL-Syntax vornehmen. Besteht bereits eine Verbindung zur Datenbank kann auch die Semantik der SQL-Befehle gegen das Datenbankschema verifiziert werden.
Als Besonderheit im Vergleich zu anderen eSQL-Übersetzern produzieren die SQLJ-Übersetzer im Anschluß an die Überprüfungen nicht direkt einen um JDBC-Aufrufe erweiterten Quellcode. Statt dessen enthält das generierte Java-Programm lediglich Aufrufe an das SQLJ-Laufzeitmodul (Package `sqlj.runtime.*`). Dort wird dann dynamisch auf die ebenfalls erzeugten *Profiles*, serialisierbare Java-Klassen, zugegriffen. Diese enthalten die eigentlichen JDBC-Befehle. Dadurch genügt für die Anpassung an ein DBMS eines anderen Herstellers ein Austausch der *Profile*-Dateien, man spricht hierbei von *Customizing*.



Übersetzung von SQLJ-Programmen

Abb. 4 SQLJ-Übersetzung

1.3.2 *Server-sided Java* mit SQLJ

Ein großes Problem für die Portabilität von Datenbankanwendungen stellen bisher *Stored Procedures* dar, also benutzerdefinierte Routinen, die im Datenbankserver selbst ablaufen. Diese sind äußerst effizient, da weniger Daten über Netzwerkverbindungen zum Client transportiert werden müssen, so daß man in vielen Situationen, bei denen Systemleistung eine Rolle spielt, nur ungern auf sie verzichten möchte. Allerdings hat bisher jeder Hersteller seine eigene Sprache für *Stored Procedures* verwendet. Soll die Anwendung also portabel sein, müssen für jedes zu unterstützende DBMS eigene *Stored Procedures* geschrieben werden. Dies macht Änderungen extrem aufwendig und damit teuer.
Der SQLJ-Standard beschreibt im ersten Teil, wie man *Stored Procedures* mit Java implementieren kann. Damit steht die erste portable *Stored Procedure*-Sprache zur Verfügung, die von jeden Datenbankhersteller für sein System unterstützt werden kann. Man spricht hier von *Server-sided Java*, von Suns Marketing-Abteilung auch gerne als „300% Java“ bezeichnet, also je 100% auf jeder der drei elementaren Schichten der heutigen *Three-Tier*-Architekturen. Auch die automatisierte Installation oder das Entfernen einer *Stored Procedure* in bzw. aus einem Datenbanksystem, wird unterstützt. Dazu sind normalerweise einige SQL-Befehle erforderlich. Um diesen Vorgang zu vereinfachen, können diese im sogenannten *Deployment Descriptor* der Prozedur bzw. Funktion beschrieben werden.

1.3.3 DB-seitige Verwendung von Java-Datentypen

Der zweite Teil der SQLJ-Spezifikation beschreibt Verfahren, um Java-Datentypen als SQL-Datentypen zu verwenden. Umgekehrt können aus bestehenden SQL-Datentypen automatisch entsprechende Java-Wrapper-Klassen erzeugt werden.
Die so erstellten SQL-Typen können ganz normal verwendet werden. Allerdings werden Konzepte wie `private`-Variablen nicht von SQL unterstützt, es sind also grundsätzlich alle Attribute des Typs zugänglich.
Die Teile 1 und 2 des SQLJ-Standards sind bisher noch nicht verabschiedet, es existieren jedoch bereits einige Implementierungen, die jedoch von SQLJ- und SQL:1999-Standard abweichen.

1.4 J2EE-Connector

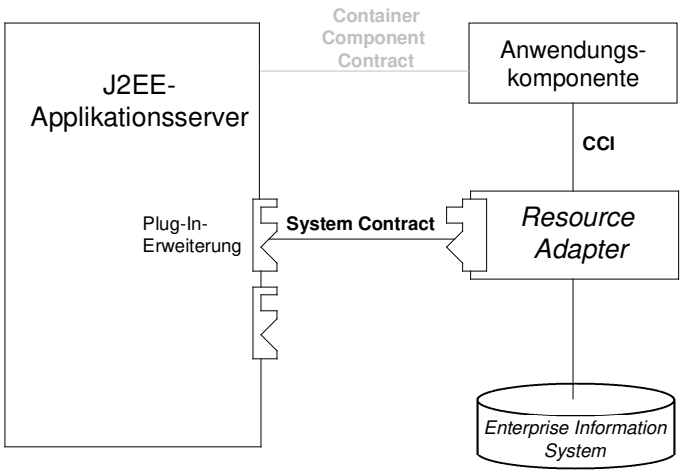


Abb. 5: J2EE Connector Architecture

1.4.1 Motivation

Mit JDBC und SQLJ stehen also weitgehend ausgereifte und gut unterstützte Schnittstellen zur Verfügung, um auf relationale Datenbanken zuzugreifen. Doch wie erfolgt die Nutzung der anderen eingangs erwähnten *Enterprise Information Systems*, wie z.B. ERP-Systeme? Bisher existierte dazu kein Standard, so daß Anwendungen, die derartige Systeme nutzen, auf proprietäre Erweiterungen der Hersteller des Informationssystems zurückgreifen mußten. Dadurch verschlechtert sich jedoch die Portabilität der Anwendung.
Langfristiges Ziel von Sun ist es daher, eine einheitliche Schnittstelle für den Zugriff auf alle Typen von Informationssystemen zu ermöglichen. Dazu befindet sich momentan die *J2EE Connector Architecture* in der Entwicklung. Sie liegt momentan als *Version 1, Final Draft* vor und befindet sich in der *public review stage* [7]. Jegliche Kooperation der Anwendungen mit dem *Enterprise Information System* läuft über diese Standardschnittstelle ab und erlaubt so dem Server, sämtliche Mechanismen der Systemschicht, wie Transaktionen, Sicherheit und Verbindungsmanagement in einer für den Anwendungsentwickler transparenten Weise zu kontrollieren.

1.4.2 Resource Adapter

Um Unterstützung für die *Connector Architectur*e anbieten zu können, muß der Hersteller eines Informationssystems dazu lediglich einen *Resource Adapter*, also eine Art Treiber, für sein System entwickeln, der dann in jeden beliebigen J2EE-Server mit *Connector*-Unterstützung integriert werden kann. Die Anwendungskomponenten selbst nutzen für den Zugriff auf das Informationssystem ein Client-API, das in einer von zwei Ausprägungen zur Verfügung gestellt werden kann:

1.4.2.1 *Resource Manager*-spezifisches API

Die erste Variante besteht aus einem API, das an den zugrundeliegenden *Resource Manager* angepaßt ist. Ein Beispiel dafür ist JDBC für relationale Datenbanken, daß durch die *Connector Architecture* nicht ersetzt sondern ergänzt werden soll.

1.4.2.2 Common Client Interface

Die wünschenswerte Realisierung ist jedoch in Form des *Common Client Interface* (CCI, Package `javax.resource.cci`) gegeben [9]. Es stellt eine Grundfunktionalität zur Verfügung, auf der dann erweiterte Funktionalität realisiert werden kann. Wie bei JDBC baut man auch mit dem CCI eine Verbindung zum *Resource Manager* auf, indem man die Methode `getConnection` der Klasse `ConnectionFactory` aufruft. Man kann als Parameter zusätzlich Client-spezifische Sicherheitsinformationen übergeben (z.B. Login, Passwort) oder diese Aufgabe an die Laufzeitumgebung delegieren. Was in JDBC die *Statements* sind, sind bei der Connector-Architektur die `Interaction`-Objekte. Über diese können mit `execute`-Methoden verschiedene Aktivitäten auf den Informationssystem durchgeführt werden, die mittels `InteractionSpec`-Objekten genauer definiert werden.
Diese enthalten dazu *Standard Properties* wie den Namen der Funktion, Interaktionsweise und Timeouts, die mit *Setter*- und *Getter*-Methoden modifiziert werden. Weiterhin läßt sich die Ergebnismenge spezifizieren, z.B. die Anzahl der gewünschten Ergebnisse.
Die Ein- und Ausgabe erfolgt über `Record`-Objekte, welche die Datenstrukturen des EIS in Java repräsentieren. Diese können als *Custom Records* mit einem Werkzeug zur

Entwicklungszeit erstellt werden, das die Meta-Daten des Informationssystems zusammen mit der Typabbildung berücksichtigt und in der Implementierung des Records verwendet. „Generische“ *Records* greifen dagegen erst zur Laufzeit auf ein Metadaten-Repository zu, um von dort Meta-Information und Abbildungsvorschriften zu erhalten. Sie werden von einer `RecordFactory` instanziiert.
Das Basisinterface `javax.resource.cci.Record` wird von drei spezialisierten Record-Interfaces erweitert:

- Ein `MappedRecord` erweitert `java.util.Map` und ermöglicht so die ungeordnete Speicherung von Schlüssel-Wert-Paaren.
- Ein `IndexedRecord` erweitert `java.util.List`, womit der gezielte Zugriff auf Elemente möglich ist, da eine Sortierung beibehalten wird.
- Ein `ResultSet`, welches `java.sql.ResultSet` erweitert und die Verarbeitung von tabellenartigen Informationen ermöglicht.

1.4.3 Kontrakte

Zusätzlich zu dem sogenannten *Container Component Contract* zwischen J2EE-Server und den darin ablaufenden Komponenten, auf den im Kapitel zu *Enterprise Java Beans* noch näher eingegangen wird, spezifiziert die Connector-Architektur mehrere *Kontrakte* zwischen J2EE-Server und *Enterprise Information System*, um eine standardisierte Schnittstelle zwischen diesen beiden Systemen zu etablieren und so die Austauschbarkeit von J2EE-Server bzw. Informationssystem zu ermöglichen:

- Ein *Kontrakt* für Verbindungsmanagement zwischen Applikationsserver und EIS, der neben dem grundlegenden Verbindungsaufbau auch *Connection Pooling* gestattet.
- Ein *Kontrakt* für Transaktionsmanagement, der es den Anwendungen beispielsweise ermöglicht, Transaktionen zu nutzen, die sich über mehrere Informationssysteme erstrecken.
- Ein *Sicherheitskontrakt* zum Schutz der Informationen des Informationssystems mittels Authentifizierungs- und Authorisationsmechanismen.

Die Connector-Architektur wird erstmals in der J2EE-Plattform, Version 1.3 vollständige Unterstützung finden, von der momentan eine Beta-Version erhältlich ist.

Nachdem nun die Grundzüge des Zugriffs auf die *Enterprise Information Systems* mit Java-Technologie vorgestellt wurde, soll nun die Umsetzung der eingangs geforderten Integritäts- und Sicherheitsanforderungen mit J2EE erläutert werden.

1.5 In aller Kürze: Das Transaktionsparadigma

Zur Realisierung müssen die Informationssysteme weitreichende Vorkehrungen treffen [6]. Man spricht in diesem Zusammenhang vom Transaktionsparadigma, welches gewisse Eigenschaften beim Zugriff auf die Informationssysteme garantiert. Eine Transaktion bildet dabei einen Geschäftsvorgang der Realität im Informationssystem nach und garantiert dabei vier wesentliche Eigenschaften, die man oft unter dem Kürzel ACID zusammenfaßt [5].

- Atomicity (Atomarität): Eine Transaktion wird als kleinste, nicht weiter unterteilbare Einheit aufgefaßt, die entweder vollständig abgearbeitet oder komplett zurückgesetzt wird („Alles oder Nichts“-Prinzip).
- Consistency (Konsistenz): Eine Transaktion hinterläßt das Informationssystem in einem konsistenten Zustand, d.h., alle Integritätsbedingungen, die auf den Daten definiert sind,

müssen am Ende der Transaktion erfüllt sein, oder sie wird komplett zurückgesetzt. Zwischenzustände während der Transaktionsverarbeitung dürfen jedoch inkonsistent sein.
- Isolation: Da Informationssysteme i.a. von vielen Clients gleichzeitig genutzt werden, muß sichergestellt sein, daß die Auswirkungen nebenläufiger Transaktionen für diese unsichtbar bleiben. Man spricht vom „logischen Einbenutzerprinzip".
- Durability (Dauerhaftigkeit): Änderungen einer erfolgreich abgeschlossenen Transaktion bleiben dauerhaft in der Datenbank gespeichert, auch bei Systemausfällen. Das Revidieren einer solchen Transaktion ist nur durch eine sogenannte „kompensierende Transaktion" möglich.

Eine Steuerung der Transaktionsverwaltung der Datenbanksysteme ist bereits mit JDBC möglich. Der Entwickler hat die Wahl zwischen automatisiertem *Commit*, indem er im aktuellen Connection-Objekt die Methode `setAutoCommit(boolean on)` mit dem Parameter *true* aufruft. Er kann aber auch selbst die Transaktionsgrenzen setzen, dazu dienen die Methoden `commit()` und `rollback()`. Ein explizites Starten einer Transaktion ist nicht nötig, der erste SQL-Befehl bzw. der jeweils erste nach einem Commit beginnt eine neue Transaktion. Dieser Ansatz ist jedoch etwas unflexibel, entweder der Entwickler liefert sich und seine Anwendung dem Automatismus aus, oder er muß sich um korrekte Transaktionsklammerung selbst kümmern. Soll sich das Transaktionsverhalten einer Komponente ändern, muß immer auch der Code geändert und die Komponente neu kompiliert werden. Zudem ist es mit JDBC nicht möglich, Transaktionen zu starten, die sich über mehrere Informationssysteme erstrecken.

1.6 Java Transaction API

1.6.1 Motivation

Wegen dieser Einschränkungen und der Bedeutung von Transaktionen für Geschäftsanwendungen hat Sun das *Java Transaction API* (JTA) entwickelt. Es ermöglicht zum einen die Nutzung von mehreren Informationssystemen unter dem Schutz einer einzigen Transaktion, und stellt zum anderen dem Entwickler zwei Möglichkeiten zur Verfügung, Transaktionen zu implementieren:
Die naheliegende, als *programmatic transaction demarcation*, bezeichnete Variante erfordert ähnlich wie bei JDBC die Markierung der Transaktionsgrenzen durch den Anwendungsentwickler mittels Aufrufen der Methoden `begin()` und `commit()` des Interfaces `UserTransaction` von JTA. Man beachte, daß hier im Gegensatz zu JDBC der Entwickler selbst entscheiden kann, ob seine Operationen unter Transaktionsschutz stattfinden. Dies erlaubt einerseits eine bessere Kontrolle des Transaktionsverhaltens, erhöht aber auch gleichzeitig die Fehleranfälligkeit. Daher wurde eine zweite Methode vorgesehen, *declarative transaction demarcation* genannt, die von den *Enterprise Java Beans* unterstützt wird, auf die später noch genauer eingegangen wird. In diesem Zusammenhang spricht man auch von Container-kontrollierter Transaktionssteuerung. Dabei beschreibt der Entwickler der *Bean* die Transaktionsanforderungen seiner Komponente im *Deployment Descriptor*, indem er für jede Methode ein Transaktionsattribut setzt. Die Laufzeitumgebung der *Bean* übernimmt dann dementsprechend die Transaktionsklammerung.

1.6.2 Transaktionsattribute

Es stehen sechs Transaktionsattributen zur Auswahl:

- *Required*: Dieses Attribut stellt sicher, daß die Methode immer innerhalb einer Transaktion aufgerufen wird. Besteht beim Aufruf bereits ein Transaktionskontext, wird dieser verwendet, andernfalls wird eine neue Transaktion begonnen. Dieses Attribut ist die empfohlene Standardeinstellung, da so einerseits Transaktionsschutz gewährleistet ist und problemlos mehrere dieser Methodenaufrufe unter einer gemeinsamen Transaktion geklammert werden können.
- *RequiresNew*: Beim Aufruf der Methode wird grundsätzlich eine neue Transaktion gestartet, die mit der Rückkehr aus dem Aufruf beendet wird. Besteht beim Aufruf bereits ein Transaktionskontext, wird dieser für die Dauer der Ausführung suspendiert und anschließend wieder aufgenommen. Die Verwendung von *RequiresNew* wird dann empfohlen, wenn die Methode auf jeden Fall zum Abschluß kommen muß, unabhängig vom Verlauf der umgebenden Transaktion. Dies ist beispielsweise beim *Logging* sinnvoll.
- *NotSupported*: Besteht beim Aufruf der Methode bereits ein Transaktionskontext, wird dieser suspendiert und dann die Methode ausgeführt. Dies wird häufig dann benötigt, wenn auf ein Informationssystem zugegriffen wird, das nicht in den J2EE-Server integriert ist. Dieser hat damit keine Kontrolle über die Transaktionsausführung dieses Informationssystems, und der Zugriff erfolgt nicht innerhalb einer JTA-Transaktion.
- *Supports*: Besteht ein Transaktionskontext zum Aufrufzeitpunkt, wird dieser auch für die Ausführung der Methode verwendet, andernfalls erfolgt die Ausführung ohne Transaktionsschutz. Von der Verwendung von *Supports* wird in vielen Quellen abgeraten, da es zu einem indeterministischen Verhalten der Methode je nach Kontext des Aufrufers führen und infolgedessen zu einer Verletzungen der ACID-Kriterien kommen kann.
- *Mandatory*: Die Ausführung der Methode muß unbedingt innerhalb eines bereits bestehenden Transaktionskontextes stattfinden, das Starten einer neuen Transaktion genügt nicht. Besteht beim Aufruf kein Transaktionskontext, wird eine *Exception* geworfen.
- *Never*: Die Methode darf nicht innerhalb eines Transaktionskontextes ausgeführt werden, aber auch das Suspendieren eines bestehenden Kontextes ist nicht zulässig. Besteht ein solcher, wird eine *Exception* geworfen, andernfalls wird die Methode ohne Transaktionsklammerung ausgeführt.
  Mandatory und *Never* bieten sich an, wenn man den Kontext des Aufrufers verifizieren möchte, reduzieren aber die Wiederverwendbarkeit der Komponente.

Es wird generell empfohlen, die deskriptive Festlegung des Transaktionsverhaltens zu verwenden. Neben der eher geringfügigen Entlastung des Entwicklers liegt der entscheidende Vorteil darin, daß sich das Transaktionsverhalten einer Komponente auch nachträglich über den *Deployment Descriptor* ändern läßt, ohne daß dazu Code-Änderungen notwendig sind. Dadurch läßt sich die Komponente problemloser wiederverwenden.
Wird auf ein Informationssystem ohne JTA-Transaktionen zugegriffen, erfolgt dieser Zugriff im allgemeinen innerhalb einer lokalen, *Resource Manager*-spezifischen Transaktion, ohne Wissen und Kontrolle des J2EE-Applikationsservers.
Sowohl in der direkt vom Entwickler kontrollierten Variante von JTA-Transaktionen, als auch bei der deklarativen Beschreibung, liegt die Hauptlast der Transaktionsverwaltung bei der J2EE-Plattform, welche die *Low-Level*-Transaktionsprotokolle implementiert und auf die entsprechenden Mechanismen des jeweiligen Informationssystems abbildet.
Für den Entwickler einer Anwendung ist es wichtig zu wissen, welche Transaktionsfähigkeiten fester Bestandteil des J2EE-Standards sind, und daher von jeder J2EE-Plattform unterstützt werden müssen, und welche nur optional sind. Soll die Anwendung maximale Portabilität aufweisen, sollte man sich auf die Minimalanforderungen beschränken. Beispielsweise können J2EE-Anwendungen grundsätzlich verteilt ablaufen, das dazu notwendige 2-Phasen-Commit für verteilte Transaktionen ist jedoch nicht im J2EE-

Standard festgeschrieben, und wird deswegen nicht von jedem J2EE-Produkt unterstützt. Geschachtelte Transaktionen werden bisher generell nicht unterstützt.

1.6.3 Verwendung von JTA-Transaktionen

Es ist generell empfehlenswert, alle Zugriffe auf Informationssysteme über JTA-Transaktionen abzuwickeln, da man nur so mehrere Informationssysteme unter eine gemeinsame Transaktion stellen kann. Kann eine Teiloperation nicht erfolgreich abgeschlossen werden, kann automatisch ein kompletter Rollback erfolgen.
Ohne JTA-Transaktionen ist es im Fehlerfall wesentlich schwieriger, die einzelnen lokalen Teiltransaktionen rückgängig zu machen, da sie bereits erfolgreich abgeschlossen sein können. Dies kann im allgemeinen nicht mehr automatisiert erfolgen, sondern muß von der Anwendung übernommen werden (in Form einer „kompensierenden Transaktion“). Problematisch ist allerdings, daß es nicht immer möglich ist, die Effekte einer erfolgreichen Transaktion rückgängig zu machen. Zudem kann die gegenseitige Isolation bei nebenläufigen Zugriffen nicht sichergestellt werden. Kommt es zu einem Absturz des Servers, versagt dieser Ansatz sogar völlig.
Dennoch kann diese Vorgehensweise notwendig werden, beispielsweise wenn auf mehrere Informationssysteme innerhalb einer einzigen Transaktion zugegriffen werden soll, aber nicht alle dieser Informationssysteme in den J2EE-Server integriert werden können.

2. *Enterprise Java Beans Tier*

Nachdem nun die Grundkonzepte für den Zugriff und die Nutzung von Informationssystemen vorgestellt wurden, soll im folgenden die Entwicklung von serverseitigen Geschäftsapplikation mit J2EE erläutert werden.

2.1 *Business Objects*

Bei Geschäftsanwendungen gilt es zunächst, die sogenannte *Business Logik* zu realisieren. Darunter versteht man im breiteren Sinne eine Menge von Richtlinien und Regeln, welche zusammengenommen die jeweilige Geschäftsaktivität umsetzen. Bei rechnergestützten Unternehmensvorgängen kann man diese Definition präzisieren. Im heute weit verbreiteten objektorientierten Entwicklungsansatz wird jede Geschäftsaktivität in eine Menge von Komponenten oder Elemente zerlegt, die man als Geschäftsobjekte oder *Business Objects* bezeichnet. Jene Regeln, die Struktur, internes Verhalten und Interaktion dieser Objekte beschreiben, bezeichnet man dann als *Business Logik*.

2.1.1 Gemeinsame Anforderungen

Business Objects sind im allgemeinen in ihrer Funktionalität recht stark an die zu lösende Aufgabe und die Anwendungsdomäne gebunden und daher sehr unterschiedlich. Dennoch haben sie viele gemeinsame Anforderungen zu erfüllen:

- **Wiederverwendung**
  Sie wird heute vor allem durch Nutzung von Klassen- oder Funktionsbibliotheken erreicht. Dies wird jedoch häufig als niedrigste Ebene von Wiederverwendung bezeichnet. Wirklich entscheidende Vorteile bei der Anwendungsentwicklung erhofft man sich erst durch Wiederverwendung in einer gröberen Granularität. Durch das Zusammensetzen von neuen Anwendungen aus größeren Komponenten, die eine benötigte Funktion bereits

vollständig realisieren, erreicht man nicht nur erheblichen Kosteneinsparungen, sondern auch geringere Fehlerraten und erhöhte Stabilität, was zu einer Verbesserung der gesamten Produktqualität führt. Gleichzeitig wird die Entwicklungszeit (*Time to Market)* verkürzt, was besonders in schnellebigen Branchen wie dem E-Commerce entscheidend sein kann, um sich einen Vorsprung vor den Mitbewerbern zu sichern. Auch die Kosten für Wartung und Erweiterung der Anwendung können durch überlegten Einsatz von Wiederverwendung deutlich reduziert werden.
Um die Auswahl an verfügbaren Komponenten möglichst groß zu machen, ist eine gewisse Standardisierung erforderlich, sowohl bei der eigentlichen Entwicklung als auch bei der genutzten Laufzeitumgebung. Ein breit akzeptierter Standard würde es gestatten, Anwendungen aus zugekauften (*off the shelf*) Komponenten zusammenzustellen. Zudem würde man unabhängig von proprietären Lösungen, wie sie heute von zahlreichen Herstellern angeboten werden.

- **Teilnahme an Transaktionen**
  Da viele Geschäftsvorfälle, die mit Geschäftsobjekten realisiert werden, zwischen zwei Parteien ablaufen, muß es die Möglichkeit geben, „Vertragsrecht“ in Rechnersystemen nachzubilden. Dazu trägt das bereits vorgestellte Transaktionsparadigma bei. Folglich müssen die Geschäftsobjekte in die Lage versetzt werden, an Transaktionen teilzunehmen und auch die Steuerung des Transaktionsverhaltens zu kontrollieren.

- **Erhaltung des Systemzustands**
  Hier wird zunächst unterschieden zwischen Objekten, die ihren Zustand nur für die Dauer der Interaktion mit einem Client bewahren, und solchen bei denen der Zustand unabhängig vom jeweiligen Client dauerhaft gespeichert werden muß. Man spricht dabei von *conversational* bzw. *persistent state*. So muß beispielsweise ein „Konto“-Objekt seinen Zustand dauerhaft bewahren, hat also einen *persistent state*, während ein „Warenkorb“, der die vom Kunden in einem Web-Shop ausgewählten Produkte speichert, diesen Zustand nur solange behalten muß, bis eine Bestellung erfolgt oder der Kunde das virtuelle Geschäft verlässt.

- **Operationen auf gemeinsamen Daten**
  Durch *Business Objects* realisierte Geschäftsvorgänge sind häufig eng miteinander verwoben und nutzen daher oft gemeinsame Daten. Dies erfordert entsprechende Mechanismen zur Kontrolle von Nebenläufigkeit und zur gegenseitigen Isolation, um sogenannte Mehrbenutzeranomalien zu vermeiden.

- **Bedienung einer großen Zahl von Clients**
  Mit dem Aufkommen des E-Commerce sind zusätzlich viele neue Anwendungsgebiete entstanden. Diese ermöglichen es theoretisch Tausenden von Clients, die Geschäftsanwendung gleichzeitig zu nutzen.
  Es gilt nun jedem Client den Eindruck zu vermitteln, daß ihm private *Business Objects* für seine Anfragen zur Verfügung steht, ohne dabei die begrenzten Ressourcen des Servers zu verschwenden. Dies erfordert *Load Balancing* in Form von Instanzenmanagement, d.h., der Server sorgt für die gezielte Kreierung und Terminierung der jeweils benötigten Geschäftsobjekte. Zudem sollte ein einzelnes Objekt in der Lage sein, mehrere Clients zu bedienen, ohne daß es dabei zu Mehrbenutzer-Anomalien kommt.

- **Remote-Zugriff**

Da die Geschäftsanwendungen im allgemeinen nicht auf dem gleichen Rechner laufen, auf dem die Clients arbeiten, müssen diese in die Lage versetzt werden, die Dienste der Anwendung über ein Netzwerk zu nutzen. Auch die Geschäftsobjekte selbst sind Elemente eines verteilten Rechnersystems und es müssen entsprechende Mechanismen verfügbar sein, um die räumliche Verteilung vor dem Anwendungsprogrammierer und dem Nutzer zu verbergen und gleichzeitig auch zu einem gewissen Grad Ausfallsicherheit zu gewährleisten.

- **Zugriffskontrolle**
  Viele der von Geschäftsobjekten angebotenen Dienste sind in gewisser Weise sicherheitskritisch und erfordern daher eine zuverlässige Authentifizierung der Dienstnutzer, um nur berechtigten Clients Zugriff zu gewähren.

Viele dieser gemeinsamen Anforderungen der Geschäftsobjekte an die Laufzeitumgebung erfordern nicht nur enormen zeitlichen und technischen Aufwand bei der Implementierung, sie liegen auch außerhalb der Domäne der meisten Anwendungsprogrammierer.
Statt alle diese Anforderungen für jede Anwendung erneut zu realisieren, sollte ein standardisiertes *Framework* zur Verfügung stehen, das diese realisiert. Der Entwickler kann dann auf diese über standardisierte Schnittstellen zugreifen, und wird so von systemnahen Aufgaben entlastet
Ziel ist es, durch die Abstraktion von diesen Systemdiensten die Entwicklung von Geschäftsanwendungen zu beschleunigen, gleichzeitig die Kosten zu reduzieren und dennoch die Qualität zu steigern. Der Anwendungsentwickler kann sich dadurch voll auf sein eigentliches Spezialgebiet konzentrieren: die Realisierung der Geschäftslogik.
Bisher boten zahlreiche Hersteller entsprechende Produkte an. Doch handelt es sich dabei im allgemeinen um Komplettlösungen, die proprietäre Technologien und divergente APIs nutzen und so die Anwendung an die Produkte des Herstellers binden.

2.2 *Enterprise Java Beans*

Es bestand also Handlungsbedarf für die Entwicklung eines herstellerübergreifenden Standards. Hier ist Sun in die Bresche gesprungen und hat zusammen mit zahlreichen Partnern aus der Industrie *Enterprise Java Beans* (EJBs)als verteilte Komponententechnologie für Applikationsserver entwickelt. Unter den zur Zeit circa 40 Partnern finden sich große Datenbankhersteller wie IBM und Oracle, Hersteller von Applikationsservern wie BEA, Fujitsu-Siemens, Compaq und viele andere.

2.2.1 Grundlagen

In den *Enterprise Java Beans* konvergieren zahlreiche bereits länger erhältliche Java-Technologien, darunter JDBC, JNDI, *JavaBeans*, *Servlets* und das „Write Once, Run Anywhere“-Konzept.
Es gibt zwei grundlegende Typen von *Enterprise Java Beans*, auf die später noch im Detail eingegangen wird: *Entity Beans* repräsentieren im allgemeinen gemeinsame Daten, während *Session Beans* die eigentliche Anwendungslogik realisieren. Wichtig ist hierbei die Unterscheidung der EJBs von den herkömmlichen *Java Beans* [3 S.10]. Letztere sind feingranulare Komponenten, die nicht alleine lauffähig sind. Sie bilden somit keine vollständige Applikation und dienen nur dazu, die Entwicklung von Anwendungen zu erleichtern. *Enterprise Java Beans* im Gegensatz sind einsatzbereite Komponenten, die in speziellen Laufzeitumgebungen, sogenannten EJB- *Containern* ablaufen.

2.2.1.1 Container

Der J2EE-Standard definiert dazu lediglich die Anforderungen an die Laufzeitumgebung, ohne dabei Vorgaben zur Realisierung zu machen. Dadurch können J2EE-konforme Produkte von zahlreichen Herstellern angeboten werden, auf denen alle in Form von *Enterprise Java Beans* implementierten Komponenten, die sich vollständig an den Standard halten, lauffähig sind. Man wird als Anbieter einer Anwendung dadurch unabhängig von einem einzelnen Hersteller, da die Migration auf einen anderen standardkonformen Server wesentlich erleichtert wird.
Prinzipiell sind EJB-*Container* eine gegenüber der Standard-Ausgabe von Java 2 erweiterte virtuelle Maschine für Anwendungskomponenten, und sind selbst Teil eines EJB-Servers, der grundlegende Systemfunktionalitäten, wie z.B. Verzeichnisdienste, zur Verfügung stellt. EJBs, *Server* und *Container* bilden zusammen den EJB-*Tier* der J2EE-Plattform.
Die Aufgabe der *Container* ist es, eine Infrastruktur für die *Enterprise Java Beans* zur Verfügung zu stellen. Dazu gehört die Verwaltung des Lebenszyklus einer *Bean*, also Erzeugung und Terminierung. Dadurch kann auch *Load Balancing* durch den *Container* erfolgen, indem intensiv benutzte Objekte mehrfach instanziiert werden.
Weiterhin werden Mechanismen für die persistente Speicherung des Zustands einer Komponente benötigt. Auch der Zugriffsschutz für sicherheitskritische Komponenten und Methoden zur sicheren Authentifizierung der Clients kann durch die Laufzeitumgebung erfolgen.
Clients interagieren dabei nicht direkt mit den *Beans*, sondern mit dem *Container*, der dabei quasi als Vermittler fungiert. Dadurch kann der *Container* Dienste erbringen, die sich über mehrere Methodenaufrufe oder sogar über mehrere verteilte *Container* erstrecken, wie beispielsweise automatische Transaktionsklammerung oder die transparente Kommunikation über Systemgrenzen.

Um diese Funktionalität erbringen zu können, müssen sich *Enterprise Java Beans* an gewisse Vorgaben, den *Container Component Contract* halten. Er besteht aus drei Teilen, im Prinzip nur eine Menge von Interfaces und Klassen, die jede EJB implementieren bzw. erweitern muß:

- Das Interface `javax.ejb.EJBHome`, kurz *Home Interface,* fast alle Methoden für die Verwaltung des Lebenszyklus einer *Bean* zusammen. Es ermöglicht dem Client die Erzeugung und Löschung von EJBs und das Erfragen der Metadaten. Weiterhin kann der Client Instanzen von *Entity Beans* über ihren Primärschlüssel suchen.
- Das *Remote Interface* `javax.ejb.EJBObject` definiert die eigentliche Client-Sicht auf die *Bean*, d.h. die Business-Methoden. An diese Schnittstelle erfolgen also die eigentlichen Methodenaufrufe. Weiterhin kann man mit dem *Remote Interface* eine Referenz auf das *Home Interface* der EJB erlangen, sie aus dem System entfernen, einen Handle auf die *Bean* anfordern und bei *Entity Beans* den Primärschlüssel erfragen.
  Home- und *Remote Interface* werden zusammen auch als Exportschnittstelle bezeichnet.
- Die eigentliche Implementierung der *Enterprise Java Bean*, die je nach Typ das Interface `javax.ejb.EntityBean` oder das Interface `javax.ejb.SessionBean` implementiert, stellt den zweiten Teil der Mechanismen dar, die vom *Container* kontrollierte Dienste ermöglichen. Diese Klasse wird vom *Container* aufgerufen, wenn der Client die entsprechenden Methoden des *Remote Interface* aufruft. Insbesondere enthält sie auch die Methoden zur Erzeugung und zum Auffinden der *Beans*, die von *Home* und *Remote Interface* angeboten werden.

Um Konflikte mit den Management-Aufgaben des Containers zu vermeiden, dürfen *Enterprise Java Beans* bestimmte Operationen nicht selbst durchführen:
- die Nutzung von *Threads*
- der direkte Zugriff auf Dateien mit dem java.io-Package
- das Darstellen von Informationen oder das Abfragen von Benutzer mit dem Abstract Windowing Toolkit (AWT) oder Swing
- der Zugriff auf Netzwerk-Sockets

Sämtliche Interaktionen mit dem Nutzer erfolgen statt dessen entweder durch Client-seitigen Code oder mittels Komponenten des Web-Containers und einem Browser.

2.2.2.2 Deployment-Deskriptoren

Die gute Wiederverwendbarkeit und Flexibilität von *Enterprise Beans* ergibt sich auch durch eine weitere Besonderheit: Das Verhalten der *Enterprise Beans* ist nicht vollständig durch ihre Implementierung festgeschrieben, vielmehr werden dort hauptsächlich die fachlichen Aspekte, also die Geschäftlogik realisiert. Zu einem späteren Zeitpunkt, man spricht von *Deployment*, also der Anpassung der *Bean* an die zu erstellende Applikation, werden dann die technischen Eigenschaften im sogenannten *Deployment Descriptor*, einer speziellen XML-Datei beschrieben. Diese Beschreibung ist dabei praktisch unabhängig vom Quellcode. Unter anderem können so das Transaktionsverhalten, der Zugriffsschutz und sogar Bezeichner und Typeigenschaften zu einem gewissen Grad geändert werden, ohne den Quellcode der *Bean* neu zu kompilieren. *Deployment*-Deskriptoren gibt es dabei auf nahezu jeder Ebene in der Komponentenhierarchie. Dadurch können mit ihnen auch beispielsweise die Beziehungen zwischen den *Beans* eines *Packages* modelliert werden, was später noch für die *EJB Query Language* benötigt werden wird.
Was auf den ersten Blick als sinnvolle, aber nicht sonderlich revolutionäre Technik wirkt, stellt eigentlich ein komplett neues Programmierparadigma dar: Mit der Trennung der Entwicklung in die Zeitphasen Kodierung und *Deployment* können zum ersten mal zumindest Teilaspekte einer Applikation ein deklarativ spezifziert werden.

2.2.2 Remote Method Invocation (RMI)

Nachdem nun die Grundidee von *Enterprise Java Beans* vorgestellt wurde, sollen noch einige technische Aspekte angesprochen werden. Da EJBs grundsätzlich verteilte Objekte sind, muß eine Technologie zur Verfügung gestellt werden, welche die möglichst transparente Kommunikation von Objekten über verschiedene virtuelle Maschinen hinweg ermöglicht, die sich auch auf verschiedenen Hosts befinden können. *Remote Method Invocation* ermöglicht genau dieses: ein Java-Programm kann Methoden eines entfernten Objekts aufrufen, sobald es eine Referenz auf das Objekt hat. Diese kann ihm als Parameter oder Rückgabewert übergeben worden sein oder über einen *Naming Service* erfragt werden. Damit Objekte per RMI ihre Dienste anbieten können, müssen sie ein *Remote Interface* implementieren, in dem alle Methoden spezifiziert sind, die das Objekt zur Verfügung stellt. Die eigentliche Kommunikation zwischen Aufrufer und dem aufgerufenen Objekt erfolgt über zwei weitere Klassen, *Stub* und *Skeleton* genannt, die mit einem eigenen Compiler automatisiert generiert werden können (Abb. 6). Die *Stub*-Klasse wird dem Aufrufer bei der Anfrage an den Naming-Dienst übergeben und fungiert dann als lokaler Stellvertreter des entfernten Objekts. An den *Stub* richtet er seine Methodenaufrufe, der sie dann zusammen mit den Parametern über das Netzwerk an die *Skeleton*-Klasse übertragen, die dann selbst die entsprechende Methode des Objekts aufruft und eventuelle Rückgabewerte über den *Stub* zurück zum Client schickt. Bei der Verwendung des neuen *Stub*-Protokolls von Java 2 kann das Skeleton auch entfallen. Seine Funktionen werden dann von generischem Code übernommen.[10]

Die Übertragung von Objekten über das Netzwerk erfolgt durch Serialisierung: das zu übertragende Objekt und die von ihm erreichbaren Objekte werden in einen Bytestrom kodiert und auf der Gegenseite rekonstruiert. Diese Kodierung kann durch den Entwickler der Klasse selbst erfolgen, oder man verwendet die Standard-Kodierung von Java.

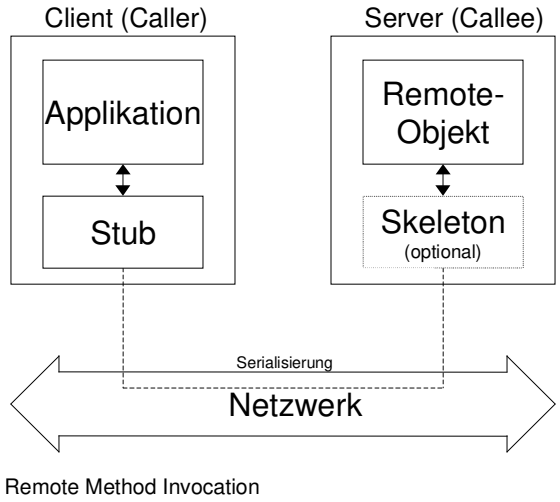


Abb 6. Remote Method Invocation

2.2.3 RMI over IIOP

Während RMI eine RPC-Schnittstelle für die Kommunikation zwischen Java-Klassen ist, stellt *RMI over IIOP* (*RMI over Internet Inter ORB Protocol*) oder kurz *RMI-IIOP* eine Erweiterung des RMI-Mechanismus dar. Über diesen gemeinsam von Sun und IBM entwickelten Mechanismus wird Java um CORBA-Fähigkeiten erweitert. Dadurch wird die transparente Kommunikation von Java-Komponenten mit Komponenten, die in anderen Sprachen entwickelt wurden, ermöglicht. Wie mit RMI ist es möglich, die komplette Spezifikation des *Remote*-Interfaces in Java vorzunehmen, es muß keine eigenständige *Interface Defintion Language* (IDL) erlernt werden. Diese Interfaces können dann in jeder von der OMG (von CORBA) unterstützten Sprache implementiert werden. Genauso können Clients, die in anderen Sprachen entwickelt werden, die IDL-Interfaces nutzen, die aus der Java-Interface-Beschreibung erstellt wurden.

Nach den Grundlagen und einigen technischen Voraussetzungen sollen im folgenden die beiden elementaren *Bean*-Typen näher vorgestellt und ihre Einsatzgebiete diskutiert werden.

2.2.4 Entity Beans

Entity Beans werden im allgemeinen genutzt, um persistente Daten der Anwendung zu repräsentieren. Die *Bean* fungiert dabei als *Wrapper* für diese Daten, um Zugang und Manipulation zu erleichtern.

2.2.4.1 Eigenschaften

Entity Beans unterstützen in hohem Maße die Wiederverwendung. Sie können durch die strikte Kapselung als Komponente in verschiedenen Anwendungen genutzt werden. Beispielsweise könnte eine „Kunde“-*Bean* sowohl für die Verwaltung von Bestellungen, die Personalisierung einer Webseite oder für Marketingzwecke zum Einsatz kommen. Eventuelle

Anpassungen an die jeweilige Anwendung könnte mit Hilfe des *Deployment Descriptor* geschehen, beispielsweise durch Modifikation von Transaktionsattributen (vgl. 1.6.2).

Entity Beans ermöglichen darüber hinaus nebenläufigen Zugriff durch mehrere Clients, und existieren auch über die Dauer einer Sitzung hinweg. Werden Update-Operationen auf einer *Entity Bean* während eines Server-Absturzes gemacht, wird Ihr Zustand automatisch auf den Zustand am Ende der letzten erfolgreichen Transaktion zurückgesetzt.

2.2.4.2 Verwendung

Diese Eigenschaften prädestinieren *Entity Beans* für gewisse Einsatzzwecke, von denen einige im folgenden exemplarisch näher vorstellen werden sollen:

- Modellierung einzelner Tabellenzeilen: Je nach Anwendung kann es sinnvoll sein, jedes Tupel einer Datenbank als eigenes Objekt zur repräsentieren. Dann sollte die Realisierung durch eine *Entity Bean* erfolgen, die Methoden zur Erzeugung, Lokalisierung und Manipulierung des Tupels anbietet. Dabei sollte man jedoch nicht vergessen, daß jeder Aufruf auf eine *Enterprise Bean* ein Zugriff über ein Netzwerk sein kann. Selbst wenn *Bean* und Aufrufer in der gleichen virtuellen Maschine arbeiten, ist immer noch kostspielige Interprozeßkommunikation notwendig. Die Aufrufe und Ergebnisse müssen über den *Container* weitergeleitet werden und es werden Überprüfungen bzgl. Sicherheit und Transaktionseigenschaften durchgeführt.
- Repräsentation komplexer Objekte: Aus diesen Gründen sollte man bevorzugt komplexere *Business Objects*, deren Zustand in einem persistenten Speicher gehalten wird, und deren Operationen sich hauptsächlich mit der Manipulation dieses Zustands befassen, als *Entity Beans* realisieren.
- Nebenläufiger Zugriff durch mehrere Clients: Wenn mehrere Clients den Zustand eines Geschäftsobjekts benötigen, sollte man es als *Entity Bean* modellieren, da diese Objekte ihren Zustand über Methodenaufrufe hinweg bewahren. Wichtig ist dabei, daß dieser Zustand global, also für alle Clients gültig ist. Der *Bean*-Entwickler kann dann darauf vertrauen, daß Zugriffe auf die *Bean* über Synchronisationsmaßnahmen durch den EJB-Server geschützt ablaufen.

2.2.4.3 Persistenz

Entity Beans repräsentieren also persistente Objekte innerhalb der Anwendung. Die Implementierung der Persistenz kann auf zwei Arten geschehen:

- *Bean-managed*: Der Entwickler implementiert selbst die nötigen Methoden innerhalb der *Bean* oder in einem gesonderten *Access Object* mit JDBC, um die Daten der *Bean* in einen persistenten Speicher zu schreiben. Dies gestattet einerseits mehr Flexibilität und Kontrolle für den Entwickler, erschwert aber andererseits die Anpassung der *Bean* bei einer Schemaevolutionen oder dem Wechsel auf andere Informationssysteme.

- *Container-managed*: Der Entwickler spezifiziert, welche Felder der *Bean* persistent zu machen sind. Der Container generiert mit dieser Information dann die notwendigen Zugriffe auf den persistenten Speicher, wenn die *Bean* innerhalb einer Anwendung zum Einsatz kommt. Dies erfolgt transparent für den Entwickler, unabhängig von Typ und Struktur des persistenten Speichers. Dadurch verbessert sich die Portabilität der *Bean* erheblich, weswegen die Persistenzverwaltung durch den *Container* grundsätzlich vorgezogen werden sollte. Allerdings sind die Fähigkeiten bzgl. der automatisierten Persistenz von *Container* zu *Container* recht unterschiedlich. Daher kann es Fälle geben, in denen der Zustand komplexer Objekte, beispielsweise solche, die mehrere *Joins*

repräsentieren, vom Entwickler selbst gesichert werden muß, wenn die Mechanismen des *Containers* für eine automatisierte Abbildung nicht ausreichen.

2.2.4.4 Interfaces

Im Folgenden werden die wichtigsten Methoden von *Entity Beans* kurz erläutert: [2, Kap 11]

- Die Methode `ejbCreate` der *Home*-Schnittstelle wird vom *Container* an die korrespondierende ejbCreate-Methode der *Bean*-Klasse delegiert, d.h., die Signatur muß übereinstimmen. Bei der Erzeugung wird der zugehöriger persistente Datensatz erstellt und mit den übergebenen Argumenten initialisiert. Danach erfolgt automatisch der Aufruf von ejbPostCreate. Da man zu diesem Zeitpunkt davon ausgehen kann, daß die *Bean* bereits existiert, kann man z.B. ihre *Remote*-Schnittstelle ermitteln und diese bei Bedarf an die *Enterprise Bean* übergeben, welche die Erzeugung veranlasst hat.
- Das Löschen der von *Entity Beans* repräsentierten Daten erfolgt mit der Methode `remove`, die in zwei Ausführungen existiert: Als parameterlose Methode der *Remote*-Schnittstelle, oder mit dem Primärschlüssel als Parameter im *Home*-Interface. Der Code, der den zugehörigen persistenten Datensatz löscht, wird generiert. Allerdings müssen eventuell notwendige Vorkehrungen, um die referentielle Integrität auf Ebene der *Entity Beans* sicherzustellen, manuell vom Entwickler in der Methode `ejbRemove` implementiert werden, die ausgeführt wird, bevor der persistente Datensatz gelöscht wird.
- Zur Synchronisation von *Entity Bean* und zugehörigem Datensatz dienen die Methoden `ejbLoad` und `ejbStore`. Hier kann der Entwickler die Speicherung seiner Daten selbst kontrollieren, was z.B. sinnvoll ist, wenn diese komprimiert gespeichert werden sollen. Die Kompression bzw. Dekompression könnte dann in diesen Methoden erfolgen.
- *Entity Beans* die längere Zeit inaktiv sind, belegen dennoch wertvolle Systemressourcen. Mit der Methode `ejbPassivate` können solche *Beans* vom *Container* durch Serialisierung aller nicht-transienten Felder ausgelagert werden, so daß belegte Ressourcen wie Dateien, Sockets etc. freigegeben werden. Umgekehrt kann eine *Entity Bean* mit `ejbActivate` wieder deserialisiert und reaktiviert werden, wobei benötigte Ressourcen erneut belegt und transiente Felder der *Bean* neu initialisiert werden.
- Ein weitere Neuheit in EJB 2.0 sind die sogenannten ejbHome-Methoden Sie werden nicht auf einer einzigen *Entity Bean* abgearbeitet, sondern global auf allen *Beans* dieses Typs. Sie sind somit mit den `static`-Methoden bei einfachen Java-Klassen vergleichbar.
- Mit den `find`-Methoden der *Home*-Schnittstelle lassen sich *Entity Beans* nach beliebigen Kriterien aufsuchen. Diese Aufrufe werden an `ejbFind`-Methoden der *Bean*-Klasse delegiert. Als Ergebnis erhält der Client entweder eine Referenz auf die *Remote*-Schnittstelle oder eine *Collection* von Referenzen auf die *Remote*-Schnittstellen aller Beans, welche die Kriterien erfüllen. Eine spezielle Behandlung erfolgt für die Methode `ejbFindByPrimaryKey`. Sie hat genau einen Parameter, den Primärschlüssel, und liefert genau ein Ergebnis, weswegen sie automatisch generiert werden kann.
- Seit EJB 2.0 lassen sich mit `ejbSelect-` Methoden allgemeinere Anfragen an *Entity Beans* stellen. Man unterscheidet zwei Typen:
  - Solche mit dem Namensmuster `ejbSelectXy` werden global auf der Menge aller *Entity Beans* eines Objektes ausgeführt.
  - Andere mit dem Namensmuster `ejbSelectXyInEntity` werden auf der *Entity Bean* ausgeführt, auf dem sie aufgerufen werden.

  Als Rückgabetypen kommen dabei je nach Anfrage folgende Objekte in Frage:
  - *Remote*-Schnittstellen von *Entity Beans*, wenn mit dieser Select-Methode nach *Beans* gesucht wird (vgl. ejbFind)

- existenziell abhängige Objekte, die in der *Entity Bean* referenziert werden (nicht bei globalen Anfragen vom Typ `ejbSelectXyInEntity`)
- reguläre Java-Typen, wenn die Anfrage bestimmte Werte der *Bean* ausliest.

2.2.4.5 EJB-Query Language (EJB-QL)

Eine Neuerung in EJB 2.0 ist die *EJB Query Language*, die sich stark an SQL92 orientiert. Mit ihr wird die Mächtigkeit der deklarativen Programmierung, wie sie mit dem *Deployment Descriptor* möglich wurde, noch erheblich erweitert. Es ist mit EJB-QL möglich, für *Entity Beans* mit vom *Container* verwalteter Persistenz die Semantik von Anfrage-Methoden wie `ejbFind`- und `ejbSelect` im *Deployment Descriptor* deklarativ zu formulieren, statt sie selbst in der *Bean* zu kodieren. Das *Deployment Tool* des jeweiligen Server-Herstellers generiert daraus dann entsprechenden JDBC-Code. Dadurch wird die Anwendung unabhängiger vom speziellen SQL-Dialekt des DB-Herstellers. Als Datenmodell der *EJB Query Language* dienen die ebenfalls im *Deployment Descriptor* formulierten Beziehungen zwischen den *Beans*. Mit ihnen wird ein abstraktes Schema beschrieben, auf dem dann die Anfragen formuliert werden können.
Wie eingangs erwähnt, verwendet EJB QL eine SQL-artige Syntax. Es gibt SELECT-, FROM- und WHERE-Klauseln und es ist möglich, den durch Felder definierten Pfaden oder Beziehungen zwischen den *Beans* zu folgen. Die WHERE-Klausel ist dabei optional, während SELECT nur für die Beschreibung der Semantik von Select-Methoden zulässig ist. Anfragen lassen sich dabei zunächst lediglich auf *Entity Beans* formulieren, die im gleichen *Package* definiert sind. Zusätzlich können „entfernte“ *Entity Beans*, also solche die außerhalb des *Packages* liegen, verwendet werden, wenn sie im *Deployment Descriptor* referenziert sind.

2.2.5 Session Beans

Der zweite Typ von *Enterprise Java Beans*, die sogenannten *Session Beans* gibt es in zwei Varianten: *Stateful Session Beans* und *Stateless Session Beans*. Beiden Typen ist gemein, daß sie im Gegensatz zu Entity-Beans keine persistenten Objekte darstellen, sondern eine Aufgabe im System für einen Client erbringen. Dazu nutzen sie im allgemeinen *Entity Beans.*

2.2.5.1 Stateful Session Beans

Stateful Session Beans speichern einen Zustand in Ihren Feldern und in Objekten, die von Ihren Feldern aus zugänglich sind, der nur für die Dauer einer Sitzung aufrechterhalten wird und Client-spezifisch ist (*conversational state). Stateful Session Beans* repräsentieren keine persistenten Daten, aber sie können auf diese zugreifen. Sie existieren i.a. so lange, wie die Verbindung zum Client besteht und man kann sie als logische Erweiterung der Client-Software auf dem Server auffassen.

Mögliche weitere Einsatzgebiete für *Stateful Session Beans* sind u.a.:
- Repräsentation nicht-persistenter Objekte: da *Session Beans* nicht in persistentem Speicher gelagert werden, bietet sich ihre Verwendung an, wenn kurzlebige Objekte modelliert werden, deren Zustand eine Sitzung oder einen Systemabsturz nicht überleben muß.
- Modellierung von Workflows: Geschäftsobjekte, welche die Interaktion anderer Objekte verwalten und dadurch Geschäftsprozesse realisieren, sollten als *Session Beans* realisiert werden.

2.2.5.2 Stateless Session Beans

Stateless Session Beans stellen rein server-seitige Funktionen zur Verfügung und enthalten keinerlei Client-spezfische Daten. So kann eine *Stateless Session Bean* prinzipiell mehrere Clients bedienen, und mehrere Instanzen einer *Bean* verhalten sich exakt gleich. Nicht-Client-spezfische Zustandsinformationen können allerdings gespeichert werden, beispielsweise offene Datenbankverbindungen.
Diese Eigenschaften empfehlen *Stateless Session Beans* insbesondere für folgende Einsatzzwecke:

- Modellierung wiederverwendbarer Dienstobjekte: Geschäftsobjekte, die generische Dienste für alle Clients anbieten, können auf natürliche Weise als *Stateless Session Bean* modelliert werden, da für diese Dienste keine Client-spezifischen Informationen notwendig sind.
- Hohe Leistungsanforderungen: da *Stateless Session Beans* weniger Systemressourcen benötigen, bietet sich ihre Verwendung besonders dann an, wenn sehr viele Clients unterstützt werden müssen. Allerdings kann dieser Vorteil auch sehr schnell in einen Nachteil umschlagen, wenn dadurch die Komplexität der Anwendung auf Client-Seite zu stark steigt, da diese dann die Speicherung von Zuständen übernehmen muß.
- Operationen auf mehreren Tupeln: Objekte, die eine Gesamtsicht auf eine größere Menge von Daten erzeugen, wie beispielsweise Kataloge, werden häufig von allen Anwendern genutzt, und sollten daher als *Stateless Session Beans* repräsentiert werden.
- Unterstützung einer prozeduralen Sichtweise auf Daten: Sollen sich Geschäftsobjekte wie Funktionsaufrufe in prozeduralen Sprachen nutzen lassen, bieten sich *Stateless Session Beans* an: Der Aufrufer stellt alle nötigen Daten zur Verfügung, und das Objekt liefert alle Ergebnisse sofort zurück, die Speicherung von Zuständen ist nicht notwendig.

2.2.5.3 Session Beans als *Facade*

Eine häufig anzutreffende Verwendung von *Session Beans* ist das sogenannte *Facade Pattern.* Dabei dient eine einzelne *Session Bean* als zentraler Einstiegspunkt für alle Clients in die Anwendung, oder man bietet unterschiedliche Fassaden für verschiedene Client-Typen an. Dadurch hält man zunächst die Anzahl der Interaktionspunkte der Clients mit der Geschäftsanwendung gering, und verringert so erheblich die Kopplung zwischen Client und den Komponenten der Anwendung. Somit können innerhalb der Anwendung quasi beliebige Änderungen vorgenommen werden, ohne auf Client-Seite Änderungen vornehmen zu müssen. Lediglich die Client-Schnittstelle der *Session Bean Facade* muß beibehalten werden. Diese Maßnahme erleichtert die Wartbarkeit des Gesamtsystems erheblich.

2.2.6 JMS und Message-driven Beans

Eine Neuerung in EJB 2.0 stellen die *Message-driven Beans* dar, die unter Verwendung des *Java Message Service API* (JMS) kommunizieren.

2.2.6.1 Java Message Service API

Zum Zeitpunkt seiner Entstehung war der Hauptzweck des JMS API, Java-Anwendungen die Kommunikation mit bestehenden nachrichtenorientierten Middleware-Systemen zu ermöglichen. Seither wurde es von zahlreichen Herstellern angenommen und in Produkten verfügbar gemacht. Mit der als Beta-Version vorliegenden Version 1.3 ist es integraler Bestandteil der J2EE-Plattform geworden, so daß es inzwischen eine geeignete Plattform für die nachrichtenbasierte Kommunikation von Geschäftsanwendungen darstellt. Es erlaubt

Java-Anwendungen, Nachrichten untereinander auszutauschen, und darüber hinaus auch die Kommunikation mit *Legacy*-Systemen, die *Messaging* unterstützen. Im Gegensatz zur eher engen Kopplung bei RMI, bei der die an der Kommunikation beteiligten Klassen die jeweiligen *Interfaces* der Gegenstelle kennen müssen, ist die Kopplung mittels JMS eher lose. Die Nachricht wird an einen Austauschort gesendet, von wo sie der Empfänger abholen kann. Dabei müssen Sender und Empfänger nicht zur gleichen Zeit verfügbar sein, sie müssen nicht einmal voneinander wissen. Es muß lediglich Übereinstimmung bezüglich des verwendeten Nachrichtenformats und des Austauschortes bestehen. Dabei wird normalerweise von JMS garantiert, daß eine einzelne Nachricht genau einmal empfangen wird, allerdings kann auf diese Sicherheit auch verzichtet werden, wenn sie nicht benötigt wird. Es gibt einige Anwendungsgebiete, in denen der Einsatz einer nachrichtenbasierten Kommunikation gegenüber synchronen Verfahren wie RMI Vorteile bietet:

- Um die kommunizierenden Komponenten leicht austauschen zu können, kann es erwünscht sein, daß die Komponenten die jeweiligen *Interfaces* der Gegenstelle nicht kennen.
- Eine Anwendung soll auch dann funktionieren, wenn nicht alle Komponenten simultan betriebsbereit sind.
- Eine Anwendung soll Informationen an eine andere Anwendung senden können, ohne eine sofortige Antwort erhalten zu müssen und beim Ausbleiben einer solchen blockiert zu werden.

Seit Version 1.3 der J2EE-Plattform bietet JMS u.a. folgende Funktionalitäten:
- Jede beliebige J2EE-Komponente kann JMS-Nachrichten senden oder synchron empfangen.
- Die neuen *Message-driven Beans* ermöglichen darüber hinaus asynchronen Empfang, der sogar nebenläufig erfolgen kann.
- Sende- und Empfangsoperationen können an JTA-Transaktionen teilnehmen.

Es gibt zwei prinzipielle Verfahren, um Nachrichten auszutauschen.

2.2.6.1.1 Point-to-Point Messaging

Point-to-Point Messaging (PTP) basiert auf den drei Elementen Sender, Empfänger und Nachrichten-Warteschlangen. Jede Nachricht wird an eine bestimmte Warteschlange gesendet, von wo sie von genau einem Empfänger abgeholt wird, wobei dieser Empfang quittiert wird. Die Nachrichten verbleiben so lange in der *Queue*, bis sie abgeholt wurden, oder ihre Lebensdauer abgelaufen ist. Es besteht also keinerlei zeitliche Abhängigkeit zwischen Sender und Empfänger.

2.2.6.1.2 Publish/Subscribe Messaging

Publish/Subscribe-Messaging verwendet einen anderen Ansatz. Die Nachrichten der grundsätzlich anonymen Sender, hier *Publisher* genannt, werden an ein *Topic* gesendet. Interessierte Empfänger oder *Subscriber* können sich an diesem *Topic* anmelden und erhalten fortan jede Nachricht, die an dieses *Topic* gesendet wurde.
Es gibt hier eine etwas stärkere zeitliche Abhängigkeit, da *Subscriber* grundsätzlich nur die Nachrichten empfangen, die nach ihrer Anmeldung an das *Topic* gesendet wurden. In der ursprünglichen Form des *Publish/Subscribe Messaging* müssen die Empfänger selbst aktiv die Nachrichten abholen. JMS hingegen gibt auch zeitweilig passiven Empfängern die Sicherheit, wie bei *Queues* alle Nachrichten zu empfangen, indem es die Empfänger aktiv benachrichtigt.

2.2.6.1.3 Empfang von Nachrichten

Mit JMS ist es möglich Nachrichten sowohl synchron als auch asynchron zu empfangen.
Zum synchronen Empfang ruft der Empfänger eine `receive`-Methode auf. Diese liefert dann entweder die nächste vorliegende Nachricht, oder sie wirkt blockierend bis eine Nachricht eintrifft oder ein Zeitlimit abläuft. Damit wird also das ursprüngliche Problem der blockierungsfreien Kommunikation zwischen Anwendungskomponten nicht zufriedenstellend gelöst.
Asynchroner Empfang erfolgt über einen *Listener*-Mechanismus, ähnlich wie er z.B. bei der GUI-Programmierung verwendet wird. Dazu implementiert der Empfänger das `MessageListener`-Interface. Mit `setMessageListener` registriert er sich bei einem *Topic*. Dieses ruft dann beim Eintreffen einer neuen Nachricht automatisch die Methode `onMessage` des Empfängers auf, in der dann der Inhalt der Nachricht verarbeitet wird.

2.2.6.2 Message-Driven Beans

Die bisher vorgestellten *Enterprise Java Beans* beherrschen nur den synchronen und daher möglicherweise blockierenden Empfang von JMS-Nachrichten. Um blockierungsfreie Kommunikation zu ermöglichen, wurde mit den *Message-driven Beans* ein neuer Typ von *Enterprise Bean* vorgestellt, der durch Implementierung des `MessageListener`-Interfaces asynchronen Empfang ermöglicht. Er läßt sich sinnvoll einsetzen, wenn durch ein Ereignis im System eine Aktivität ausgelöst werden soll, auf deren Beenden man aber nicht warten möchte. So könnte z.B. das Erzeugen eines neuen Kunden in Form einer „Kunde“- *Entity Bean* als Auslöser genutzt werden, um Aktivitäten im Marketing-Bereich auszulösen. Würde dies per RMI erfolgen, müßte die *Entity Bean* auf den Abschluss der Methode in den Marketing-Komponenten warten und wäre folglich für einige Zeit blockiert. Eine asynchron gesendete Nachricht an eine *Message-driven Bean* löst das Problem elegant und effizient.
Message-driven Beans sind in gewisser Weise vergleichbar mit zustandslosen *Session Beans*. Auch ihre Instanzen sind relativ kurzlebig und speichern keine Client-spezifischen Informationen, lediglich offene JMS-Verbindungen oder Objekt-Referenzen auf andere *Enterprise Java Beans*. Es ist somit auch hier möglich, mehrere Instanzen gleichzeitig im System zu halten, um so Nachrichtenströme nebenläufig verarbeiten zu können. Das Verwalten dieser Instanzen erfolgt durch den *Container*, der auch weitere Aufgaben übernimmt, die bei „gewöhnlichen“ `MessageListenern` manuell durchgeführt werden müssen, z.B. registriert er die *Bean* automatisch bei der im *Deployment Descriptor* spezifizierten Nachrichtenquelle. Ein manueller Aufruf von `setMessageListener` ist nicht erforderlich.
Im Unterschied zu anderen *Enterprise Beans* haben *Message-driven Beans* kein *Home*- oder *Remote*-Interface, da sie nicht direkt mit den Clients interagieren.

2.2.7 Allgemeine Entwurfsrichtlinien

Neben den speziellen Richtlinien, die den Entwickler bei der Wahl des „richtigen“ Bean-Typs unterstützen sollen, gibt es allgemeinere Entwurfsrichtlinien, die sich weiteren, vom Entwickler zu treffenden Entwurfsentscheidungen annehmen [1]. Dazu gehört auch die Festlegung, welche Objekte des Systems tatsächlich *Enterprise Java Beans* sein sollten.
Da es sich bei *Enterprise Java Beans* um verteilte Objekte handelt, die einen nicht unerheblichen Ressourcenbedarf und Bandbreitenhunger haben, ist es nicht empfehlenswert, jedes Objekt als EJB zu modellieren. Nur solche Geschäftsobjekte, die direkt mit Clients interagieren, müssen *Enterprise Beans* sein, da es sonst zu Mehrbenutzeranomalien kommen

könnte. Im folgenden möchte ich zwei Beispiele von Komponenten aufführen, welche nicht notwendigerweise als *Enterprise Java Beans* modelliert werden müssen.

2.2.7.1 Access-Objekte

Unter *Access*-Objekten versteht man eigenständige Klassen, die den bei der Verwendung von *Bean-managed Persistence* Zugriff auf Informationssysteme kapseln. Sie werden nur von anderen *Enterprise Beans* aufgerufen und sind daher nicht auf besondere Unterstützung durch den Container angewiesen, weswegen sie aus Effizienzgründen nicht als *Enterprise Java Beans* implementiert werden sollten. Die Verwendung von Access-Objekten bietet einige Vorteile:

- Der Code der EJBs wird einfach und verständlich gehalten.
- Die spätere Migration auf *Container-managed Persistence* wird erleichtert, da nicht im ganzen Code der *Bean* Datenbankaufrufe verstreut sind.
- Die Portabilität zwischen verschiedenen Datenbanken bzw. Datenbankschemas wird verbessert, da lediglich das Access-Objekt ausgetauscht werden muß.

2.2.7.2 Value-Objekte

Um die Anzahl der *Remote*-Zugriffe auf eine *Bean* zu verringern, kann es Sinn machen, statt feingranularen *Get*-Methoden eine einzige Methode anzubieten, die als Ergebnis ein sogenanntes *Value*- Objekt zurückliefert, das alle möglichen Informationen beinhaltet. Es handelt sich dabei um ein serialisierbares Java-Objekt das im *Call By Value*-Verfahren an den Client geliefert wird, der so alle seine weiteren Anfragen an dieses Objekt richten kann. Dies macht insbesondere deshalb Sinn, da im allgemeinen nicht die reine Menge der über ein Netz übertragenen Daten der limitierende Faktor ist, sondern die Anzahl der Kommunikationsvorgänge.

2.2.8 Nutzung von EJBs

Wie kann nun mit *Enterprise Java Beans* eine Anwendung realisiert werden? Da die *Beans* keine direkte Nutzerinteraktion erlauben, müssen sie über Dritte mit den Nutzern kommunizieren. Das kann z.B. dadurch geschehen, daß ein Client-seitiges Java-Programm über einen JNDI-Service auf eine als Einstiegspunkt oder Fassade vorgesehene *Session Bean* zugreift, über deren *Remote-Interface* er dann seine Geschäftstätigkeiten ausführt. Alternativ können *Enterprise Beans* auch von Komponenten des *Web-Containers* wie Servlets oder JSPs genutzt werden, die dann dem Client ein auf HTML basierendes Browser-Interface anbieten. Mit der genauen Realisierung der Nutzerinteraktion wird sich die dritte Ausarbeitung zum Thema J2EE näher befassen. Im folgenden soll lediglich ein einfaches Beispiel für eine EJB-basierte Anwendung vorgestellt werden, die Aspekte der Präsentationsschicht vernachlässigt.
Die Anwendung realisiert Teile eines Pizza-Service-Systems. (vgl. Abb. 7) Die Kunden können beispielsweise über einen Browser auf das System zugreifen. Als zentraler Einstiegspunkt dient die „PortalBean“, die das *Facade Pattern* realisiert. Über diese werden die beiden zentralen Interaktionsmöglichkeiten, das Anzeigen der Speisekarte und das Aufgeben einer Bestellung angeboten. Die Speisekarte ist für alle Kunden gleich, und könnte als *Stateless Session Bean* realisiert sein. Zu Darstellung der Information greift sie auf die als *Entity Beans* modellierten Pizzas zu. Die Bestellung erfordert dagegen das Speichern eines Client-spezifischen Zustands, beispielsweise die bereits ausgewählten Pizzas, und sollte als *Stateful Session Bean* („BestellenBean“) implementiert werden. Sie greift für die Durchführung ihrer Funktion auf drei *Entity Bean*-Typen zu: die „PizzaBeans“ werden

beispielsweise für das Berechnen der Bestellsumme benötigt. Die Daten des Kunden werden in Form einer „KundeBean“ persistent gemacht, wenn es sich um einen Erstbesteller handelt. Anschließend wird die Bestellung selbst als *Entity Bean* gespeichert. Sie kann dann für viele Funktionen genutzt werden, die hier nicht aufgeführt sind, um das Beispiel nicht unnötig aufzublähen. So könnte der Chef als weiterer Client-Typ Statistiken über die Lieblingspizzas seiner Kundschaft wünschen, oder für die Monatsabrechnung einen Überblick über die Umsätze benötigen.
Nachdem die Bestellung gesichert ist, muß nun das Backen der Pizza veranlaßt werden. Dies sei an dieser Stelle über eine JMS-Nachricht realisiert, die von der „BestellenBean“ an eine *Message-driven Bean* „NaechsterAuftragBean“ gesendet wird. Diese sichert den Auftrag persistent im System, bis er erledigt ist, und benachrichtigt den zweiten Client-Typ „Küchenpersonal“ über den vorliegende Auftrag.

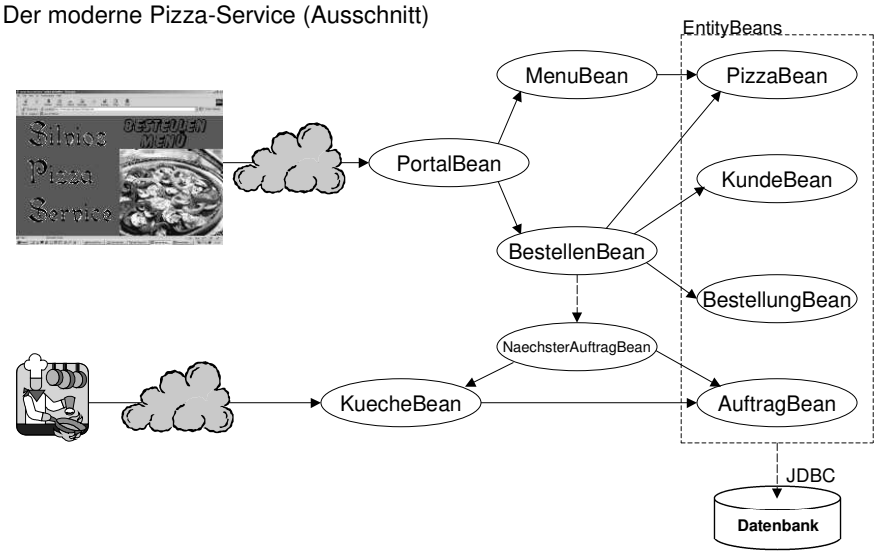


Abb. 7 EJB-Beispiel

3. Zusammenfassung und Ausblick

Informationssysteme bilden die Grundlage jeder modernen Geschäftsanwendung, da sie die Persistenz der Daten sicherstellen. Allerdings stellt die Vereinheitlichung des Zugriffs auf diese Systeme wegen ihrer Heterogenität eine Herausforderung dar, da sie in sehr vielen verschiedenen Ausprägungen auftreten können. Die J2EE-Plattform bietet mit dem JDBC-API und SQLJ bereits heute eine ausgereifte Schnittstelle für den Zugriff auf den überwiegenden Anteil der Systeme, die (objekt-)relationalen Datenbanksysteme.
Die noch bestehenden Lücken im Standard, die z.T. die Verwendung von proprietären Interfaces nötig machen, werden zunehmend geschlossen. Mit der offiziellen Verfügbarkeit der *Connector*-Architektur wird erstmals ein Standard existieren, der eine große Menge heterogener Datenquellen unter einer gemeinsamen Schnittstelle vereint. Dabei wird JDBC seine Bedeutung nicht verlieren, und insbesondere im Bezug auf die Unterstützung von SQL:1999 mit der Version 3.0 noch um einiges ausgereifter werden.
Über die Anforderungen an eine persistente Datenhaltung hinaus wurden weitere gemeinsame Anforderungen vorgestellt, die eine verteilte Geschäftsanwendung an ihre Laufzeitumgebung hat. Die mittlere Schicht des klassischen Drei-Schichten-Modells hat sich dabei als der ideale Platz für die Realisierung dieser Anforderungen erwiesen. Die J2EE-Plattform stellt mit *Enterprise Java Beans* eine *Framework* zur Verfügung, das nicht nur erstmals die

zahlreichen, bisher eher lose zusammenhängenden Java-APIs zu einem großen Ganzen zusammenfaßt, sondern als offener Standard auch unabhängig von der Implementierung eines einzelnen Herstellers ist. Dadurch hat der Anwender jederzeit die Möglichkeit, die Plattform zu wechseln, was einen nicht unerheblichen Investitionsschutz darstellt. Zudem erhofft man sich durch den offenen Standard die Bildung eines einheitlichen und damit größeren Marktes für fertige Anwendungskomponenten, im Gegensatz zu den eher eng begrenzten Märkten für proprietäre *Frameworks.* Trotz dieser entscheidenden Vorteile hat Java noch nicht die erhoffte Verbreitung im Serverbereich erfahren. Es gibt zwar bereits zahlreiche Anbieter von Applikationsservern und Entwicklungswerkzeugen, aber bisher nur wenige Anbieter von Applikationen und Komponenten [11]. Mit der zunehmenden Reife der J2EE Plattform könnte sich die jedoch schnell ändern. Ein Zeichen für die steigende Akzeptanz von serverseitigem Java ist die Entwicklung von zahlreichen Web-Portalen auf EJB-Basis. So ist beispielsweise die Anwendungslogik des „Firmenfinanzportals“ der Dresdner Bank ebenso wie die Seite der Deka Bank komplett mit EJBs realisiert. Wenn sich diese Systeme bewähren, könnte in absehbarer Zeit der erhoffte Durchbruch für die J2EE-Plattform erfolgen.

Anhang

Quellenverzeichnis

[1] 1. N. Kassem: Designing Enterprise Applications with the JavaTM 2 Platform, Enterprise Edition (Blueprints), ISBN 0-20-1702770, http://java.sun.com/j2ee/blueprints/aboutthebook.html.

[2] J. Zimmermann, G. Beneken: Verteilte Komponenten und Datenbankanbindung, Addison- Wesley, München, 2000.

[3] S. Allamaraju et. al.: Professional Java Server Programming J2EE Edition, Wrox Press Ltd., Birmingham, UK, 2000.

[4] E. Roman: Mastering Enterprise Java Beans, and the Java 2 Platform, Enterprise Edition, Wiley Computer Publishing, New York, 1999.

[5] C.J. Date: An Introduction to Database Systems (Seventh Edition), Addison-Wesley Publ. Comp., Reading, Mass., 2000

[6] Härder, T., Rahm, E.: Datenbanksysteme - Konzepte und Techniken der Implementierung, Springer-Verlag, März 1999, 600 Seiten.

[7] http://java.sun.com/j2ee Die offizielle Seite vom J2EE-„Erfinder“ Sun. Umfassende Informationen zu allem was mit J2EE zu tun hat.

[8] http://java.sun.com/docs/books/tutorial/jdbc Ein Tutorial für angehende JDBC-Programmierer

[9] http://java.sun.com/j2ee/download.html#connectorspec J2EE *Connector Architecture Specification*

[10] http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmi-arch2.html

[11] http://java.sun.com/products/ejb/industry.html