4 Objektorientierte Middleware

In diesem Kapitel werden objektorientierte Technologien für Middleware vorgestellt. Dabei zeigt sich der deutlich gestiegene Komfort im Vergleich zu Sockets und RPCs. Zunächst wird die *Common Object Request Broker Architecture* (CORBA) vorgestellt, die von der Object Management Group (OMG) standardisiert wurde – ein Konsortium von über 1000 Mitgliedern weltweit. Die Entwicklung von CORBA musste zu Beginn mit dem *Distributed Component Object Model* (DCOM) von Microsoft konkurrieren; bei Windows 2000 wurde DCOM in COM+ umbenannt.

Für die Entwicklung verteilter Applikationen wurde durch die Java-Plattform *Remote Method Invocation* (RMI) kostenlos bereitgestellt. So kann man heute auch einige RMI-Server antreffen. Prinzipiell kann man die reine RMI-Technologie als eine Alternative zu einem puristischen Object Request Broker (ORB) sehen.

Während die OMG über viele Jahre hinweg ein Komponentenmodell für CORBA diskutiert hat, wurde durch Enterprise JavaBeans (EJB) von Sun Microsystems ein De-facto-Standard für Komponenten in Java vorgeschlagen. Die Verbreitung der EJB-Technologie hat dann auch dazu geführt, dass sich die OMG zügig auf das CORBA Component Model (CCM) geeinigt hat.

In den nachfolgenden Abschnitten werden die vier objektorientierten Middleware-Technologien CORBA, COM+, RMI und EJB vorgestellt. Danach wird ergänzend auf Technologien im WWW-Umfeld eingegangen, und es werden Servlets und Java Server Pages (JSP) erläutert.

Damit sind verschiedene Technologien vorgestellt worden, mit denen man verteilte Systeme entwickeln kann. Es sind also lediglich die Grundlagen für eine Komponententechnologie gelegt worden. Deshalb wird zum besseren Verständnis abschließend auf *JavaBeans* eingegangen, die auf der Client-Seite mittlerweile die dominierende Komponententechnologie darstellen.

4.1 CORBA und IIOP

Die Object Management Group (OMG) wurde 1989 gegründet, um Standards für verteilte, heterogene Umgebungen zu schaffen. Heute hat das OMG-Konsortium über 1000 Mitglieder weltweit, welche die ganze Spannbreite von Herstellern über Entwickler bis zu Endanwendern abdecken. Bei jedem einzelnen Standard handelt es sich deshalb zwangsläufig um einen sinnvollen Kompromiss, der sich an den Bedürfnissen der Praxis orientiert.

Wenn bei der OMG ein Standardisierungsvorschlag eingereicht wird, so muss dabei auch die Praxistauglichkeit nachgewiesen werden (Proof of Concept). Beispiels-

weise wurde der Vorschlag für den Naming Service in Version 1.1 von BEA Systems, Cooperative Research Centre for Distributed Systems Technology (DSTC Pty Ltd.), Inprise und IONA Technologies als Dokument orbos/98-10-11 bei der OMG eingereicht. Diese Firmen bzw. Institute hatten damals bereits Implementierungen, die sich in der Praxis bewährt hatten.

Der wichtigste Standard der OMG ist die *Common Request Broker Architecture* (CORBA). Zu Beginn der 90er-Jahre wurden die ersten Produktentwicklungen für CORBA gestartet; z.B. wurde IONA 1991 in Dublin gegründet, und das Produkt Orbix war 1993 die erste kommerziell erhältliche CORBA-Implementierung für C++ und C.

Ab der Mitte der 90er-Jahre wurden die ersten seriösen Anwendungsentwicklungen mit CORBA-Produkten durchgeführt wurden, und so findet man große Referenzen z.B. bei:

- *Boing Commercial Airplane Group (BCAG).* Bestehende Entwurfs-, Produktions-, Wartungs- und Verkaufssysteme werden durch das Projekt *Define and Control Airplane Configuration/Manufacturing Resource Management (DCAC/MRM)* entweder ersetzt oder integriert. Davon sind 70 Niederlassungen, 200 Server und bis zu 45.000 Benutzer betroffen.
- *Hongkong Telecom Interactive Multimedia Services (HT IMS).* Die HT ist einer der weltweit größten Telekommunikationsabieter, und CORBA wurde als Plattform für interaktives Fernsehen, Video-on-Demand, Home Shopping, Home Banking und Internetzugang ausgewählt.
- *Deutsche Bahn AG.* Die Anwendung DaRT (*Da*tenbank für *R*eisezugwagen und *T*riebfahrzeuge) ist seit 1996 im produktiven Einsatz und wurde sukzessive ausgebaut. Über 800 Benutzer arbeiten mit DaRT an den verschiedensten Standorten in Deutschland, um den Fahrzeugbestand zu pflegen und die Instandhaltung zu planen.

Dieser Abschnitt wird einen Überblick über CORBA geben und die wesentlichen Prinzipien an praktischen Beispielen erläutern. Der Detaillierungsgrad kann deshalb nicht so ausgeprägt sein, wie bei einem ausschließlich auf CORBA ausgerichteten Buch. Die Darstellung ist aber so ausgelegt, dass ein Vergleich mit Enterprise JavaBeans (EJB) ermöglicht wird.

4.1.1 Object Request Broker (ORB)

Im Mittelpunkt von CORBA steht der *Object Request Broker* (ORB), der als Objektbus die Kommunikation zwischen Server und Clients abwickelt, wie das in Abbildung 4.1 dargestellt ist. Ein ORB kann Aufrufe zwischen Objekten unter verschiedenen Konstellationen verteilen:

- Objekte innerhalb desselben Prozesses
- Objekte in verschiedenen Prozessen auf demselben Rechnerknoten
- Objekte in verschiedenen Prozessen auf verschiedenen Rechnerknoten mit ggf. verschiedenen Betriebssystemen

Durch die *Interface Definition Language* (IDL) ist es möglich, die Schnittstellen für Server und Clients unabhängig von einer bestimmten Plattform zu definieren. Die üblichen Programmiersprachen für CORBA-Anwendungen sind heutzutage Java, C++ und Cobol; C, Smalltalk und Ada trifft man immer weniger an.

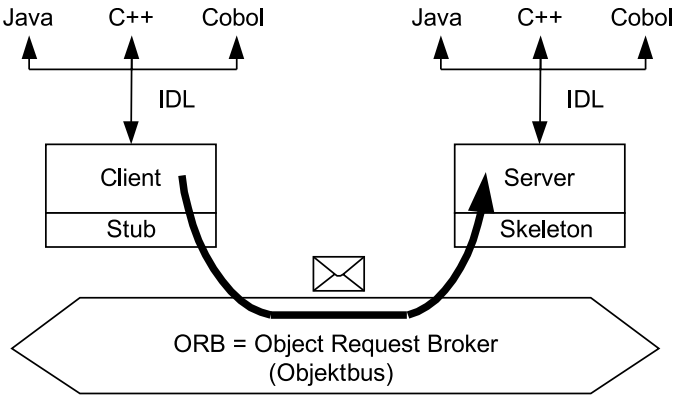

Abbildung 4.1: ORB und Heterogenität

Durch den IDL-Compiler werden Stubs für die Client-Seite und Skeletons für die Serverseite generiert. Ein Client-Stub ist ein Proxy, der die entfernten Methodenaufrufe an den Server weiterreicht, die Methodenargumente verpackt und die Resultatsobjekte entpackt. Erst ab CORBA 2.3 können auch ganze Objekte verschickt werden.

Ein Server-Skeleton enthält die technische Implementierung, um die Methodenaufrufe eines Clients entgegenzunehmen, die Methodenargumente auszupacken, die Methodenaufrufe anzustoßen und abschließend die Resultatsobjekte zu verpacken und zurückzuschicken.

4.1.2 IDL

Durch die IDL, die nicht mit der IDL von DCE-RPC verwechselt werden darf, können die Schnittstellen von Server und Client von den dazugehörigen Implementierungen getrennt werden. Eine IDL-Definition für den beispielhaften Pizza-Service sieht prinzipiell so aus:

```
module PizzaService {
  interface BestellungsDienst {
    void neueBestellung(in long kundennr, out long bestellNr);
    void weitereBestellPosition(in long bestellnr,
                                in long pizzanr,
                                in long anzahl);
  };
  interface AuslieferungsDienst {
     long auslieferung(in long kundennr);
  };
};
```

Module werden durch das Schlüsselwort `module` markiert und ermöglichen eine hierarchische Gruppierung wie die Packages bei Java oder die Namensräume (Namespace) bei C++. Letzten Endes besteht ein Modul dann aus den Schnittstellen, die durch das Schlüsselwort `interface` gekennzeichnet sind und die Signaturen der aufrufbaren Methoden enthalten.

Eine IDL-Schnittstelle kann man mit einer Java-Schnittstelle vergleichen, bei der nur die Methodensignaturen spezifiziert sind – aber nicht deren Implementierung und nicht die benötigten Attribute. Ebenso gibt es wie bei Java Vererbung zwischen den Schnittstellen. Polymorphie bei den Methoden ist dagegen nicht möglich, d.h., die Namen der Methoden müssen paarweise verschieden sein, damit auch prozedurale Sprachen, wie z.B. Cobol und C, unterstützt werden können.

Bei jedem Methodenparameter wird wie in Ada ein Schlüsselwort hinzugefügt (`in`, `out` oder `inout`), um zu charakterisieren, ob es sich um einen Eingabeparameter, einen Ausgabeparameter oder beides handelt. Wenn Get- und Set-Methoden benötigt werden, so gibt es durch das irreführende Schlüsselwort `attribute` eine Kurzform als IDL-Definition. Der angegebene Typ legt nur den Rückgabetyp für die Get-Methode und den Argumenttyp für die Set-Methode fest; es wird also kein Attribut wie bei einer Java- oder C++-Klasse festgelegt. Soll die Schnittstelle nur eine Get-, aber keine Set-Methode enthalten, so kann zusätzlich das Schlüsselwort `readonly` verwendet werden, wie das im nachfolgenden Codefragment dargestellt ist:

```
interface Bestellung {
  readonly attribute long id;    // nur Get-Methode
  attribute Datum lieferDatum;   // Datum sei ein IDL-Interface
  void neueBestellPosition(in long pizzaId, in long pizzaAnz);
};
```

Durch die IDL kann man auch eine asynchrone Kommunikation zwischen Client und Server definieren, indem das Schlüsselwort `oneway` vor den Rückgabetyp einer Methode gestellt wird. Wird später zur Laufzeit der Aufruf einer solchen Methode vom Client abgesetzt, so ist er nicht blockiert, bis diese Methode durch den Server abgearbeitet ist.

Ein typischer Fehler bei den ersten CORBA-Anwendungen bestand darin, aus jeder C++ Klasse eine IDL-Schnittstelle zu machen. Dadurch wurde ein hoher Verwaltungsaufwand bei der Kommunikation zwischen zwei Objekten eingeschleust, auch dann, wenn die beiden Objekte lokal beim Server waren. Andererseits muss natürlich für jedes Objekt, das in einer Methode einer IDL-Schnittstelle benötigt wird, eine eigene IDL-Schnittstelle verfügbar sein. Das ist z.B. bei der obigen Schnittstelle `Bestellung` in der Methode `lieferDatum` beim Typ `Datum` der Fall.

Man kann das in gewisser Weise mit Datenbanken vergleichen: Datenbanken sind ein absolut notwendiges Hilfsmittel, um große Anwendungen zu erstellen. Daraus folgt aber nicht, dass jede einzelne Anweisung auf der Datenbank operiert. Genauso ist es auch mit CORBA: Es wird eine gute Infrastruktur für verteilte Anwendungen bereitgestellt. Das bedeutet aber nicht, dass so programmiert werden muss, als wäre jedes Objekt auf einem anderen CORBA-Server.

Zusammenfassend lässt sich zur CORBA-IDL folgendes festhalten:

- Server und Client werden entkoppelt und haben nur noch eine gemeinsame Schnittstelle.
- Beim Entwurf einer Client-/Server-Schnittstelle sollte die Anzahl von entfernten Aufrufen gering gehalten werden.
- Durch Module kann eine Anwendung partitioniert werden. Dabei sollte man darauf achten, dass es nur wenige Querbeziehungen zwischen verschiedenen Modulen gibt.

4.1.3 Bestandteile für die Kommunikation

Die IDL spezifiziert nur die Schnittstelle zwischen Client und Server; die Methoden der Schnittstelle werden in der IDL nicht implementiert. Der IDL-Compiler generiert den Stub-Code für die Clients und den Skeleton-Code für die Serverseite, damit die Netzwerkkommunikation transparent abgewickelt werden kann. Auf der Serverseite müssen dann noch die IDL-Methoden implementiert werden, wie das in Abbildung 4.2 dargestellt ist.

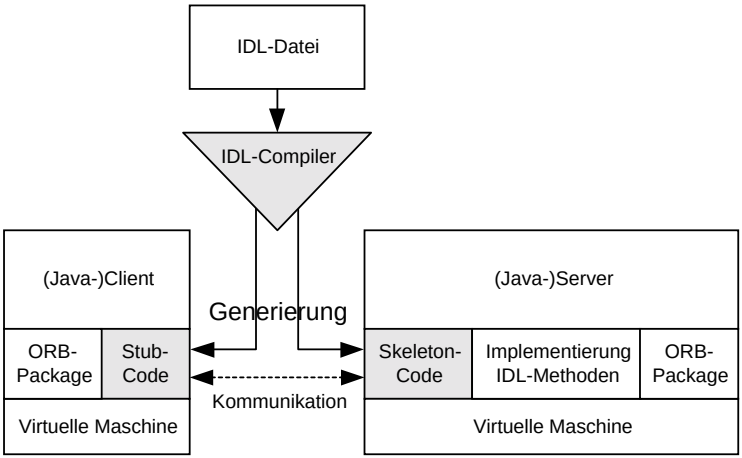

Abbildung 4.2: Prinzipieller Ablauf des IDL-Compilers

Zum Verständnis der Client-/Server-Kommunikation werden noch CORBA-Begriffe eingeführt, um die verfeinerte Architektur aus Abbildung 4.3 zu erhalten:

- Ein *CORBA-Objekt* ist eine virtuelle Einheit auf Serverseite und nimmt Methodenaufrufe der Clients entgegen. Deshalb hat ein CORBA-Objekt eine eindeutige Objektreferenz, mit der es adressiert und lokalisiert werden kann.
- Ein *Servant* implementiert ein CORBA-Objekt. Für prozedurale Sprachen wie C oder Cobol bedeutet das, dass ein Servant nichts weiter ist als eine Menge von Funktionen. Mit diesen Funktionen können die Serverdaten – ein `struct` in C oder ein `RECORD` in Cobol – gelesen oder verändert werden. Die eigentliche Ar-

beit wird also durch einen Servant verrichtet, während der aufrufende Client glaubt, dass er mit einem (virtuellen) CORBA-Objekt interagiert.

- Ein *Objektadapter* (OA) leitet eingehende Client-Aufträge an den richtigen Servant weiter, der das adressierte CORBA-Objekt implementiert, auf dem eine Methode aufgerufen werden soll. Der OA muss also die Objektreferenzen richtig auflösen können, weshalb er mit den Kommunikationseinheiten des ORBs zusammenarbeiten muss. Dadurch, dass ein OA die eingehenden Client-Aufträge an die betroffenen Servants verteilt, kann ein OA innerhalb eines CORBA-Produkts durchaus zum Flaschenhals werden. Dabei wird übrigens das Adaptermuster (Adapter Pattern) umgesetzt, weil der ORB durch verschiedene OA an verschiedene Servant-Implementierungen angepasst werden kann.
- Ein *Skeleton* ist das Programmstück, das einen Servant mit einem OA verbindet und dabei die Transformation der Methodenargumente für den Servant durchführt. Bei Java und C++ ist ein Skeleton i.A. eine Basisklasse, von der die Servant-Klasse abgeleitet ist. Bei C hingegen ist ein Skeleton nichts weiter als eine Menge von Pointern auf Servant-spezifische C-Funktionen.

Das Zusammenspiel von Client und Server über Client-Stubs, Objektdapter, Server-Skeletons und Servants ist in Abbildung 4.3 dargestellt.

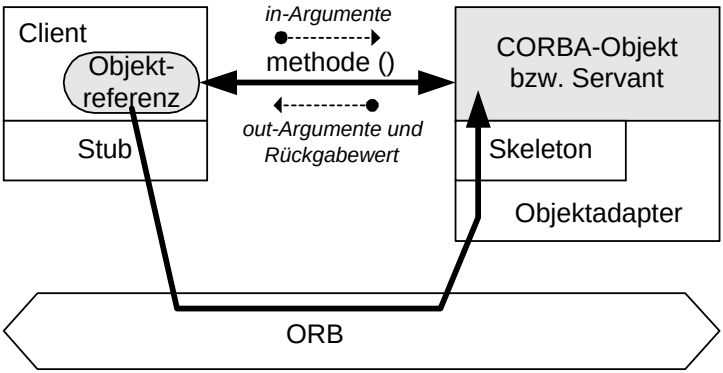

Abbildung 4.3: Kommunikationskomponenten

4.1.4 Objektadapter: BOA und POA

Zur Umsetzung eines Objektadapters gibt es seit der Version CORBA 1.0 den *Basic Object Adapter (BOA)*. Allerdings war die Spezifikation lückenhaft, so dass jedes CORBA-Produkt seine eigene Interpretation der Spezifikation vornehmen musste und deshalb seine eigene, proprietäre BOA-Implementierung hatte. Infolgedessen war es sehr aufwendig, eine Anwendung von einem CORBA-Produkt auf ein anderes zu portieren. Die wesentlichen Schwachstellen waren:

- *Keine portable Verknüpfung zwischen Skeletons und Servants.* Es ist weder beschrieben, wie die Skeletons aussehen müssen noch wie die Servants mit den Skeletons assoziiert werden. Es sind zwar die Signaturen und die Rümpfe der Servant-

Methoden definiert, aber nicht die Basisklassen, von denen die Servants abgeleitet sind.

- *Undefinierte Registrierung der Servants*. Die Implementierungen der Servants müssen auf Serverseite registriert werden, damit sie vom Objektadapter gefunden werden können. Das API zur Registrierung ist jedoch nicht spezifiziert, so dass es CORBA-Produkte gibt, bei denen die Registrierung explizit durch einen bestimmten Methodenaufruf erfolgen muss, und andere Produkte, bei denen die Registrierung implizit durch den Konstruktor des Servants erfolgt.
- *Multithreading ist nicht berücksichtigt*. Um einen hohen Durchsatz bei vielen konkurrierenden Clients zu gewährleisten, kann man i.A. nicht je Kommunikationsbeziehung einen eigenständigen Prozess auf Serverseite bereitstellen. Deshalb ist ein multithreaded CORBA-Server notwendig. In der BOA-Spezifikation wurde Multithreading jedoch nicht berücksichtigt und vollständig den Herstellern überlassen.
- *Bereitschaft für Client-Aufträge ist unpräzise*. Wenn ein Server hochgefahren wird, sind zunächst diverse Vorkehrungen wie z.B. die Erzeugung von Servants notwendig. Ab einem bestimmten Zeitpunkt ist ein Server bereit, Aufträge der Clients, d.h. Methodenaufrufe, durchzuführen. Dazu gibt es beim BOA-Ansatz die beiden Methoden `impl_is_ready` und `obj_is_ready`, die jeweils unter spezifischen Bedingungen aufgerufen werden können. An dieser Stelle ist die BOA-Spezifikation jedoch sehr vage und unpräzise, so dass die verschiedenen Hersteller zwangsläufig verschiedene Implementierungen produziert haben.

Auf Grund der Unzulänglichkeiten des BOA-Ansatzes haben sich die OMG-Mitglieder 1997 darauf verständigt, die Spezifikationslücken nicht zu schließen und den BOA-Ansatz nicht zu reparieren. Stattdessen wurde der *Portable Object Adapter (POA)* als neuer Standard bereitgestellt und ist heute in Produkten, wie z.B. Orbix2000, enthalten. Dadurch, dass der POA völlig neu spezifiziert ist, konnten die Hersteller ihren jeweils proprietären BOA weiterhin für ihre Kunden aus Wartungsgründen beibehalten.

4.1.5 Codegenerierung

In Abbildung 4.4 ist der prinzipielle Generierungsvorgang für den POA-Ansatz und die Programmiersprache Java dargestellt. Ausgehend von einer IDL-Schnittstelle `Bestellung` werden verschiedene Java-Klassen und -Schnittstellen generiert:

- Die Java-Schnittstelle `BestellungOperations` enthält die Methoden, so wie sie in der IDL-Definition spezifiziert wurden.
- Die Java-Schnittstelle `Bestellung` kann innerhalb von Methodensignaturen als Argumenttyp verwendet werden. Sie erbt die Methoden von `BestellungOperations` und die CORBA-Funktionalität von `org.omg.CORBA.Object`.
- Die Java-Klasse `BestellungPOA` stellt die Funktionalität des Objektadapters bereit.

In der Klasse `BestellungImpl` realisiert der Anwendungsprogrammierer die Methodenrümpfe zu den Methoden, die in der IDL-Schnittstelle deklariert wurden. Im vorliegenden Beispiel wird unterstellt, dass die Implementierung mit Java erfolgt; es wäre aber genauso möglich, C++, C, Ada oder Cobol zu verwenden, falls das ORB-Produkt die jeweilige Programmiersprache unterstützt.

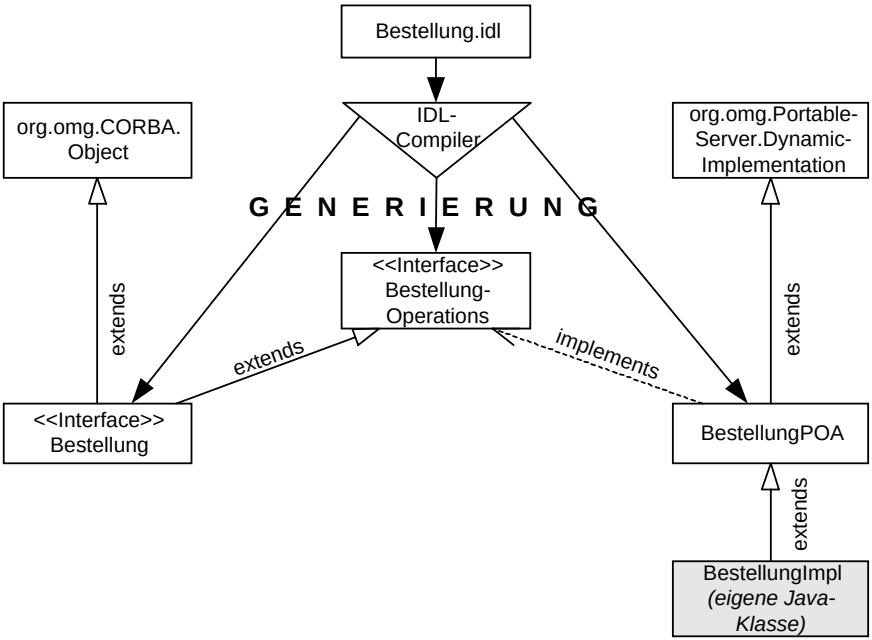

Abbildung 4.4: Codegenerierung des IDL-Compilers

Der Implementierungscode zu einer gegebenen IDL-Schnittstelle `Bestellung` kann dann folgendermaßen aussehen, wobei aus Gründen der Übersichtlichkeit auf die Fehlerbehandlung verzichtet wurde.

```
public class BestellungImpl extends BestellungPOA {
  protected int    f_id;  // CORBA "long" entspricht Java "int"
  protected Datum  f_lieferDatum;
  protected Vector f_bestellPositionen;
  public BestellungImpl() {
    f_id = ...;               // Zuweisung einer eindeutigen ID
    f_bestellPositionen = new Vector();
  }
  public int id() {           // nur eine Get-Methode
    return f_id;
  }
  public Datum lieferDatum() {   // Get- und Set-Methode
    return f_lieferDatum;
  }
```

```
  public void lieferDatum(Datum d) {
    f_lieferDatum = d;
  }
  public void neueBestellPosition(int pizzaId, int pizzaAnz) {
    BestellPosition p = new BestellPosition(pizzaId, pizzaAnz);
    f_bestellPositionen.addElement(p);
  }
};
```

4.1.6 IIOP und IOR

Bei der Spezifikation von CORBA 2.0 wurde das *General Inter-ORB Protocol (GIOP)* eingeführt, damit Interoperabilität zwischen verschiedenen ORBs von verschiedenen Herstellern gewährleistet ist. Dazu gehört auch die Beschreibung, wie IDL-Typen in ein Nachrichtenformat abgebildet werden, wenn ein Client eine Methode auf einem Server aufruft.

Das *Internet Inter-ORB Protocol (IIOP)* konkretisiert eine Implementierung des GIOP für TCP/IP als Transportprotokoll. Durch IIOP können dann verschiedene ORBs über das Internet kommunizieren, d.h., das Internet ist im Prinzip ein »Backbone-ORB«. Bei der Kommunikation benötigen die verschiedenen ORBs keine Kenntnisse über die interne Implementierung des jeweils anderen ORBs, wie z.B. Protokolle und Datenstrukturen für die ORB-spezifische Interprozesskommunikation.

Wie bereits erwähnt, braucht ein Client entfernte Objektreferenzen, wenn er auf einem CORBA-Objekt auf Serverseite eine Methode aufrufen möchte. Im Zusammenhang mit GIOP wurde bei CORBA 2.0 auch die *Interoperable Object Reference (IOR)* spezifiziert. Sobald eine Objektreferenz durch einen ORB zu übertragen ist, muss er eine IOR übertragen. Eine IOR besteht aus einem Identifikator für den Typ des referenzierten Objekts und einer Datensequenz, welche die ORB-Domäne beschreibt, zu der IOR gehört. Dazu gehört u.a. die IP-Adresse des Rechnerknotens und die Identifikation des Prozesses, in dem das CORBA-Objekt erzeugt wurde.

Zur Handhabung von IORs gibt es die beiden Methoden `object_to_string` und `string_to_object` der Java-Schnittstelle `org.omg.CORBA.ORB`. Diese Methoden werden auch im Zusammenhang mit dem Verzeichnisdienst (Naming Service) in Kapitel 5 benötigt. Im folgenden Codefragment wird eine Stringrepräsentation für eine IOR erzeugt, wenn eine Java-Referenz auf ein CORBA-Objekt zur Java-Schnittstelle `Bestellung` vorliegt, die vom IDL-Compiler generiert wurde.

```
import org.omg.CORBA.ORB;
public static void main(String args[]) {
  ORB orb = ORB.init(args, null);
            // Konfig.-Parameter nur durch die Kommandozeile
            // keine Systemproperties java.util.Properties
  Bestellung bestellung = ...;
  String bestellungStr = orb.object_to_string(bestellung);
  ...
};
```

Beim umgekehrten Weg, wenn aus einer Zeichenkette eine Objektreferenz konstruiert werden soll, liefert `string_to_object` nur eine Referenz auf `org.omg.CORBA.Object`. Deshalb ist anschließend eine Konvertierung in eine von `org.omg.CORBA.Object` abgeleitete Klasse oder Schnittstelle notwendig (Downcast); in unserem Beispiel ist das die Java-Schnittstelle `Bestellung` (siehe Abbildung 4.4). Für diese Konvertierung wird der vom IDL-Compiler generierte Stub-Code verwendet. Das nachfolgende Codefragment illustriert diesen Vorgang mit der ebenfalls generierten Hilfsklasse `BestellungHelper`, welche die Methode `narrow` für die Konvertierung bereitstellt.

```
import org.omg.CORBA.ORB;
import org.omg.CORBA.Object;
public static void main(String args[]) {
  ORB orb = ORB.init(args, null
  String bestellungStr = ...;       // siehe oben
  Object obj = orb.string_to_object(bestellungStr);
  Bestellung best = BestellungHelper.narrow(obj)
  // ggf. Exception CORBA.BAD_PARAM abfangen
  // ...
};
```

4.1.7 Systemdienste, Domänenobjekte und CORBAfacilities

Die bisherigen Aspekte, die bislang von CORBA vorgestellt wurden, könnte man unter dem Schlagwort »objektorientierter RPC« zusammenfassen. Das ist zwar ein wesentlicher Bestandteil von CORBA; die gesamte Palette an Funktionalität ist damit aber keineswegs abgedeckt. Erst die *Object Management Architecture (OMA)*, die in Abbildung 4.5 dargestellt ist, zeigt die gesamte Mächtigkeit von CORBA.

Systemdienste (*CORBAservices*, *COS*) erleichtern dadurch die Anwendungsprogrammierung, dass Basisfunktionalität in Bausteinen, wie z.B. Java-Packages, geliefert wird, die in sich abgeschlossen sind. Solche Systemdienste sind z.B. der Verzeichnisdienst (Naming Service), der Nachrichtendienst (Notification Service) und der Transaktionsdienst (Objekt Transaction Service), die in Kapitel 5 vorgestellt werden.

Die horizontalen *CORBAfacilities* dagegen setzen Spezifikationen auf einer höheren Ebene um, wie z.B. Systemmanagement oder verteilte Dokumente. Die vertikalen *Domänenobjekte* (Domain Objects) adressieren Anwendungsgebiete wie z.B.:

- Telekommunikation
- Finanzdienstleistungen
- E-Commerce
- Fertigung und Produktion (Manufacturing)
- Transportwesen
- Medizin
- Biotechnologie

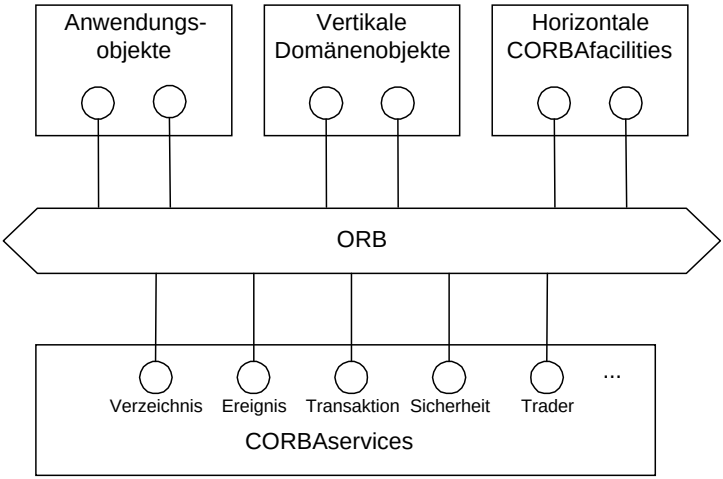

Abbildung 4.5: Object Management Architecture (OMA)

Weitere Aspekte von CORBA konnten in diesem Abschnitt nicht angesprochen werden. Dazu gehören u.a. die dynamische Zuordnung von Methodenaufrufen und das *CORBA Component Model (CCM)*. Auf das CCM wird erst in Kapitel 11 eingegangen, wenn Enterprise JavaBeans (EJB) im Detail vorgestellt werden.

4.2 COM+

4.2.1 Historische Entwicklung und Einordnung

Object Linking and Embedding (OLE) wurde ursprünglich 1990 von Microsoft für das Betriebssystem Windows 3.1 entwickelt. Dadurch war eine Plattform geschaffen, um Dokumente zu komponieren. Die Grundlage bildete eine Bibliothek von vordefinierten Schnittstellen mit einigen Default-Implementierungen.

Die Weiterentwicklung zu OLE2 und heute ActiveX basiert auf dem *Component Object Model (COM)*. COM wurde 1993 entwickelt und stellt eine Infrastruktur bereit, damit COM-Komponenten innerhalb eines Prozesses oder zwischen verschiedenen Prozessen auf demselben Rechnerknoten interagieren können. Die Interaktion der COM-Komponenten erfolgt dabei auf einer binären Ebene, so dass verschiedene COM-Komponenten auch in verschiedenen Programmiersprachen implementiert werden können. COM bildet heute die Grundlage für die Microsoft-Technologien OLE, ActiveX und auch DCOM.

Ein COM-Objekt kann mehrere Schnittstellen bereitstellen, und jede Schnittstelle enthält eine Menge von funktional zusammengehörigen Methoden. Ein COM-Client interagiert mit einem COM-Objekt, indem eine Referenz auf eine der Schnittstellen des COM-Objekts benutzt wird, um so Methoden aufzurufen. Dabei muss jede Schnittstelle einem bestimmten Speicherlayout folgen, was mit der virtuellen Funktionstabelle (Virtual Function Table, vtbl) von C++ verglichen werden kann. Damit ist eine Spezifikation auf binärer Ebene möglich, so dass Komponen-

ten in verschiedenen Programmiersprachen implementiert werden können. Das sind heute im Wesentlichen die Sprachen Java, C++ und Visual Basic.

Distributed COM (DCOM) ist eine Erweiterung von COM, so dass auch Objekte von verschiedenen Rechnerknoten miteinander interagieren können; wobei das Kommunikationsprotokoll auf DCE-RPC aufbaut. DCOM wurde erstmals 1996 zusammen mit Windows NT 4.0 ausgeliefert. Selbst wenn es von der Software AG heute eine Unix-Portierung unter dem Namen EntireX gibt, so ist DCOM immer noch auf die Betriebssystemfamilie Windows fokussiert; für Windows 95 und Windows 98 kann man sich DCOM von der Webseite von Microsoft herunterladen.

In Abbildung 4.6 ist dargestellt, wie DCOM in *die Distributed interNet Architecture (DNA)* von Microsoft einzuordnen ist: Im Zentrum steht ein NT-Server. Auf dem NT-Server kann der *Internet Information Server (IIS)* als Webserver laufen, und Webbrowser können über HTTP angebunden werden; mit dem SNA-Server kann eine Verbindung zu Großrechnern hergestellt werden.

Der *Microsoft Transaction Server (MTS)* ist seit 1996 verfügbar und ermöglicht die Kommunikation über DCOM mit Windows- oder Unix-Clients. Eine wesentliche Funktionalität des MTS ist die Speicherung von Objekten und die Koordination verteilter Transaktionen. Dabei werden auch kritische Ressourcen verwaltet, so dass der MTS eine hohe Skalierbarkeit verspricht. Sicherheits- und Managementfunktionen werden durch die Basismechanismen von Windows NT bereitgestellt. Der MTS kann deshalb als Laufzeitumgebung für die Komponenten in der mittleren Stufe einer verteilten Architektur angesehen werden.

COM+ wird erstmals mit Windows 2000 ausgeliefert und ist die Integration von DCOM, MTS und neuen COM+ Services. Im Vergleich zu CORBA haben bei DCOM gerade die Dienste gefehlt, was auch die Implementierung mühsam gemacht hat. COM+ ist damit die Synthese von:

- Komponentenmodell durch den Anteil von COM
- Technologie für verteilte Anwendungen durch den Anteil von DCOM
- Laufzeitumgebung durch den Anteil von MTS und COM+ Services

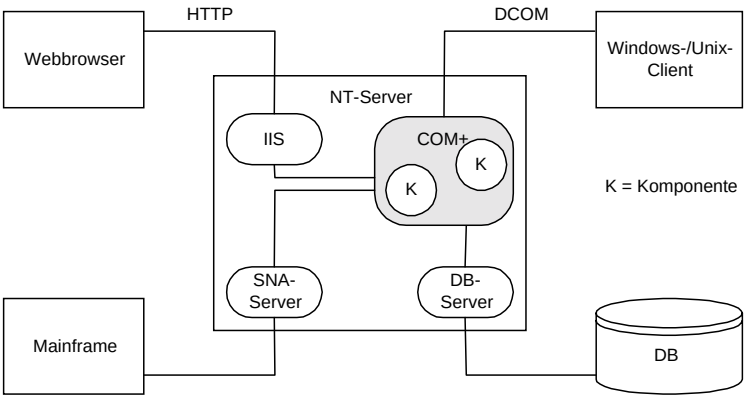

Abbildung 4.6: Distributed interNet Architecture (DNA)

Diese eher abstrakte Sichtweise wird bei Microsoft durch konkrete Produkte und Technologien hinterlegt. Deshalb finden die Designer und Entwickler aus technologischer Sicht folgende Systemsoftware bei mehrstufigen Architekturen in einer Windows-Umgebung vor:

Client	Applikationsserver	Backend
Visual Basic	IIS	SQL Server
XML	COM+	
HTML		
Java		

4.2.2 MIDL im Vergleich zur CORBA-IDL

Auch wenn die Schnittstellenbeschreibungen für COM+ üblicherweise aus grafischen Werkzeugen wie z.B. Visual Basic, J++ oder Visual C++ heraus generiert werden, so soll dennoch in diesem Abschnitt die Microsoft Interface Definition Language (MIDL) der IDL von CORBA gegenübergestellt werden. Das ermöglicht einen Eindruck auf der Ebene der ASCII-Dateien, da die zuvor genannten Werkzeuge nur in proprietären Windows-Umgebungen eingesetzt werden können. Allerdings muss man sich bewusst sein, dass der Vergleich hinkt, weil man MIDL-Dateien i.A. nicht mit einem gewöhnlichen Editor bearbeitet.

Zunächst werden zwei Schnittstellen `IBestellung1` und `IBestellung2` in der MIDL beschrieben. Beide Schnittstellen werden abschließend in der COM+-Klasse `CBestellung` zusammengefasst. Dadurch sieht man, wie mehrere Schnittstellen für ein späteres COM+ Objekt definiert werden können. Es ist Konvention, dass Schnittstellen mit einem »I« beginnen und Klassen mit einem »C«.

Die erste Schnittstelle stellt drei Methoden bereit, um ein Lieferdatum zu setzen und zu lesen, sowie um eine neue Bestellposition zur Bestellung hinzuzufügen. Die zweite Schnittstelle ermöglicht es, die Bestellposition zu einer bestimmten Pizza-(Sorte) zu stornieren. Bei dieser Definition fällt auf, dass sehr viel Programmiercode anfällt bzw. generiert wird.

Weiterhin wird immer wieder dieselbe Nummer verwendet, nämlich die *Globally Unique ID (GUID)*, die an die Funktion `uuid` übergeben wird. Die GUID ist ein eindeutiger Bezeichner für eine COM+-Komponente und wird durch einen Hash-Algorithmus aus einem der grafischen Werkzeuge heraus generiert. Dieser Algorithmus entstammt DCE und gewährleistet, dass dieselbe GUID zu keiner Zeit, auf keinem Rechner und in keinem Prozess ein zweites Mal generiert werden kann.

Die beide Schnittstellen `IBestellung1` und `IBestellung2`, die in der COM+-Komponente `CBestellung` angeboten werden, sind beide von der Basisschnittstelle `IUnknown` abgeleitet. `IUnknown` stellt die Methode `QueryInterface` bereit, um zwischen verschiedenen Schnittstellen umschalten zu können. Das war insbesondere zu Zeiten von DCOM wichtig, weil eine Schnittstelle die kleinste Einheit für den Zugriffsschutz (Security) war. Deshalb musste man immer wieder Schnittstellen künstlich in kleinere Schnittstellen aufteilen. Bei COM+ sind Schutzmaßnahmen

auch auf Ebene der Methoden möglich, was mehrfache Vererbung häufig obsolet werden lässt. Die beiden anderen Methoden von `IUnknown` sind `AddRef` und `Release`, um zu verwalten, welche (Proxy-)Objekte auf COM+-Objekte verweisen. Zunächst ist der rein deklarative Code für die COM+-Komponente `CBestellung` angegeben. Danach wird auf die Implementierung der Methoden eingegangen.

```
// uuid und Definition von IBestellung1
[ object,
  uuid(3CFDB283-CCC5-11D0-BA0B-00A0C90DF8BC),
  helperstring("IBestellung1 Interface"),
  pointer_default(unique)
]
interface IBestellung1 : IUnknown {
  import "unknwn.idl";
  HResult getLieferDatum([out] LONG *resultat);
  HResult setLieferDatum([in] LONG neuesDatum);
  HResult neueBestellPosition([in] SHORT pizzaId,
                              [in] SHORT pizzaAnzahl);
};
// uuid und Definition von IBestellung2
[ object,
  uuid(3CFDB283-CCC5-11D0-BA0B-00A0C90DF8BC),
  helperstring("IBestellung2 Interface"),
  pointer_default(unique)
]
interface IBestellung2 : IUnknown {
  import "unknwn.idl";
  HResult reset([in] SHORT pizzaId);
};
// uuid und Definition der "Type Library"
[ uuid(3CFDB283-CCC5-11D0-BA0B-00A0C90DF8BC),
  version(1.0),
  helpstring("Pizza Service 1.0 Type Library")
]
library PizzaLib
[ importlib("stdole32.tlb");
  // uuid und Definition der Klasse
  [ uuid(3CFDB283-CCC5-11D0-BA0B-00A0C90DF8BC),
    helpstring("Bestellung Class")
  ]
  // mehrere Schnittstellen
  coclass CBestellung {
    [default] interface IBestellung1;
    interface IBestellung2;
  };
]
```

Im Vergleich dazu sieht der Code für die CORBA-IDL viel einfacher aus:

```
interface Bestellung1 {
  attribute long lieferDatum;
  void neueBestellPosition(in short pizzaId,
                           in short pizzaAnzahl);
};
interface Bestellung2 {
  void reset(in short pizzaId);
};
// Multiple Vererbung der Schnittstellen
interface Bestellung : Bestellung1, Bestellung2 {
};
```

Wie bereits erwähnt, ist der Vergleich nicht ganz fair, da die MIDL aus grafischen Werkzeugen heraus generiert wird, die aber ausschließlich in Windows-Umgebungen angewandt werden können. Im nächsten Schritt werden die Methoden der COM+-Schnittstellen und -Komponenten implementiert.

Der Implementierungsteil ist weniger intuitiv, schwerer verständlich und damit fehleranfälliger als bei CORBA und Enterprise JavaBeans, was durch den historischen Ursprung COM bedingt ist. Deshalb wird auch auf eine Gegenüberstellung mit CORBA-Code verzichtet. Gerade die Implementierung der Methoden für `IUnknown` ist alles andere als intuitiv und soll hier nicht weiter vertieft werden. An der Stelle hat sich aber der Komfort bei COM+ gegenüber DCOM dadurch verbessert, dass bei COM+ die Implementierung der Methoden `QueryInterface`, `AddRef` und `Release` generiert wird.

```
#include "bestellung.h"   // generierte Header-Datei für MIDL
class CClassFactory : public IClassFactory {
  public:
    // IUnknown
    STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
    STDMETHODIMP_(ULONG) AddRef(void)    { return 1; }
    STDMETHODIMP_(ULONG) Release(void)   { return 1; }

    // IClassFactory
    STDMETHODIMP CreateInstance(LPUNKNOWN punkOuter,
                                REFIID riid, void** ppv);
    STDMETHODIMP LockServer(BOOL fLock)  { return E_FAIL; }
};
class CBestellung : public IBestellung1, public IBestellung2 {
  public:
    // IUnknown
    STDMETHODIMP QueryInterface(REFIID riid, void** ppv);
    STDMETHODIMP_(ULONG) AddRef(void)
                    { return InterlockedIncrement(&m_cRef); }
```

```
    STDMETHODIMP_(ULONG) Release(void)
                         { if (InterlockedDecrement(&m_cRef) == 0) {
                             delete this;
                             return 0;
                           }
                           return 1;
                         }
    // IBestellung1
    STDMETHODIMP getLieferDatum(OUT LONG *resultat);
    STDMETHODIMP setLieferDatum(IN LONG neuesDatum);
    STDMETHODIMP neueBestellPosition(IN SHORT pizzaId,
                                     IN SHORT pizzaAnzahl);
    // IBestellung2
    STDMETHODIMP reset(IN SHORT pizzaId);
    // CBestellung
    CBestellung(LONG lieferDatum);
    ~CBestellung();

  private:
    long m_lieferDatum;
    ...
};
```

4.2.3 Verbindungen zwischen CORBA und COM

Tatsache ist, dass CORBA und COM+ bzw. DCOM bzw. COM weit verbreitet sind: Das OMG-Konsortium hat über 1000 renommierte Firmen als Mitglieder, und Windows ist ein weit verbreitetes Betriebssystem. Deshalb ist es eine natürliche Folge, dass die beiden (Kommunikations-)Welten miteinander verbunden und die notwendigen Abbildungen bereitgestellt werden müssen. Eines der ersten Produkte war ORBIX COMet von IONA Technologies, um CORBA und DCOM in heterogenen Umgebungen zu verbinden.

Eine Verbindung (Bridge) zwischen CORBA und COM muss Abbildungen für den Zugriff in beiden Richtungen abdecken. Wenn ein Client eine Referenz auf eine COM-Schnittstelle hat, dann muss es möglich sein, dass das COM-Objekt auf Serverseite auf ein CORBA-Objekt zugreift. Dabei müssen die elementaren Datentypen von COM auf die elementaren Datentypen von CORBA abgebildet werden, und die Referenz auf die COM-Schnittstelle wird in eine Objektreferenz von CORBA transformiert. Das ist in Abbildung 4.7 dargestellt.

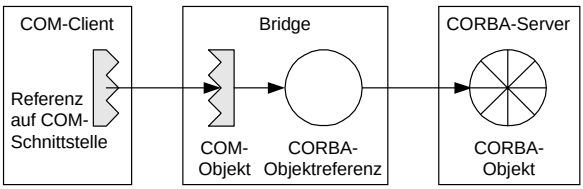

Abbildung 4.7: Zugriff eines COM-Clients auf ein CORBA-Objekt

Auch wenn dieser Abbildungsmechanismus einfach aussehen mag, so ist dennoch viel Fleißarbeit bei der Implementierung notwendig. Beispielsweise müssen Aggregationen in der COM-Welt auf multiple Vererbungen in der CORBA-Welt abgebildet werden, oder Get- und Set-Methoden der MIDL müssen zu `attribute`-Klauseln in der CORBA-IDL transformiert werden.

Der umgekehrte Fall tritt ein, wenn ein CORBA-Client auf ein COM-Objekt zugreifen will. Die Abbildung der Zugriffe ist ziemlich analog zum obigen Fall. Das einzige Problem kann sein, dass das benötigte COM-Objekt keine Schnittstelle bereitstellt und direkt in irgendeiner Programmiersprache implementiert ist. Dann muss der CORBA-Programmierer ggf. manuelle Nacharbeiten für die COM-Programmierung vornehmen.

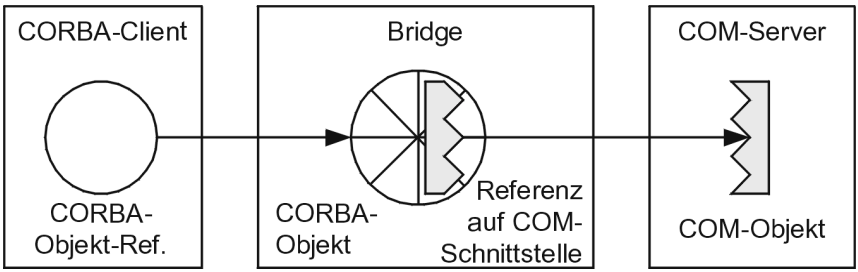

Abbildung 4.8: Zugriff eines CORBA-Clients auf ein COM-Objekt

4.2.4 Das Forschungsprojekt Millennium

Microsoft ist sich bewusst, dass zwar COM+ eine deutliche Verbesserung und mehr Programmierkomfort als DCOM bringt, weil der MTS und die COM+ Services integriert sind. Dennoch gibt es weitere Verbesserungspotenziale. Deshalb wurde bei Microsoft das Forschungsprojekt Millennium durchgeführt (siehe dazu *http://research.microsoft.com/os/millennium*). Das Hauptziel war, eine höhere Abstraktionsebene für Programmierer und Anwender bereitstellen zu können, wozu mehrere Prototypen entwickelt wurden:

- Der Prototyp Continuum hat eine einzige *abstrakte Maschine* bereitgestellt, die mehrere verteilte Rechner abgebildet hat. Durch diese abstrakte Maschine ist es den Programmierern nicht bewusst, dass sie eigentlich eine verteilte Anwendung zu erstellen haben.
- Mit dem Prototyp Coign können verteilte Anwendungen zur Laufzeit *automatisch neu partitioniert* werden, um die Kommunikationskosten zu reduzieren. Diese Neupartitionierung erfolgt auf der Basis von Binärdateien und nicht von Quellcode. Das ermöglicht eine automatische Reaktion auf Veränderungen, sei es in der Netzinfrastruktur oder bei den Zugriffsmustern der Anwender.
- Im Prototyp Falcon wurde DCOM für ein *Gigabit-Netzwerk* optimiert, um die Performance bereitzustellen, die für den Overhead notwendig ist, der sich bei einem höheren Abstraktionslevel wie bei Continuum oder Coign ergibt.

4.3 RMI

Durch Remote Method Invocation (RMI) können Java-Objekte in verschiedenen virtuellen Maschinen (Virtual Maschine, VM) so miteinander kommunizieren, als wären sie in derselben VM allokiert. Dabei sehen entfernte Methodenaufrufe so aus wie lokale Methodenaufrufe. RMI ist also – im Unterschied zu CORBA – ausschließlich auf Java fokussiert und ein Mechanismus zur Kommunikation

- zwischen verschiedenen Java-Programmen und
- zwischen Java-Programmen und Applets.

4.3.1 Architektur und Aufgaben

Die wesentlichen Aufgaben, die bei verteilten Java-Anwendungen mit RMI auftreten, lassen sich folgendermaßen charakterisieren:

- *Lokalisierung entfernter Objekte*. Mit Referenzen auf entfernte Objekte kann man auf zwei Arten umgehen. Einerseits kann durch den einfachen Verzeichnisdienst `rmiregistry` über einen bekannten Namen die entfernte Referenz ermittelt werden, und andererseits können Referenzen als Methodenargumente und -resultate weitergereicht werden.
- *Transparente Kommunikation mit entfernten Objekten*. Wie bei CORBA sind dem Anwendungsprogrammierer die Details der Netzwerkkommunikation verborgen, und er braucht sich nicht darum zu kümmern. Ein Aufruf einer entfernten Methode sieht aus wie ein Aufruf einer lokalen Methode.
- *Laden von Java-Bytecode für entfernte Objekte*. Bei einem entfernten Methodenaufruf können Java-Objekte durch Serialisierung übertragen werden. Deshalb ist es notwendig, dass nicht nur die Daten, sondern auch der Bytecode der zugehörigen Klasse übertragen bzw. geladen wird.

Diese Aufgaben sind in Abbildung 4.9 dargestellt. Über RMI greift ein Client auf Objekte der Serverseite zu. Dazu kann er sich die Objektreferenzen über den Verzeichnisdienst oder direkt vom RMI-Server holen. Damit sich ein Client eine Objektreferenz vom Verzeichnisdienst holen kann, muss der Server das Objekt zuvor beim Verzeichnisdienst registriert haben, und der Client muss den mit dem Objekt assoziierten Namen kennen.

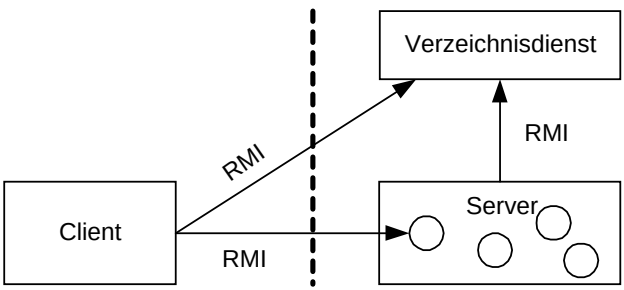

Abbildung 4.9: Kommunikationsbeziehungen bei RMI

Wenn ein Client eine entfernte Methode aufruft, dann sind eine Reihe von Aktivitäten vom Client-Stub auszuführen, die ähnlich wie bei CORBA sind:

- Die Verbindung mit der VM, die das entfernte Objekt enthält, muss hergestellt werden.
- Die Methodenargumente müssen verpackt (Marshalling) und anschließend an die entfernte VM per Serialisierung übertragen werden.
- Der Client wartet, bis der Server die Methode abgearbeitet hat und die Resultate zum Client überträgt.
- Der Rückgabewert – oder die auf Serverseite ausgelöste Ausnahme – wird vom Stub ausgepackt (Unmarshalling).
- Der ermittelte Wert wird an die ursprüngliche Aufrufstelle des Clients weitergereicht.

Als Methodenargumente wie auch als Rückgabewerte von Methoden können alle Objekte verwendet werden, die serialisierbar sind, weil Serialisierung zur Kommunikation zwischen zwei virtuellen Maschinen eingesetzt wird. Beim Übertragen des Datenstromes müssen lediglich noch die netzwerktypischen Informationen, wie z.B. die Adresse des Senders und des Empfängers, mitgegeben werden. Wenn die übertragenen Objekte empfangen worden sind, braucht die VM Informationen über die Klasse. Dazu sucht sie über den Klassennamen zuerst in ihrem lokalen Klassenkontext (Class Loading Context). Wenn die Klassendefinition nicht verfügbar ist, wird sie dynamisch über das Netzwerk nachgeladen.

Bei einem entfernten Methodenaufruf des Clients finden sich auf der Serverseite die dazu komplementären Aktivitäten im Skeleton-Code. Der Skeleton-Code wurde im JDK 1.1 durch den `rmic`-Compiler erzeugt; ab dem JDK 1.2 gibt es ein weiteres Protokoll, so dass generischer Code die Aufgaben der dedizierten Skeletons übernehmen kann.

- Die Argumente, die der Client an die entfernte Methode übergeben hat, müssen ausgepackt werden (Unmarshalling).
- Die gewünschte Methode wird mit den übertragenen Argumenten ausgeführt.
- Das Resultat wird verpackt (Marshalling) und an den aufrufenden Client übertragen.

4.3.2 Ein Beispiel

Anhand des Beispiels mit dem Pizza-Service wird die Wirkungsweise von RMI genauer vorgestellt. Die Schnittstelle `Bestellung` legt fest, welche Methoden des Servers durch den Client aufgerufen werden können. Der Einfachheit halber beschränken wir uns bei der Schnittstelle auf eine Bestellung; weitere Schnittstellen sind analog zu realisieren. Bei der Definition einer Schnittstelle für RMI muss man lediglich dafür sorgen, dass von der Basisschnittstelle `Remote` aus dem Paket `java.rmi` abgeleitet wird. Dieses Paket stellt auch die Ausnahme `RemoteException` bereit, die vom Server im Fehlerfall ausgelöst wird und deshalb bei jeder Methode

der Schnittstelle deklariert werden muss. Eine solche Java-Schnittstelle kann man dann mit dem Java-Compiler `javac` übersetzen.

Nachfolgend findet sich der Quellcode der Java-Schnittstellen und Klassen für eine lauffähige Anwendung. Dazu gehören folgende Klassen, die im Zusammenhang in Abbildung 4.10 dargestellt sind:

- `Bestellung` definiert die Schnittstelle für Client und Server.
- `BestellungImpl` implementiert die Methoden der Schnittstelle auf der Serverseite.
- `BestellungClient` implementiert das Frontend auf der Client-Seite.
- `BestellungServer` implementiert den Hochlauf des RMI-Servers, wobei die für die Clients wichtigen Objekte erzeugt werden.

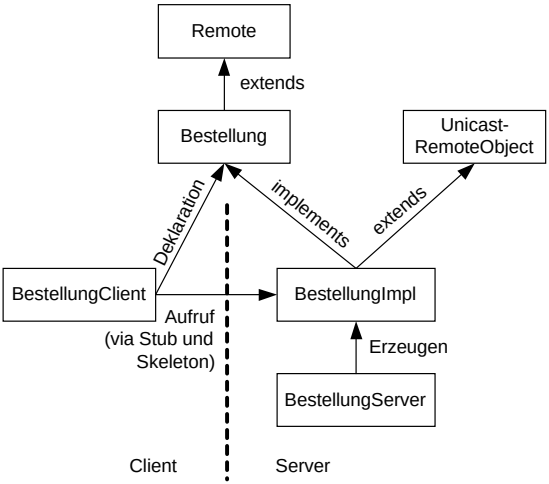

Abbildung 4.10: Zusammenspiel der Klassen bei RMI

```
import java.rmi.*;
import java.util.Date;
public interface Bestellung extends Remote {
   public void neueBestellPosition(int pizzaId, int anzahl)
                                   throws RemoteException;
   public Date getLieferDatum() throws RemoteException;
   public void setLieferDatum(Date neuesDatum)
                              throws RemoteException;
}
```

Wenn der Client eine dieser Methoden aufruft, muss sie vom Server ausgeführt werden. Diese Methoden werden durch die Klasse `BestellungImpl` implementiert, die von der Klasse `UnicastRemoteObject` aus dem Paket *java.rmi.server* abgeleitet sein muss. Die Ausnahme `RemoteException` muss beim Konstruktor und bei allen Methoden deklariert werden, welche die jeweilige Methode aus der korrespondierenden Schnittstelle `Bestellung` implementieren.

Um die Übersicht zu behalten, wurde für dieses Beispiel nur die Schnittstelle `Bestellung` verwendet. Folglich kann ein Client auch nur auf eine Bestellung zugreifen, wenn er den beim Verzeichnisdienst registrierten Namen für eine Bestellung kennt. Deshalb wird innerhalb des Konstruktors von `BestellungImpl` die Methode `rebind` der Klasse `java.rmi.Naming` aufgerufen, und das neue Objekt der Klasse `BestellungImpl` wird unter dem spezifizierten Namen registriert. Dieser Name hat das Format `//Rechnername:Port/ObjektName`, wobei der Rechnername und die Portadresse optional sind.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
public class BestellungImpl extends UnicastRemoteObject
                            implements Bestellung {
   private Vector fBestellPositionen;
   private Date fLieferDatum;
   public BestellungImpl(String name) throws RemoteException {
      super();
      try {
         Naming.rebind(name, this); // Registrierung beim Nameserver
         fBestellPositionen = new Vector();
         fLieferDatum = null;
      }
      catch (Exception e) {
         System.err.println("Ausnahme: " + e.getMessage());
         e.printStackTrace();
      }
   }
   public void neueBestellPosition(int pizzaId, int anzahl)
                                   throws RemoteException {
      // die Klasse BestellPosition sei bekannt
      BestellPosition bestellPosition =
                      new BestellPosition(pizzaId, anzahl);
      fBestellPositionen.addElement(bestellPosition);

   }
   public Date getLieferDatum() throws RemoteException {
      return fLieferDatum;
   }
   public void setLieferDatum(Date neuesDatum)
                              throws RemoteException {
      fLieferDatum = neuesDatum;
   }
}
```

Die vom Server implementierten Methoden kann nun ein Client aufrufen, wie das im nachfolgenden Programmfragment illustriert ist. Der Client kontaktiert den Verzeichnisdienst auf dem Rechner `"berlin"` unter dem Port `9000` und ermittelt so das Bestellobjekt mit dem Namen `"meine_best"`. Dann wird diesem Bestellobjekt

durch die Schnittstellenmethode `neueBestellPosition` eine neue Bestellposition hinzugefügt, die aus der Nummer der bestellten Pizza und der zugehörigen Anzahl besteht.

```
import java.rmi.*;
public class BestellungClient {
   public static void main(String args[]) {
      try {
         Bestellung bestellung = (Bestellung)
                    Naming.lookup("rmi://berlin:9000/meine_best");
         int pizzaId = Integer.parseInt(args[0]);
         int anzahl = Integer.parseInt(args[1]);
         bestellung.neueBestellPosition(pizzaId, anzahl);
      }
      catch (Exception e) {
         System.err.println("Systemfehler: " + e);
      }
   }
}
```

Implementierungstechnisch gesehen fehlt jetzt nur noch der Code, um den Server zu starten. Der Einfachheit halber wird angenommen, dass der Server nur eine Bestellung handhaben kann. Dieses Objekt von der Klasse `BestellungImpl` wird beim Hochlauf erzeugt, durch seinen Konstruktor beim Verzeichnisdienst registriert und kann anschließend von RMI-Clients kontaktiert werden.

Dabei wird deutlich, wie ein verbesserter Entwurf aussehen müsste: Man braucht einen Anwendungsdienst, z.B. `BestellDienst`, und von diesem ein singuläres Objekt. Dieses Objekt verwaltet die Bestellungen und damit indirekt auch die Bestellpositionen. Allerdings hätte das den Beispielcode nur aufgebläht und damit den Blick auf die wichtigen programmiertechnischen Konstrukte erschwert.

```
import java.rmi.*;
import java.rmi.server.*;
public class BestellungServer {
   public static void main(String args[]) {
      try {
         BestellungImpl bestellung =
                        new BestellungImpl("meine_best");
         System.out.println("Der Bestellungsserver läuft");
      }
      catch (Exception e) {
         System.err.println("Ausnahme: " + e.getMessage());
         e.printStackTrace();
      }
   }
}
```

Die Übersetzung besteht dann aus Routinearbeit, die mit der üblichen Suche nach syntaktischen und semantischen Fehlern verbunden ist:

- Mit dem Kommando `javac Bestellung.java BestellungImpl.java BestellungClient.java BestellungServer.java` wird der Quellcode in den Java-Bytecode übersetzt.
- Durch den Aufruf `rmic BestellungImpl` wird der Stub- und ggf. Skeleton-Code erzeugt. Das ist vergleichbar mit dem Aufruf des IDL-Compilers bei CORBA. Jetzt liegt der gesamte übersetzte Bytecode vor.
- Damit das Server- und Client-Programm richtig ablaufen kann, sind noch folgende administrativen Schritte notwendig; die jeweiligen Kommandos beziehen sich auf die Betriebssystemfamilie Windows:
  - Der Verzeichnisdienst muss mit `start rmiregistry` gestartet werden.
  - Der RMI-Server wird mit `start java BestellungServer` gestartet.
  - Schließlich kann ein Client mit `java BestellungClient` aufgerufen werden.

4.4 Enterprise JavaBeans (EJB)

4.4.1 Integration von Systemdiensten und Komponenten

Neben dem reinen Verteilungsaspekt wurden bei EJB von Anfang an zwei weitere wesentliche Aspekte adressiert, die für die (Weiter-)Entwicklung von verteilten Anwendungen essenziell sind:

- *Systemdienste*, wie z.B. Verzeichnisdienst, Transaktionsdienst und Nachrichtendienst, sind wie bei CORBA von Anfang an berücksichtigt. Das ermöglicht eine in sich schlüssige und abgerundete Spezifikation, zumal auf »Altlasten« und Rückwärtskompatibilität keine Rücksicht genommen werden muss.
- Die mittlere Stufe einer verteilten Anwendung kann durch *Komponenten* realisiert werden. Man kann auch sagen, dass durch EJB der technische Rahmen geliefert wird, um Komponenten zu entwickeln und später wieder zu verwenden. Das ist ein wesentlicher Unterschied zu CORBA: Die Systemarchitekten und Entwickler werden durch die EJB-Technologie dazu angeleitet, Komponenten zu definieren und wieder zu verwenden. Bei CORBA hingegen gab es über viele Jahre hinweg die uneingeschränkte Freiheit, eine Anwendung nach eigenem Gutdünken zu entwerfen. Erst durch die EJB-Spezifikation sah sich das OMG-Konsortium gezwungen, ein Komponentenmodell zu verabschieden, nachdem man Jahre darüber diskutiert hatte.

4.4.2 Server, Container und Enterprise Beans

Der grundsätzliche Aufbau eines Applikationsservers in der mittleren Stufe durch die EJB-Technologie ist in Abbildung 4.11 dargestellt. Dabei gibt es im Prinzip eine

hierarchische Einbettung der EJB-Konzepte, damit sie ihre unterschiedlichen Aufgaben erfüllen können:

- Enterprise JavaBeans (kurz: *Enterprise Beans*) realisieren die Anwendungslogik und bilden dabei die serverseitigen Komponenten. Bei den EJB-Spezifikationen 1.0 und 1.1 gab es dafür zwei Ausprägungen: SessionBeans und EntityBeans. In der neuen EJB-Spezifikation 2.0 gibt es auch noch die MessageDrivenBeans, um eine bessere Integration mit dem Nachrichtendienst JMS (Java Message Service) zu ermöglichen. Auf diese drei Bean-Arten wird nach den grundsätzlichen Architekturbetrachtungen noch genauer eingegangen.
- Ein *EJB-Container* stellt die Infrastruktur bereit, damit Enterprise Beans ausgeführt werden können. Dazu gehört vor allem der Transaktionsdienst, die systemtechnische Unterstützung für den Lebenszyklus von Enterprise Beans (d.h. Erzeugen, Auslagern und Löschen), die technische Verbindung zu den Clients und der Zugriffsschutz (Security), damit nur autorisierte Clients die Methoden der Enterprise Beans aufrufen können. Prinzipiell kann man auch sagen, dass ein EJB-Container der Ort ist, an dem ein Enterprise Bean existiert – genauso wie ein Datensatz in einer Datenbank abgespeichert ist.
- Durch den *EJB-Server* wird wiederum die Infrastruktur für einen EJB-Container bereitgestellt. Dabei wird vor allem eine komfortable Unterstützung für die systemnahe Programmierung angeboten, für z.B. die Verwaltung von Threads, die Verwaltung von Datenbankverbindungen und die Wiederverwendung von Instanzen der Enterprise Beans (Pooling).

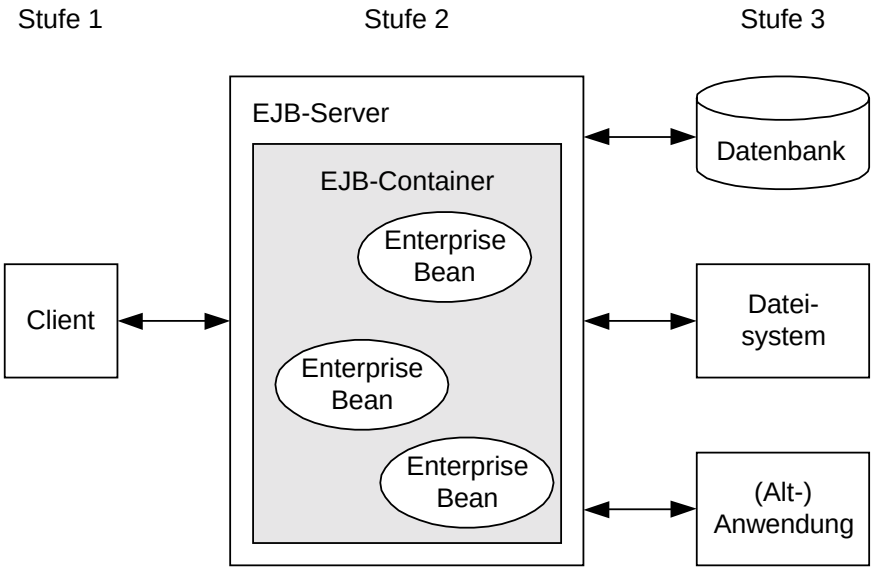

Abbildung 4.11: Enterprise JavaBeans in der mittleren Stufe

Als Backend-Systeme sind natürlich (relationale) Datenbanken von Interesse, weil sie heutzutage die dominierende Technik darstellen, um die unternehmenskritischen Daten zu verwalten. Ein EJB-Server kann aber auch auf Dateisysteme zugrei-

fen und zusammen mit der neuen Connector-Technologie auf andere Anwendungen zugreifen, insbesondere Altanwendungen, die nach einem anderen Paradigma entwickelt wurden. Dieser Aspekt wird in Kapitel 12 wieder aufgegriffen, wo auch die Integration mit CORBA und COM+ betrachtet wird.

4.4.3 SessionBeans und EntityBeans

Die beiden Bean-Arten, die bereits in der Spezifikation 1.0 und 1.1 enthalten waren, sind die SessionBeans und die EntityBeans.

SessionBeans

Innerhalb des EJB-Servers wird ein Client als *SessionBean* repräsentiert. Der Name soll dabei bewusst suggerieren, dass das SessionBean eine interaktive Verbindung (Session) mit einem Client darstellt. Deshalb hat ein SessionBean auch zwei wesentliche Eigenschaften:

- Ein SessionBean kann nicht zwischen mehreren Clients aufgeteilt werden, sondern ist genau einem Client zugeordnet.
- Mit dem Ende einer Client-Session wird auch das zugehörige SessionBean beendet; es kann also nicht dauerhaft (persistent) abgespeichert werden.

Zur Unterstützung des Clients stellt ein SessionBean die Anwendungsdienste bereit, die der Client zur Ausführung einer bestimmten Aufgabe benötigt. Das kann man auch mit dem aus Smalltalk bekannten MVC-Konzept (Model View Control) vergleichen. Der »View«-Anteil wird durch die Darstellung realisiert, wie man sie in einem Dialog auf der Client-Seite findet. Das bedeutet nicht zwangsläufig, dass die Informationen über die grafische Präsentation in einem »Thick-Client« gespeichert sind; auch diese Informationen können auf der Serverseite gespeichert sein und zu Beginn eines Dialoges oder einer Dialogsequenz vom Server zum Client übertragen werden.

Der »Model«-Anteil besteht aus dem Code, der die eigentlichen Anwendungsdaten bereitstellt. Dazu gehören insbesondere die Verarbeitungsroutinen, die von der Client-Seite aus aufgerufen werden können. Ähnlich ist die Aufgabe von SessionBeans: Sie realisieren im Prinzip den »Model«-Anteil in der MVC-Denkweise. Der Verarbeitungsaspekt für die realisierte Geschäftslogik steht dabei im Vordergrund, wie das in Abbildung 4.12 dargestellt ist.

EntityBeans

Ein *EntityBean* wird verwendet, um die persistenten Daten eines Geschäftsobjekts zu verwalten; das sind die Daten, die nach Beendigung einer Applikation immer noch verfügbar sind, weil sie z.B. in einer Datenbank gespeichert sind. Im einfachsten Fall entspricht ein EntityBean einem Datensatz in einer Tabelle einer relationalen Datenbank; es ist aber auch möglich, dass ein EntityBean mehreren Datensätzen entspricht oder sich auf Daten aus anderen (Legacy-)Anwendungen bezieht.

Zum Abspeichern der Datenfelder eines EntityBeans gibt es zwei Mechanismen: *Bean-kontrollierte Persistenz (Bean-Managed Persistence, BMP)* und *Container-kontrollierte Persistenz (Container-Managed Persistence, CMP)*. Bei BMP muss explizit auspro-

grammiert werden, wie auf die persistenten Datensätze mit z.B. JDBC zugegriffen wird. Bei CMP wird diese Abbildung der persistenzfähigen Datenfelder eines EntityBeans auf ein Backend-System generiert, so dass an dieser Stelle keine Programmierung notwendig ist. Allerdings ist die von CMP bereitgestellte Funktionalität bis zur EJB-Spezifikation 1.1 ziemlich dürftig, weil keine Beziehungen unterstützt werden. Erst ab der Spezifikation 2.0 kann man auch komplexe Anwendungen mit CMP adäquat modellieren.

Die persistenten Datensätze werden i.A. von mehreren Clients angefordert, weshalb ein EntityBean mehreren Clients zur Verfügung stehen kann bzw. muss. Das ist ein wesentlicher Unterschied zu einem SessionBean, das für genau einen Client existiert. Wenn auf Datensätze einer Datenbank zugegriffen wird, muss das innerhalb einer Transaktion erfolgen, wie man in Kapitel 6 sehen wird. Deshalb ist es wichtig, dass EntityBeans mit dem Transaktionsdienst JTS und seiner Schnittstelle JTA zusammenspielen.

Es ist zu erwarten, dass hauptsächlich auf persistente Datensätze aus einer relationalen Datenbank zugegriffen wird. Deshalb gibt es auch das Konzept der Primärschlüssel für EntityBeans, so wie man das bei relationalen Datenbanken kennt: Über den Primärschlüssel wird ein Datensatz bzw. ein EntityBean eindeutig identifiziert. Beim Beispiel des Pizza-Services ist das z.B. die Kundennummer für die Kunden oder die Bestellnummer für die Bestellungen. Beim Anlegen eines EntityBeans muss man eine separate Klasse bereitstellen, um den Primärschlüssel zu modellieren. Daran kann man auch erkennen, dass Sun Microsystems mit den EntityBeans vor allem auf (objekt-)relationale Datenbanken abzielt und weniger auf objektorientierte Datenbanken.

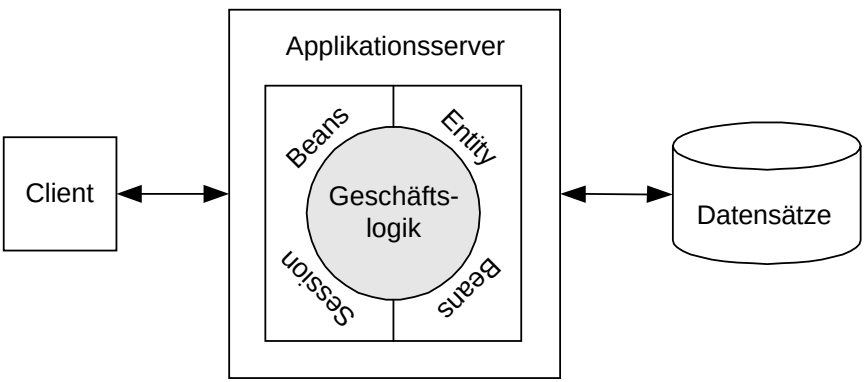

Abbildung 4.12: Zusammenspiel von SessionBeans und EntityBeans

Deployment Descriptor

Zur Beschreibung der Enterprise Beans gibt es noch jeweils einen so genannten Deployment Descriptor. Der Deployment Descriptor ist eine XML-Datei, in der vor allem die transaktionalen Eigenschaften beschrieben und der Zugriffsschutz (Security) anhand von Rollen festgelegt wird.

4.4.4 EJB-Spezifikation Version 1.1

Die EJB-Spezifikation hat sich hinsichtlich des Komponentenmodells sukzessive weiterentwickelt, ohne dass es zu Technologiebrüchen oder Inkompatibilitäten gekommen ist. Dabei sind stets die Erfahrungen aus neuen EJB-Produkten wie auch aus den Projekten der Praxis mit eingeflossen. Die Spezifikation in der Version 1.1 hatte sich darauf konzentriert, die Funktionalität aus Version 1.0 abzurunden und Spezifikationslücken zu beseitigen. Dazu gehören vor allem die folgenden Bereiche, die bereits jetzt erwähnt werden, auch wenn die Grundlagen für die EJB-Technologie noch nicht gelegt sind; das verschafft aber einen Eindruck über die Entwicklung der Spezifikation:

- EntityBeans sind nicht mehr optional, sondern müssen unterstützt werden.
- Das Format für den Deployment Descriptor ist durch XML festgeschrieben und nicht mehr frei wählbar.

4.4.5 EJB-Spezifikation 2.0

Die Spezifikation 2.0 hingegen hebt sich deutlich von den Vorgängerversionen ab. Rein optisch kann man das daran erkennen, dass für die Spezifikation 1.1 noch ca. 300 Seiten Beschreibung genügt haben, während für die Spezifikation 2.0 über 500 Seiten notwendig sind. Das kommt daher, dass einige neue Konzepte eingeführt wurden, um große verteilte Anwendungen noch einfacher entwickeln zu können. Die folgenden Konzepte werden nachfolgend kurz erklärt und in Kapitel 11 detailliert beschrieben:

- Ein neuer Persistenzmanager arbeitet mit dem abstrakten Persistenzschema des Deployment Descriptors, um CMP besser zu unterstützen.
- Beziehungen zwischen EntityBeans werden bei CMP berücksichtigt.
- Die Anfragesprache EJB QL vereinfacht das Aufsuchen von EntityBeans.
- Durch MessageDrivenBeans wird die Integration mit dem Nachrichtendienst JMS vereinfacht.

Persistenzmanager und abstraktes Persistenz-Schema

Ein neuer *Persistenzmanager* (Persistence Manager) wird in der Spezifikation 2.0 eingeführt, um die CMP zu verbessern, d.h. die automatische Abbildung der persistenzfähigen Felder eines EntityBeans auf eine relationale Datenbank wird durch den Persistenzmanager verbessert. Voraussetzung dafür ist, dass im Deployment Descriptor das *abstrakte Persistenzschema* in XML-Notation definiert wird. Aus dieser Definition wird später die Implementierung generiert, um aus der Datenbank die Werte für die persistenzfähigen Felder zu lesen und nach Modifikationen später wieder zurückzuschreiben.

Zusammen mit dem abstrakten Persistenzschema ist es gemäß der Spezifikation 2.0 auch möglich, *Beziehungen zwischen EntityBeans* zu definieren. Ohne diese Funktionalität war es bislang eigentlich nicht möglich, komplexe Anwendungen mit CMP nach der Spezifikation 1.1 zu entwickeln. Zur Definition von Beziehungen gibt es

im Deployment Descriptor der Version 2.0 die XML-Schlüsselwörter `<relationships>`, `<ejb-relation>`, `<ejb-relation-name>`, `<ejb-relationship-role-name>` usw. Damit kann man 1:1, 1:n und n:m Beziehungen darstellen. Die Spezifikation berücksichtigt sogar noch abhängige Beziehungen, um z.B. auszudrücken, dass ein Objekt der Klasse `Adresse` von einem Objekt der Klasse `Kunde` existenziell abhängig ist. Wenn also ein Kundenobjekt gelöscht wird, soll automatisch auch das Adressobjekt gelöscht werden; bei relationalen Datenbanken nennt man das »Cascading Delete«. Im Deployment Descriptor gibt es für abhängige Beziehungen die XML-Schlüsselwörter `<dependents>`, `<dependent>`, `<dependent-class>` usw.

EJB QL

Durch die Abfragesprache *EJB QL* kann man im Deployment Descriptor angeben, wie man EntityBeans finden kann, die bestimmten Suchkriterien genügen. Aus einer solchen Definition wird dann Code generiert, der die SQL-ähnliche Anfrage in JDBC umsetzt.

MessageDrivenBeans

Die letzte wesentliche Neuerung in der Spezifikation 2.0 besteht in der Einführung der *MessageDrivenBeans*. Die SessionBeans und EntityBeans kann man im Prinzip als RPC-basierte Komponenten ansehen, deren Methoden von einem Client aufgerufen werden können. Beispielsweise kann ein SessionBean als Client eines EntityBeans dessen Methoden aufrufen. Alle Methodenaufrufe sind dabei synchron, d.h., der Client ist blockiert, bis die Methode auf Serverseite abgearbeitet ist. Wenn nun ein SessionBean oder ein EntityBean eine Nachricht abschicken oder verteilen will, aber nicht weiter daran interessiert ist, was der oder die Empfänger mit einer solchen Nachricht tun, so kann die Nachricht an ein MessageDrivenBean geschickt werden, und das sendende Session- oder EntityBean ist nicht mehr blockiert. Durch die neue Kategorie der MessageDrivenBeans wird der Nachrichtendienst JMS besser mit der EJB-Technologie integriert.

Wie man sieht, wurde die EJB-Spezifikation in der Version 2.0 erheblich erweitert. Dazu haben viele Hersteller von Middleware- und/oder Datenbankprodukten beigetragen: Allaire, Art Technology Group, BEA Systems, Bluestone, Forte, Fujitsu, Gemstone (jetzt: Brokat AG), IBM, InLine Software, Inprise, IONA Technologies, iPlanet, Luna Information Systems, Oracle, Persistence, Progress Software, Secant, Siemens, Silverstream, Software AG, Sun Microsystems, Sybase, The Object People (jetzt: WebGain), Tibco und Vitria.

4.4.6 Einordnung von EJB 2.0, J2SE 1.3 und J2EE 1.3

Zu J2EE (Java 2 Enterprise Edition) gehören die APIs (Application Programming Interface) von J2SE (Java 2 Standard Edition) sowie eine Reihe von Erweiterungen. Bereits in J2SE 1.3 und damit auch in J2EE 1.3 enthalten sind:

- der Verzeichnisdienst JNDI 1.2.1 (Java Naming and Directory Interface)
- das Zugriffsprotokoll für (objekt)relationale Datenbanken JDBC 2.1 (Java Database Connectivity)

Nachfolgend sind die Erweiterungen in ihren aktuellen Versionen aufgeführt, die ebenfalls zu J2EE gehören:

- EJB 2.0
- RMI-IIOP 1.0 als interoperables Zugriffsprotokoll zwischen EJB- und CORBA-Umgebungen
- der Transaktionsdienst JTS 1.0 (Java Transaction Service) und JTA 1.0.1 (Java Transaction API)
- der Nachrichtendienst JMS 1.0.2 (Java Message Service)
- Servlets 2.3 und JSP 1.2 (JavaServer Pages) als Erweiterungen für (Web-)Server

In Abschnitt 4.5 werden Servlets und JSP vorgestellt. Die Systemdienste JNDI, JTS und JMS werden in Kapitel 5 erläutert und den korrespondierenden CORBA-Diensten gegenübergestellt. JDBC wird in Kapitel 8 eingeführt, und auf RMI-IIOP wird im Rahmen von heterogenen Plattformen in Kapitel 12 eingegangen.

Durch die Ergänzung von EJB um Systemdienste ist die Voraussetzung gegeben, um unternehmenskritische, verteilte Anwendungen mit J2EE zu erstellen. Im Juli 2000 hat Sun Microsystems erstmals eine Produktfamilie mit dem J2EE-Zertifikat ausgezeichnet: WebLogic von BEA Systems bestehend aus WebLogic Server, WebLogic Enterprise, WebLogic Commerce Server und WebLogic Personalization Server.

Es gibt noch weitere wichtige Funktionalitäten, die nicht in der aktuellen J2EE-Spezifikation 1.3 enthalten sind. Zunächst wurde nämlich der Schwerpunkt darauf gelegt, dass die wichtigsten Basistechnologien enthalten sind, und in späteren Versionen werden weitere ergänzende Technologien aufgenommen. Das betrifft die Connector-Spezifikation, die in Kapitel 12 im Zusammenhang mit heterogenen Plattformen vorgestellt wird. Ebenso ist SQLJ, das in Kapitel 9 zur Einbettung von SQL in Java eingeführt wird, noch nicht Bestandteil von J2EE 1.3.

4.5 Webtechnologien und JavaBeans

Nachdem die grundsätzlichen Technologievarianten für eine objektorientierte Middleware eingeführt sind, wird in diesem Abschnitt vorgestellt, wie ein Webserver in eine mehrstufige Architektur mit einem Applikationsserver integriert werden kann. Danach werden Servlets und die darauf aufbauenden JavaServer Pages (JSP) eingeführt, mit denen die Programmierung eines Webservers im Vergleich zur herkömmlichen CGI-Programmierung (Common Gateway Interface) erheblich vereinfacht wird. Abschließend werden JavaBeans als Komponententechnologie für die Clients vorgestellt, um das gesamte Bild abzurunden.

4.5.1 Mehrstufige Architektur mit einem Webserver

In Abbildung 4.13 ist eine mehrstufige J2EE-Architektur dargestellt, bei der zwei Aspekte gegenüber den bislang angestellten Überlegungen auffallen: Es werden zwei Arten von Clients unterschieden, und in der mittleren Stufe ist ein Webserver berücksichtigt. Die Clients außerhalb der Firewall sind i.A. Webbrowser, die auf den

Webserver zugreifen; die Clients innerhalb der Firewall können ebenfalls Webbrowser sein oder aber normale Java-Anwendungen, die dann direkt auf einen EJB-Server zugreifen.

Der Zugriff auf den EJB-Server erfolgt mit RMI-IIOP, wie es in der EJB-Spezifikation 2.0 vorgesehen ist. Das betrifft also den Webserver wie auch die echten Clients innerhalb der Firewall. Abbildung 4.13 veranschaulicht, dass mit den J2EE-Technologien die Funktionalitäten bzw. Anwendungsdienste eines Applikationsservers auch über einen Webserver verfügbar gemacht und somit durch Webbrowser genutzt werden können.

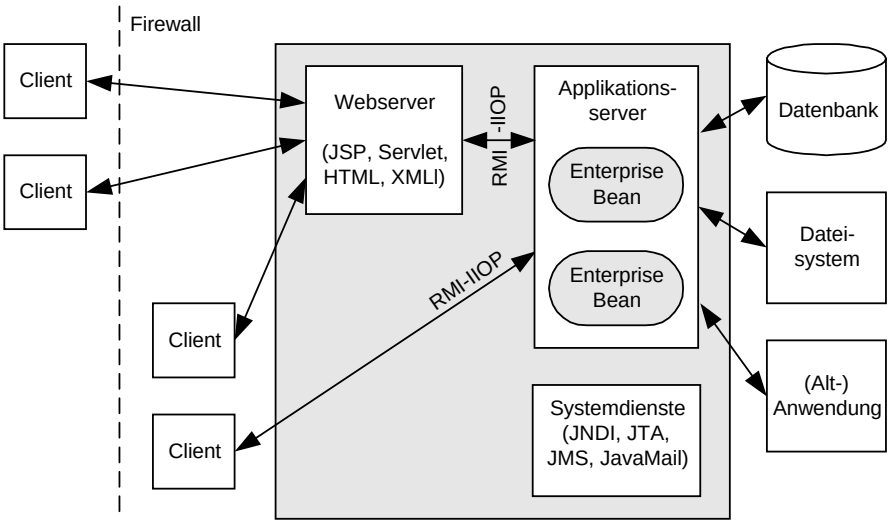

Abbildung 4.13: Mehrstufige J2EE-Architektur mit einem Webserver

Ob der Webserver und der EJB-Server auf derselben oder auf heterogenen Plattformen laufen, ist für die folgenden Betrachtungen irrelevant, die auf Servlets und JSP ausgerichtet sind, mit denen die Funktionalität innerhalb des Webservers realisiert wird. Servlets und JSP sind Bestandteil von J2EE 1.2 und lassen sich folgendermaßen charakterisieren:

- *Servlets* erweitern die Funktionalität von Servern, die nach einem Request-Response-Prinzip arbeiten, d.h., ein Client fordert Daten oder – allgemeiner – einen Dienst an, der Server führt die angeforderte Dienstleistung aus und liefert das Resultat an den Client zurück. In erster Linie werden Servlets bei Webservern als plattformunabhängiger Java-Ersatz für die CGI-Programmierung eingesetzt, um Webseiten mit einem dynamischen Inhalt zu erzeugen. Servlets kann man als serverseitige Applets ansehen, die für (Web-)Server eine ganze Reihe von Vorteilen bieten, die im nachfolgenden Abschnitt erläutert werden.
- *JavaServer Pages (JSP)* bauen technologisch gesehen auf Servlets auf. Der Fokus richtet sich aber darauf, dass die Benutzbarkeit vereinfacht wird, was insbesondere für Autoren von dynamischen Webseiten und für die von ihnen benutzten Werkzeuge wichtig ist.

4.5.2 Servlets

Aus Sicht eines Clients ist eine Webanwendung nichts weiter als eine Ansammlung von HTML- und XML-Seiten, die beim Server gegebenenfalls noch durch Servlets, JSP und weitere Ressourcen angereichert sind. Die Organisationsform beim Server kann eine Verzeichnisstruktur sein, oder die Bestandteile können in einem Archivierungsformat, wie z.B. WAR (Web ARchive), vorliegen. Diese Gegebenheit hat sich historisch entwickelt.

Bereits bei den ersten Webanwendungen hat sich gezeigt, dass statische HTML-Seiten nicht ausreichen und zu unflexibel sind, wenn man z.B. Datensätze aus einer Datenbank in einer HTML-Seite anzeigen will. Deshalb war es ein Ziel, dass man die darzustellenden Daten ändern kann, ohne den Code der Anwendung ändern zu müssen. Dann kann ein Administrator von z.B. einer E-Commerce-Anwendung neue (Produkt-)Daten hinzufügen, die dann unmittelbar in der Webanwendung verfügbar sind.

Von CGI-Skripten zu Servlets

Um Webseiten mit dynamischem Inhalt erstellen zu können, wurde *ursprünglich CGI (Common Gateway Interface)* entwickelt. Damit kann man mit Skripten Daten aus einer Datenbank oder aus Dateien extrahieren und in z.B. einer HTML-Tabelle (Form) darstellen. In der Folgezeit entstanden unzählige plattformspezifische CGI-Skripten, die aber alle dieselben Schwächen aufwiesen. Wenn ein Webserver ein CGI-Skript aufruft, wird bei jedem Aufruf ein neuer Prozess durch das Betriebssystem erzeugt. Das ist ressourcenintensiv, teuer und nur bis zu einem gewissen Grad skalierbar.

Eine weitere Schwäche betrifft die architektonische Konzeption von CGI. Es wird nämlich keine Trennung zwischen dem Inhalt (d.h. den Daten) einer Webseite und der Präsentationslogik vorgenommen. Deshalb muss ein CGI-Programmierer beide Aspekte beherrschen. Das stellt hohe Anforderungen an die Programmierer und erschwert im Normalfall auch die Wartbarkeit.

Die ersten Antworten auf CGI-Skripten waren herstellerspezifische Erweiterungen für Webserver wie z.B. *NSAPI* für den Netscape Enterprise Server, *Modules* für Apache und *ISAPI* für den Internet Information Server von Microsoft. Damit war eine serverseitige, multithreaded Verarbeitung möglich, ohne dass jedes Mal ein neuer Prozess gestartet werden musste. Allerdings war die resultierende Anwendung nicht plattformneutral, sondern abhängig von proprietären Erweiterungen der Hersteller. Ebenso gab und gibt es proprietäre Lösungen der Datenbankhersteller. Damit kann ein Webserver effizient auf ihre Datenbanken zugreifen, und es ist i.A. ein Pooling der Datenbankverbindungen verfügbar, so dass nicht bei jedem Zugriff eine neue Datenbankverbindung hergestellt werden muss.

Servlets haben gegenüber CGI-Skripten und proprietären Servererweiterungen eine Reihe von Vorteilen. Zunächst einmal basieren Servlets auf Java und sind damit sowohl plattformunabhängig als auch typsicher – im Vergleich zu CGI-Skripten mit beispielsweise Perl. Diese plattformunabhängige Lösung bringt aber auch die notwendigen Performance-Verbesserungen, weil ein Servlet(-Objekt) einmal in den

virtuellen Speicher eines Webservers geladen wird und fortan beliebig oft wieder verwendet werden kann.

Ein Servlet-Objekt läuft beim Webserver in einem leichtgewichtigen Thread statt in einem teuren Prozess. Dadurch, dass ein Servlet-Objekt im Speicher verbleibt, wird eine Datenbankverbindung auch nur ein Mal aufgebaut und kann fortan beliebig oft benutzt und aufgerufen werden. Weiterhin können mit Servlets sogar die Daten einer so genannten Benutzer-Session verwaltet werden, was angesichts des zustandslosen HTTP-Protokolls weitere Performance-Vorteile bringt.

Bei der Spezifikation der Servlets hat Sun Microsystems mit vielen renommierten Firmen und Einrichtungen zusammengearbeitet: The Apache Developer Community, Art Technology Group, BEA Systems, Clear Ink, Gefion Software, IBM, Live Software, Netscape Communications, New Atlanta Communications und Oracle. Servlets sind heute auf den meisten Serverplattformen verfügbar. Dazu gehören beispielsweise Apache, Netscape Enterprise Server, WebLogic Server von BEA Systems und WebSphere von IBM. Wenn eine Serverplattform keine Servlets unterstützen sollte, gibt es einige so genannte Servlet-Engines, um die Servlet-Funktionalität bereitzustellen, wie z.B. ServletExec, Jrun und WAICoolRunner.

Lebenszyklus eines Servlets

In Abbildung 4.14 ist der Lebenszyklus für ein Servlet-Objekt dargestellt, damit man die prinzipielle Arbeitsweise verstehen kann. Zunächst wird vom Server die Servlet-Klasse geladen, was prinzipiell mit folgendem Aufruf erfolgt:

```
Class cls = Class.forName("com.pizzaservice.BestellServlet");
```

Von dieser Klasse wird mindestens ein Objekt erzeugt, und auf jedem Objekt wird die Methode `init` aufgerufen, in der die initialen Parameter durch Aufruf der Methode `getInitParameter` gelesen werden können.

```
Servlet srv = (Servlet) cls.newInstance();
srv.init();
```

Nun ist das Servlet-Objekt bereit, Anforderungen eines Clients entgegenzunehmen. Wenn eine solche Anforderung eingeht, wählt der Server ein Servlet-Objekt aus, und die Methode `service` oder eine spezifischere Methode (z.B. `doGet` oder `doPost`) wird aufgerufen. Dazu werden zwei weitere Objekte benötigt: Ein Objekt zur Java-Schnittstelle `ServletRequest` oder `HttpServletRequest` wird aus den Daten erzeugt, die vom Client bei dessen Anfrage mitgeliefert wurden; und ein Objekt zur Java-Schnittstelle `ServletResponse` oder `HttpServletResponse` wird erzeugt, um das Ergebnis der Verarbeitung aufzunehmen. Ob `ServletRequest` oder `HttpServletRequest` zu verwenden ist, hängt davon ab, ob die Servlet-Klasse von der Java-Schnittstelle `GenericServlet` oder von `HttpServlet` abgeleitet ist.

```
ServletRequest request = ...;      // oder HttpServletRequest
ServletResponse response = ...;    // oder HttpServletResponse
srv.service(request, response);
```

Danach wird das Resultatsobjekt an den Client zurückgeliefert, und das Servlet-Objekt wartet auf seinen nächsten Auftrag. Dieser Zyklus wiederholt sich so lange, bis der Server die Methode `destroy` auf dem Servlet-Objekt aufruft.

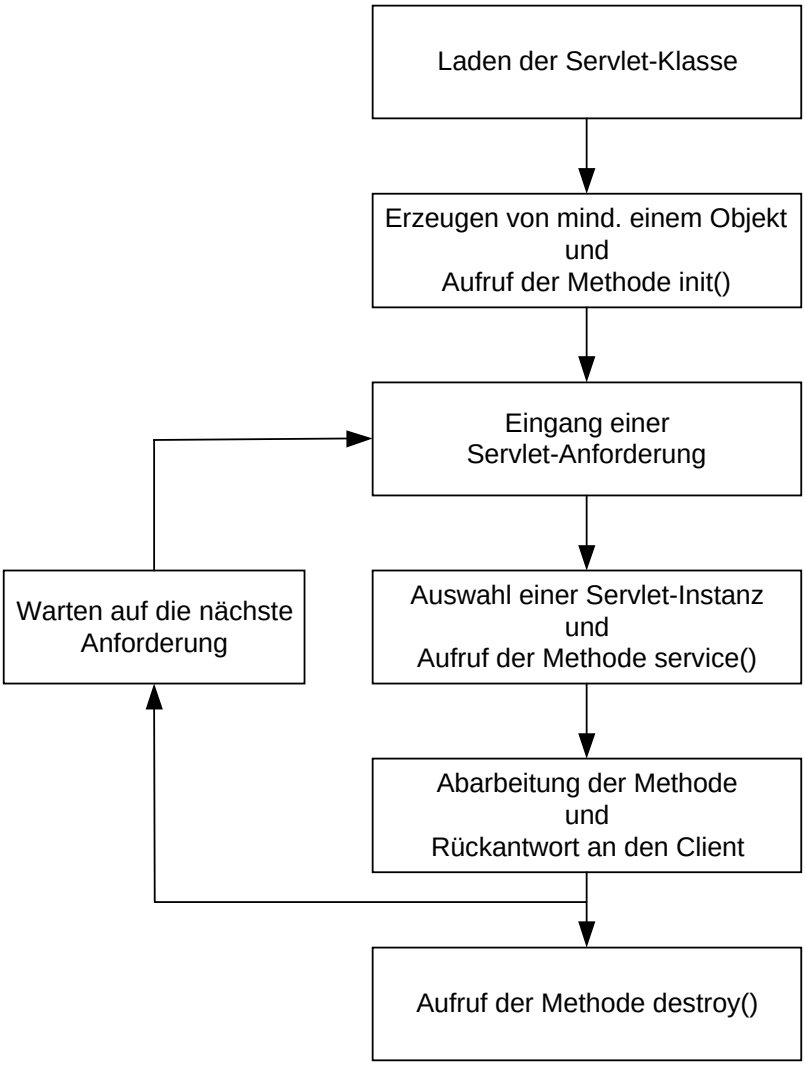

Abbildung 4.14: Lebenszyklus eines Servlet-Objekts

Java-Pakete: javax.servlet und javax.servlet.http

Bei der Betrachtung des Lebenszyklus wurden bereits einige Java-Schnittstellen und Java-Klassen erwähnt, die für Servlets relevant sind. Abbildung 4.15 zeigt in einer Übersicht die Pakete *javax.servlet*, das allgemeine und HTTP-unabhängige Funktionalität enthält, sowie *javax.servlet.http*, das die Funktionalität für das HTTP-Protokoll enthält.

In der Java-Schnittstelle `Servlet` werden die grundlegenden Methoden `init`, `service`, `destroy` und `getServletConfig` deklariert. Um ein generisches Servlet zu implementieren, das nicht für die Client-Interaktion über das HTTP-Protokoll bestimmt ist, leitet man die neue Servlet-Klasse von der Klasse `GenericServlet` ab und implementiert die Methode `service`. Standardimplementierungen für `init` und `destroy` werden in `GenericServlet` bereitgestellt, können aber für spezifischere Initialisierungen oder Aufräumarbeiten überladen werden.

Die Java-Schnittstelle `ServletRequest` deklariert die Methoden, um Daten aus einer Client-Anfrage zu extrahieren. Dazu sind vor allem drei Methoden wichtig. Mit `getParameter` kann man zu einem gegebenen Parameternamen den übergebenen Wert extrahieren, und `getParameterValues` kann man verwenden, um eine `Enumeration` zu erhalten, wenn es zu einem Parameternamen mehrere Werte gibt. Mit `getInputStream` erhält man ein Objekt der Klasse `ServletInputStream`, mit dem man binäre Daten lesen kann, was häufig bei den HTTP-Operationen `POST` und `PUT` verwendet wird. `ServletInputStream` ist eine von `java.io.InputStream` abgeleitete abstrakte Klasse.

Eine weitere administrative Methode ist `getContentLength`, um die Länge der Client-Daten zu ermitteln. Das ist sinnvoll im Zusammenhang mit den HTTP-Operationen `POST` und `PUT`, wenn z.B. Dateien zum Server übertragen werden (File Upload). Mit `getContentType` kann man den MIME-Typ der Client-Daten ermitteln. Die typischen MIME-Typen für Webanwendungen sind `text/plain`, `text/html`, `image/gif` und `image/jpeg`. Über die Methoden `getServerName` und `getServerPort` kann man schließlich den Rechnernamen und die Portnummer ermitteln, wo die Client-Anfrage eingegangen ist.

Durch die Java-Schnittstelle `ServletResponse` kann man die Daten zusammenfassen, die an den Client als Antwort auf seine Anfrage zu schicken sind. Dazu gehören zunächst die administrativen Methoden `setContentLength` und `setContentType`. Mit `setContentLength` werden die HTTP-Kopfdaten für die Länge gesetzt, damit der Client überprüfen kann, ob er auch alle Daten erhalten hat. Mit `setContentType` wird der MIME-Typ für die Rückgabedaten gesetzt. Durch die Methode `getOutputStream` erhält man ein Objekt der Klasse `ServletOutputStream`, die von `java.io.OutputStream` abgeleitet ist. Damit kann man dann Binärdaten zum Client zurückübertragen. Mit der Methode `getWriter` erhält man analog ein Objekt der Klasse `PrintWriter`, um textuelle Daten mit der Unicode-Codierung zurück zum Client zu schicken.

Die Schnittstelle `ServletConfig` deklariert die Methoden, um eine Konfiguration aufzubauen, mit der ein Servlet-Objekt in seiner Methode `init` initialisiert wird. Die Schnittstelle `ServletContext` ist insbesondere für Webserver interessant, bei denen Servlets laufen. Damit kann man dann z.B. Protokolldateien erstellen. Schließlich gibt es noch die beiden Klassen `ServletException` und `UnavailableException` für die Ausnahmebehandlung in Fehlersituationen.

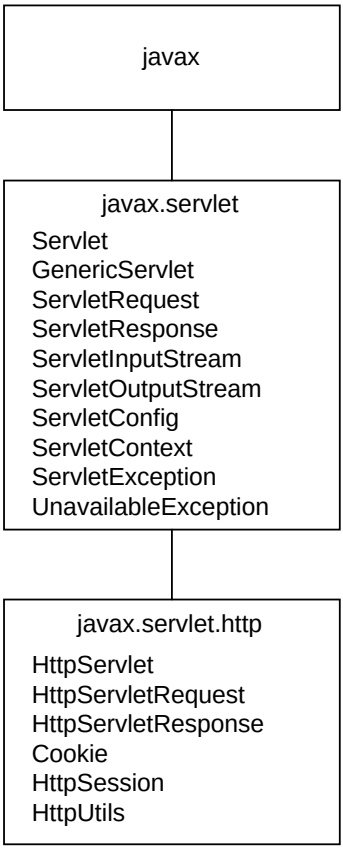

Abbildung 4.15: Java-Pakete für Servlets

Damit sind die Schnittstellen und Klassen aus `javax.servlet` vorgestellt. Im Paket `javax.servlet.http` finden sich Erweiterungen für den Einsatz von Servlets bei Anfragen mit dem HTTP-Protokoll. Die abstrakte Klasse `HttpServlet` ist von `GenericServlet` aus `javax.servlet` abgeleitet. In `HttpServlet` gibt es die Methode `doGet`, um auf eine `GET`-Anfrage zu reagieren, und die Methode `doPost` für eine `POST`-Anfrage. In den selbst implementierten Servlet-Klassen, die von `HttpServlet` abgeleitet sind, muss man die Methoden `doGet` und `doPost` überschreiben, um eine spezifische Verarbeitung der `GET`- und `POST`-Anfragen durchführen zu können. Die Standardimplementierung von `doGet` liefert nämlich nur die Fehlermeldung `"error 400, BAD_REQUEST"`.

Während mit einer `GET`-Anfrage Daten vom Webserver angefordert werden, kann man mit einer `POST`-Anfrage Daten zum Webserver übertragen. Diese Variante wird auch häufig verwendet, um umfangreiche Formulare oder lange Parameterlisten zu übertragen, falls die Länge der URL ein Problem sein könnte. Dabei muss man sich immer bewusst sein, dass `POST`-Anfragen vom Webbrowser speziell behandelt und nicht wiederholt werden können, weshalb es nicht sinnvoll ist, darauf ein Bookmark zu setzen. `GET`-Anfragen hingegen lassen sich wiederholen. Unter diesen

Umständen ist es also immer angebracht, die POST-Methode zu verwenden, wenn auf der Serverseite eine sensible Verarbeitung angestossen wird, z.B. eine Bezahlung mit einer Kreditkarte oder wie in unserem Beispiel eine Bestellung einer Pizza.

Wenn im Client bzw. Webbrowser Werte in ein Formular (HTML Form) eingegeben und via POST zum Webserver übertragen werden, so kann man damit auch die aktuellen Werte für eine parametrisierte SQL-Anfrage übertragen, die durch doPost ausgelöst wird. Das Resultat der SQL-Anfrage wird dann durch ein Objekt der Klasse HttpServletResponse zurück zum Client übertragen

Analog zu HttpServlet sind HttpServletRequest und HttpServletResponse von ServletRequest bzw. ServletResponse abgeleitet. Mit HttpServletRequest kann man die Kopfdaten extrahieren, die zu einer HTTP-Anfrage gehören. Diese Methoden korrespondieren mit den bekannten CGI-Variablen. Beispielsweise liefert die Methode getQueryString denselben Text, wie er auch in der CGI-Variablen QUERY_STRING enthalten ist. Weitere administrative Informationen sind erhältlich über die Methoden:

- getDateHeader, um die Anzahl an Sekunden zu erhalten, die seit dem 1.1.1970 vergangen sind.
- getMethod, um die durchgeführte HTTP-Operation (z.B. GET, POST, PUT) zu ermitteln.
- getQueryString, um die Parameter in der Anfrage zu extrahieren.

  Beispielsweise erhält man den Textstring "pizza=Tonno&anzahl=3" bei einem Servlet BestellServlet mit folgendem URL (Uniform Resource Locator)

  "http://localhost:8080/servlet/BestellServlet?pizza=Tonno&anzahl=3"

- getRequestURI, um den URI (Uniform Resource Identifier) zu ermitteln und als URL darzustellen. Dabei ist der Rechnername noch über die geerbte Methode getServerName und die Portnummer über die geerbte Methode getServerPort zu ergänzen.
- getCookies, um ein Array von Cookies zu erhalten, die der Client an den Server zurückschickt. Auf Cookies wird nachfolgend noch eingegangen.

In HttpServletResponse sind analog set-Methoden enthalten:

- setDateHeader, um das Datum in den Kopfdaten zu setzen.
- setHeader, um (Name,Wert)-Paare zu setzen, bei denen der Wert ein String ist.
- setIntHeader, um (Name,Wert)-Paare zu setzen, bei denen der Wert ein Integer ist.

Eine ganz wichtige Methode von HttpServletResponse ist die Methode addCookie, um ein *Cookie* zum Client zu übertragen. Damit kann ein Webserver Daten zum Client schicken, die dieser später mit jeder Anforderung mitschickt. Damit kann man benutzerspezifische Daten und Zustandsinformationen verwalten. Das erleichtert die Verarbeitung, weil das HTTP-Protokoll zustandslos ist.

Ein einfaches Beispiel

Damit sind die Java-Schnittstellen und Java-Klassen für Servlets eingeführt. Das nachfolgende Codefragment illustriert ein einfaches Servlet für einen Webserver, wozu der Einfachheit halber nur die Methode `doGet` überschrieben wird. Auf Fehler- und Ausnahmebehandlung wurde zu Gunsten der Übersichtlichkeit verzichtet.

In der generierten HTML-Seite wird das Datum enthalten sein, das in den Kopfdaten der Client-Anfrage zu sehen war. Dabei wird vorausgesetzt, dass dort das Datum bei dem Parameternamen `"Datum"` angegeben wurde.

```
import javax.servlet.*;
import javax.servlet.http.*;
public class BestellServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                    HttpServletResponse response) {
    response.setContentType("text/html");
    ServletOutputStream out = response.getOutputStream();
    out.println("<HTML>");
    // Kopfdaten
    out.println("<HEAD>");
    out.println("<TITLE>Bestell-Servlet</TITLE>");
    out.println("</TITLE>");
    // Nutzdaten im Rumpf
    out.println("<BODY>");
    out.println("<H1>Eingabe der Bestelldaten</H1>");
    long requestDatum = request.getDateHeader("Datum");
    Date datum = new Date(requestDatum);
    out.println("Bestelldatum: " + datum.toString() + "<P>");
    ...
    out.println("</BODY>");
    out.println("</HTML>");
    response.setContentLength(out.length());
    out.close();
  }
}
```

Aufruf von Servlets

Die Verarbeitung eines Servlets wird angestoßen, wenn in einer HTML-Seite das Tag `SERVLET` vorkommt oder wenn bei der Verarbeitung eines Formulars ein Servlet für `ACTION` verwendet wird. Das erklärt auch, warum im Zusammenhang mit Servlets von *serverseitigen Includes* gesprochen wird.

Das nachfolgende Codefragment illustriert, wie eine einfache HTML-Seite aussieht, mit der man in einem Formular die Pizza Tonno bestellen kann.

```
<HTML>
<HEAD>
<TITLE>Bestellung mit einem Servlet</TITLE>
```

```
<BODY>
Beginn der Bestellung<P>
<FORM METHOD=GET ACTION="/servlet/BestellServlet">
   Wie h&auml;fig wird die Pizza Tonno bestellt?
   <INPUT TYPE=TEXT NAME=Tonno SIZE=2>
   <INPUT TYPE=SUBMIT VALUE="Abschicken">
</FORM>
Ende der Bestellung<P>
</BODY>
</HTML>
```

Statt eines Formulars könnte man auch das Tag `SERVLET` verwenden, um die Pizza Tonno fünf Mal zu bestellen. Die HTML-Seite würde dann folgendermaßen aussehen:

```
<HTML>
<HEAD>
<TITLE>Bestellung mit einem Servlet</TITLE>
<BODY>
Beginn der Bestellung<P>
<SERVLET CODE=BestellServlet>
   <PARAM NAME=Tonno VALUE=5>
</SERVLET>
Ende der Bestellung<P>
</BODY>
</HTML>
```

Das nachfolgende Codefragment veranschaulicht die Implementierung des zugehörigen Servlets, das eine solche einfache Bestellung entgegennimmt. Auf die eigentliche Verarbeitung durch ein nachgeschaltetes Enterprise Bean wurde dabei verzichtet. Dass der Inhalt der HTML-Seite dynamisch aufgebaut wird, ist durch die Verwendung der Methode `println` offensichtlich.

```
import javax.servlet.*;
import javax.servlet.http.*;
public class BestellServlet extends HttpServlet {
  public void doPost(HttpServletRequest request,
                     HttpServletResponse response) {
    response.setContentType("text/html");
    PrintWriter out = response.Writer();
    // Anzahl für Tonno ermitteln
    String strTonnoAnzahl = out.getParameter("Tonno");

    int tonnoAnzahl = Integer.parseInt(strTonnoAnzahl);
    // ... eigentliche Verarbeitung der Bestellung ...
    out.println("Tonno: " + tonnoAnzahl + " Mal bestellt<P>");
  }
}
```

Cookies und Sessions

Cookies wurden erstmals mit dem Netscape Navigator eingeführt, damit ein Webserver Daten auf der Client-Seite puffern kann. Details dazu können unter *http://home.netscape.com/newsref/std/cookie_spec.html* nachgelesen werden. Mit einem Cookie ist es auch möglich, den Ablauf einer *Session* zu verfolgen; unter einer Session versteht man eine Abfolge von Anfragen desselben Client(-Rechners) für die Dauer einer Verbindung. Die Verfolgung einer Session ist allerdings besser durchführbar mit der Java-Schnittstelle `HttpSession` aus *javax.servlet.http*, die anschließend diskutiert wird.

In der Klasse `Cookie` aus *javax.servlet.http* ist die Funktionalität der Cookies zusammengefasst und gekapselt. Beim Konstruktor kann ein (Name,Wert)-Paar mitgegeben werden, und dann gibt es noch eine Reihe von `set`-Methoden, um z.B. Domäne oder maximale zeitliche Gültigkeit zu setzen. Ein Cookie kann der Webserver dann mit den Kopfdaten bei einer HTTP-Antwort mitschicken, wozu es in `HttpServletResponse` die Methode `addCookie` gibt. Später kann das Cookie vom Webserver wieder verarbeitet werden, wenn die Methode `getCookies` aus `HttpServletRequest` aufgerufen wird, die eine `Enumeration` von `Cookie`-Objekten liefert. Das nachfolgende Codefragment illustriert den prinzipiellen Umgang mit Cookies:

```
HttpServletResponse response = ...;
String userId = ...;
Cookie cookie = new Cookie("userId", userId);
cookie.setDomain(".pizzaservice.com");
cookie.setPath("/");
response.addCookie(cookie);
```

Unter einer Session versteht man also die Abfolge von Client-Anfragen (Requests) an den Server, bis sich der Client ausloggt oder bis ein Timeout-Wert von z.B. 15 Minuten erreicht wird. Die Verwaltung von Daten, die sich auf eine Session eines Endbenutzers beziehen, ist bei Webanwendungen notwendig. Signifikante Beispiele sind:

- Wenn sich ein Endbenutzer bzw. Client einloggen muss, so soll das nur am Anfang und nicht bei jeder einzelnen HTML-Seite notwendig sein.
- Wenn die Zusammenstellung einer Bestellung Aktionen erfordert, bei denen verschiedene HTML-Seiten besucht werden, so muss sich der Server die Historie der bereits besuchten Seiten merken.

Die Client-/Server-Kommunikation mit dem HTTP-Protokoll ist jedoch zustandslos: Nachdem ein Server eine Client-Anfrage abgearbeitet hat, haben Client und Server keine Kenntnis mehr voneinander. Deshalb kann ein Server eigentlich keine Client-spezifischen Werte zwischenpuffern.

Um dennoch Zustandsinformationen zu puffern, haben die Programmierer von Webanwendungen bislang vielfältige (Not-)Lösungen eingesetzt. Dazu gehören beispielsweise versteckte Felder in Formularen, das Setzen und Lesen von Cookies sowie die Codierung in URLs. Ab der Servlet-Spezifikation 2.0 wurde das Aufzeich-

nen von Session-Informationen jedoch einfacher: Es wurde die Java-Schnittstelle `HttpSession` eingeführt, mit welcher der Webserver die Aufzeichnung von Session-Informationen erledigen kann.

Das Objekt, das die aktuelle Session repräsentiert, kann man mit der Methode `getSession` aus der Java-Schnittstelle `HttpServletRequest` ermitteln. Wenn noch kein Session-Objekt existiert, wird ein neues erzeugt, falls an `getSession` der Parameter `true` übergeben wurde. Ansonsten wird `null` zurückgeliefert. Ein Objekt von `HttpSession` erhält bei der Erzeugung eine eindeutige Session-ID, die eine Zuordnung zum Client beinhaltet. Diese Session-ID wird dann vom Client implizit mit jeder Anfrage mitgeschickt. Damit kann der Server den abschickenden Client und die zugehörigen zwischengepufferten Daten identifizieren.

`HttpSession` enthält eine Reihe von Methoden zur Verwaltung einer Session:

- `getId` liefert die Session-ID als String. Diese Session-ID kann man dann mit dem Resultat der Methode `getRequestedSessionId` aus `HttpServletRequest` vergleichen, um festzustellen, ob die Session noch gültig oder mittlerweile abgelaufen ist.
- `setMaxInactiveInterval` setzt den Timeout-Wert in Sekunden, wozu ein Integerwert als Parameter übergeben wird.
- `getLastAccessedTime` liefert die Anzahl an Millisekunden, die seit dem 1.1.1970 bis zum letzten Request vergangen sind.
- `invalidate` beendet die Session.
- `setAttribute` hat zwei Parameter: einen String für den Attributnamen und ein `Object` für den dazugehörigen Wert. Bis zur Servlet-Spezifikation 2.1 war der Methodenname `putValue`, der ab 2.2 »depreciated« ist.
- `getAttribute` liefert das `Object` zum angegebenen Namen. Bis zur Servlet-Spezifikation 2.1 war der Methodenname `getValue`, der ab 2.2 »depreciated« ist.

Auswirkungen auf J2EE

Damit eine verteilte J2EE-Anwendung mit Servlets interagieren kann, dürfen innerhalb von `HttpSession` (aus dem Paket *javax.servlet.http*) nur Objekte folgender Schnittstellen verwendet werden:

- `java.io.Serializable`
- `javax.ejb.EJBObject`
- `javax.ejb.EJBHome`
- `javax.transaction.UserTransaction`
- `javax.naming.Context`

Die Java-Schnittstellen `UserTransaction` und `Context` werden bei den Systemdiensten in Kapitel 5 vorgestellt, und die Java-Schnittstellen `EJBObject` und `EJBHome` werden im Zusammenhang mit der EJB-Technologie in Kapitel 11 eingeführt.

4.5.3 JavaServer Pages (JSP)

Schichtenarchitektur mit Servlets als Grundlage

JavaServer Pages (JSP) sind wie Servlets in J2EE enthalten, um Java und HTML in verteilten Anwendungen zu kombinieren. Während bei Servlets viel Arbeit mit der Handhabung und Programmierung des dynamischen Inhalts einer HTML-Seite zugebracht wird, liegt der Schwerpunkt bei JSP auf dem Design einer HTML-Seite. Dabei soll außerdem möglichst viel vorhandene Technologie wieder verwendet werden.

Deshalb baut JSP auf Servlets auf, und die Clients tauschen mit einem Server Objekte aus, die zu den Servlet-Schnittstellen `HttpServletRequest` und `HttpServletResponse` gehören, wie es in Abbildung 4.16 dargestellt ist. Ähnlich wie ein Enterprise Bean in einen EJB-Container eingebettet ist, hat eine JSP-Seite einen JSP-Container als Laufzeitumgebung. Eine JSP-Seite wird dabei durch eine Java-Klasse implementiert, die von der Basisschnittstelle `Servlet` (aus dem Paket *javax.servlet*) abgeleitet ist. Durch diese Vererbung ist auch die Möglichkeit gegeben, auf Anwendungslogik zuzugreifen, die durch Enterprise Beans realisiert ist, was in Abbildung 4.13 bereits skizziert wurde.

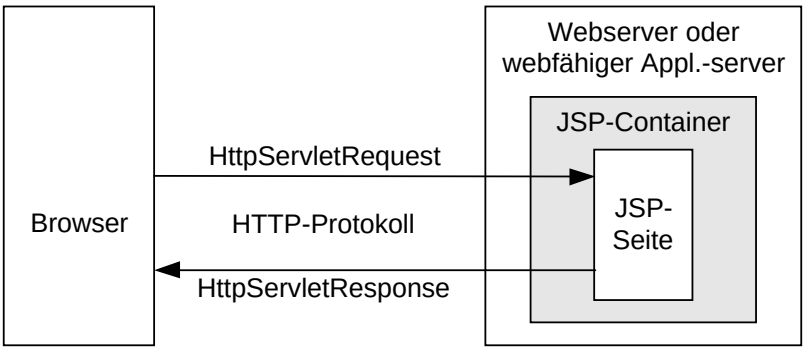

Abbildung 4.16: Einordnung von JSP-Container und JSP-Seite

Übersetzung

Ein JSP-Compiler ist notwendig, um eine JSP-Seite in Servlet-Code zu übersetzen, wie das in Abbildung 4.17 dargestellt ist. Nach Übersetzung einer JSP-Seite können die resultierenden Servlet-Klassen mit statischen HTML-Seiten, Multimedia-Daten (z.B. Video und Audio) und serverseitigen JavaBeans zu einem Webarchiv (WAR) zusammengebunden werden, was ab der Servlet-Spezifikation 2.2 möglich ist.

Bei diesem Übersetzungsvorgang wurde unterstellt, dass die JSP-Seite zu Beginn übersetzt und in das Archiv eingebunden wurde. Es ist aber auch möglich, JSP-Seiten im Quellcode zu installieren. Dann werden sie beim ersten Zugriff übersetzt, was die Zugriffszeit für den Endbenutzer allerdings entsprechend verlängert.

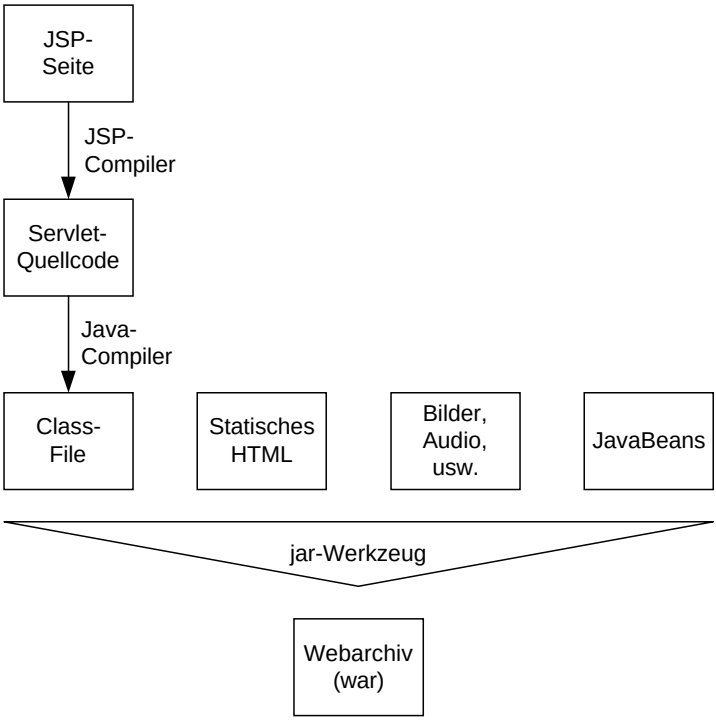

Abbildung 4.17: Übersetzungs- und Archivierungsvorgang

Bestandteile von JSP

Eine JSP-Seite ist aus folgenden Bestandteilen zusammengesetzt:

- feste Template-Daten
- Direktiven
- Skriptelemente
- Aktionen

Feste Template-Daten sind HTML-Elemente wie `<UL>` und `<LI>`, um Listen von zu berechnenden Elementen zu definieren. Damit wird eine Trennung zwischen statischem und dynamischem Inhalt einer Seite möglich.

Eine *Direktive* beinhaltet globale Informationen, die unabhängig von einer konkreten Client-Anfrage sind. Solche Informationen werden oft für die Übersetzungsphase genutzt, wenn z.B. die Skriptsprache festgelegt wird. Auf jeden Fall erzeugen die Direktiven keine Daten für den Ausgabestrom einer JSP-Seite.

Generell lautet die Syntax für Direktiven folgendermaßen:

```
<%@ directive attr1="value1" attr2="value2" ... %>
```

Die JSP-Spezifikation sieht einige vordefinierte Standarddirektiven vor:

- Mit der Direktive `page` legt man Attribute für eine JSP-Seite fest, wie z.B.
  - `<%@ page info="Meine erste Seite für JSP 1.1">` enthält die zu einer Seite gehörige Information, die später auch der Endbenutzer sieht.
  - `<%@ page isThreadSafe="yes" errorPage="/allgFehler.jsp" %>` spezifiziert, dass die Seite Thread-sicher ist und dass im Fehlerfall die Seite `allgFehler.jsp` anzuzeigen ist.
  - Weitere Attribute sind z.B. `language`, um die Skriptsprache festzulegen, `import` (wie in einem normalen Java-Programm) und `buffer` für die Puffergröße des Ausgabestroms. In der JSP-Spezifikation 1.1 kann für das Attribut `language` nur `"java"` angegeben werden.
- Die Direktive `include` ermöglicht, dass eine Datei in eine JSP-Seite eingefügt wird, wie z.B. `<%@ include file="copyright.html" %>`.
- Die Direktive `taglib` ermöglicht, dass die bei JSP standardmäßig vorhandenen Tags erweitert werden können. Weitere Standard-Tags werden weiter unten im Zusammenhang mit den so genannten Aktionen noch vorgestellt.

Zu den Skriptelementen gehören Deklarationen, Scriptlets und Ausdrücke. Mit den *Deklarationen* werden Variablen und Methoden für die Skriptsprache bekannt gemacht, d.h. auch sie produzieren keine Daten für den Ausgabestrom. Eine Deklaration beginnt mit `<!` und sieht für eine Integervariable beispielsweise so aus:

```
<! int i = 0; %>
```

Eine Methode `m` mit einem Integerargument `i` und dem Rückgabetyp `String` wird folgendermaßen deklariert:

```
<! public String m(int i) { if (i < 50) ... } %>
```

In einem *Scriptlet* können irgendwelche Codefragmente enthalten sein, welche die Sprache zulässt, die beim Attribut `language` der Direktive `page` angegeben wurde; in der JSP-Spezifikation 1.1 ist nur Java möglich. Die Scriptlets werden ausgeführt, wenn die Client-Anfrage abgearbeitet wird, und können Daten für den Ausgabestrom produzieren. Die Kombination aller Scriptlets (bzw. Codefragmente) müssen eine gültige Anweisung oder Sequenz von Anweisungen in der Skriptsprache ergeben.

Um beispielsweise eine dynamisch berechnete Grußformel für den Endbenutzer einzubauen, kann folgendes Scriptlet verwendet werden:

```
<% if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) { %>
   Guten Morgen!
<% } else { %>
   Guten Tag!
<% } %>
```

Als letztes Element für eine JSP-Seite gibt es noch die *Aktionen*. Wie die Skriptlets werden die Aktionen bei der Abarbeitung einer konkreten Anfrage ausgeführt. Dabei können auch Objekte erzeugt werden, die über deklarierte Variablen an die Scriptlets übergeben werden können. Die Syntax der Aktionen lehnt sich an XML an, wie das nachfolgende Beispiel zeigt. Durch das Tag `jsp:useBean` wird auf ein Objekt zugegriffen, das mit einem eindeutigen Bezeichner beim Attribut `id` deklariert wird. Die zugehörige Klasse wird mit dem Attribut `class` bekannt gemacht. Der eindeutige Bezeichner kann gleichzeitig als Variable verwendet werden und wird im nachfolgenden Beispiel an zwei Scriptlets übergeben.

```
<HTML>
  Hallo!
  <BR>
  <jsp:useBean id="clock"
               class="calendar.JspCalendar" />
  Heute ist
  <UL>
    <LI>Tag:  <% clock.getDayOfMonth() %>
    <LI>Jahr: <% clock.getYear() %>
  </UL>
</HTML>
```

Ein weiteres Attribut für `jsp:useBean` ist `beanName`, um ein JavaBean bekannt zu machen; JavaBeans werden im nächsten Abschnitt vorgestellt. Mit dem Attribut `scope` wird der Gültigkeitsbereich für die Variable festgelegt. Dazu gibt es `page` für eine Seite, `request` für eine konkrete Anfrage, `session` für eine Endbenutzer-Session als Abfolge mehrerer Anfragen und `application` für die gesamte Anwendung.

Weitere wichtige Standard-Tags sind:

- `<jsp:include>` wird verwendet, um weitere Ressourcen (z.B. Dateien) dynamisch einzubinden und so Präsentation und Anwendungslogik zu trennen. Mit der weiter oben vorgestellten Direktive `include` kann man einen Dateinamen statisch spezifizieren.
- `<jsp:forward>` erlaubt, dass eine eingehende Client-Anfrage auf einen anderen URL umgeleitet wird.
- `<jsp:param>` ermöglicht die Definition von (Name,Wert)-Paaren, die dann innerhalb von `<jsp:forward>` und `<jsp:include>` verwendet werden können.

JSP und/oder Servlets

Damit sind JSP mit ihren Bestandteilen eingeführt, und es lässt sich abschließend die Frage beantworten, wann JSP und wann Servlets verwendet werden sollten:

- Wenn die Zielsprache weder HTML noch XML ist, kann man nur Servlets, aber nicht JSP verwenden.
- Ansonsten kann man JSP einsetzen, wenn der Fokus auf dem Layout liegt und einfache Skripten ausreichend sind.

- Für komplexe Anwendungen sollte man Servlets und JSP kombinieren. Mit Servlets kann man die Client-Anfragen (Requests) empfangen und verarbeiten, und mit JSP wird dann die Antwort dynamisch erzeugt. Das ist in Abbildung 4.18 illustriert.

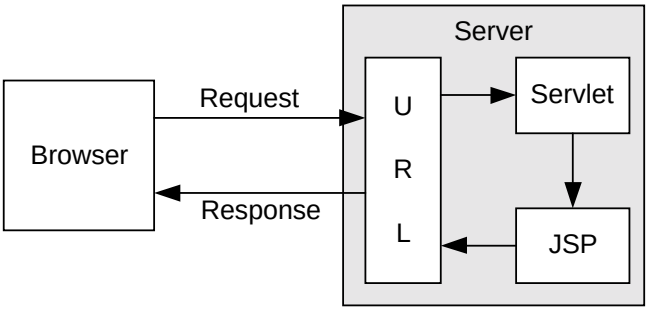

Abbildung 4.18: Verwendung von Servlets und JSP

4.5.4 Einordnung von JavaBeans

JavaBeans stellen die erste Komponententechnologie für die Java-Plattform dar und unterstützen die Wiederverwendung von Funktionalität für die Client-Seite. Dazu heißt es in der Spezifikation: »*A JavaBean is a reusable software component that can be manipulated visually in a builder tool*«, was einen breiten Spielraum zulässt.

Wiederverwendung hat sich in vielen Anwendungsgebieten bewährt, um eine bessere Stabilität, Qualität und Effizienz zu erhalten. Für die Wiederverwendung von Klassen gibt es verschiedene etablierte Konzepte:

- Vererbung von Attributen und Methoden
- Delegation einer durchzuführenden Verarbeitung an eine andere Klasse
- Instanziierung von Template-Klassen (z.B. bei C++)

Diese Art der Wiederverwendung ist auf jeden Fall feingranular. Ein gröberes Granulat liegt vor, wenn man ganze Verzeichnisse (Directory) oder Bibliotheken (Library) bzw. Java-Pakete (Package) wieder verwendet. Dabei stellt sich jeweils die Frage, ob das wieder zu verwendende Element in das neu zu erstellende System passt.

Deshalb ist es wichtig, dass wieder verwendbare Elemente bzw. Komponenten über anpassbare Eigenschaften verfügen und über Methoden und/oder Ereignisse konfiguriert werden können. Diese abstrakte Anforderung wird sowohl von JavaBeans als auch von Enterprise Beans erfüllt, wie man in Kapitel 11 sehen wird. Idealerweise erfolgt die Konfigurierung dabei über ein grafisches Werkzeug, wie z.B. die BeanBox aus dem Programmpaket BDK (Bean Development Kit).

Bei der Arbeit mit JavaBeans können drei verschiedene Kategorien von Anwendern unterschieden werden:

- Anwendungsprogrammierer nutzen vorhandene JavaBeans für ihre Progammierung und kombinieren sie zu einer Anwendung. Dabei setzen sie i.A. ein Werkzeug ein. Für die Anwendungsprogrammierer genügt es, wenn sie die Schnittstellen der JavaBeans kennen; sie müssen nicht mit der internen Implementierung der JavaBeans vertraut sein.
- Bean-Entwickler müssen mit dem API für JavaBeans vertraut sein und selbstverständlich die Interna der von ihnen entwickelten JavaBeans kennen.
- Entwickler von grafischen Entwicklungswerkzeugen, wie z.B. GUI-Builder oder BeanBox, müssen mit dem API für JavaBeans vertraut sein und die Infrastruktur für die Bean-Entwickler bereitstellen.

Ein JavaBean muss nach bestimmten Konventionen erstellt werden, um die Anforderung nach Anpassbarkeit und Konfigurierbarkeit zu erfüllen. Zunächst einmal gibt es *Namenskonventionen* für die öffentlichen (public) Methoden. Diese werden unterschieden in Methoden für den Zugriff auf Eigenschaften (Properties), in Methoden für die Ereignisverarbeitung und in normale Methoden.

Eigenschaften (Properties)

Die Gesamtheit aller anpassbaren Eigenschaften ergibt den Zustand eines JavaBeans. Methoden für den Zugriff auf einzelne anpassbare Eigenschaften müssen verschiedenen Konventionen folgen. Dazu lassen sich die Eigenschaften in verschiedene Kategorien unterteilen:

- *Boolesche Eigenschaften* (Boolean Property) können nur zwei verschiedene Zustände annehmen. Die Namenskonvention zum Lesen des Zustandes lautet:

```
boolean get<Property>()
boolean is<Property>()
```

- *Indizierte Eigenschaften* (Indexed Property) basieren auf Arrays, damit Werte an bestimmten Positionen gelesen und gesetzt werden können. Die Namenskonvention lautet:

```
<PropertyTyp>[] get<Property>()
void            set<Property>(<PropertyTyp>[] wert)
<PropertyTyp>   get<Property>(int index)
void            set<Property>(int index, <PropertyTyp> wert)
```

- *Gebundene Eigenschaften* (Bound Property) verschicken ein Ereignis, sobald sich ihr Wert ändert. Zu den Lese- und Schreibmethoden gibt es noch Methoden, um so genannte Listener zu verwalten, was weiter unten bei der Ereignisverarbeitung noch erläutert wird:

```
<PropertyTyp> get<Property>()
void          set<Property>(<PropertyTyp> neuerWert)
void addPropertyChangeListener(PropertyChangeListener cl)
void removePropertyChangeListener(PropertyChangeListener cl)
```

- Bei *eingeschränkten Eigenschaften* (Constraint Property) wird ebenfalls ein Ereignis verschickt, wenn sich der Wert des Ereignisses ändert. Allerdings haben die Listener dann ein Vetorecht und können eine Änderung verhindern. Dazu gibt es die Klasse `VetoableChangeListener`. Die Namenskonvention der Methoden lautet:

```
<PropertyTyp> get<Property>()
void          set<Property>(<PropertyTyp> neuerWert)
void addVetoableChangeListener(VetoableChangeListener vl)
void removeVetoableChangeListener(VetoableChangeListener vl)
```

- Schließlich gibt es noch die regulären Eigenschaften (Regular Property), die eine Get- und Set-Methode bereitstellen:

```
<PropertyTyp> get<PropertyName>()
void          set<PropertyName>(<PropertyTyp> neuerWert)
```

Ereignisverarbeitung

Die *Ereignisverarbeitung* beruht darauf, dass Ereignisse erzeugt werden, auf die anschließend reagiert werden kann. Bei einer grafischen Benutzungsoberfläche wird z.B. ein Ereignis erzeugt, wenn ein Button gedrückt wird; das ist dann ein `ActionEvent`. Nach dem Muster »Event-Listener« können sich Objekte als Listener registrieren, werden dann vom Eintritt eines relevanten Ereignisses informiert und können letztendlich mit einer parameterlosen Methode darauf reagieren. Die Listener gehören zu einer Klasse, die von `java.util.EventListener` abgeleitet ist. Die Klasse eines Ereignisses muss von `java.util.EventObject` abgeleitet sein.

Die Namenskonvention für Methoden der Ereignisverarbeitung berücksichtigt, dass neue Listener hinzugefügt und vorhandene Listener entfernt werden können:

```
void add<EventListenerName>(<EventListenerName> neuerListener)
void remove<EventListenerName>(<EventListenerName> listener)
```

Hilfsklassen

Zusätzlich zu den Konventionen für die Eigenschaften und Ereignisse gibt es normalerweise noch einige *Hilfsklassen*:

- BeanInfo-Klassen beschreiben die öffentliche Schnittstelle eines JavaBean.
- PropertyEditor-Klassen können verwendet werden, um die Eigenschaften der JavaBeans an eine neue Anwendung anzupassen.
- Customizer-Klassen sind eine AWT-Komponente und ermöglichen eine komplexere Anpassung von JavaBeans, die über die Möglichkeiten der PropertyEditor-Klassen hinausgeht.

Anhand der Konventionen konnte man sehen, dass JavaBeans nicht von einer bestimmten Basisklasse oder -schnittstelle abgeleitet sein müssen. Da aber viele Beans AWT-Komponenten darstellen, sind sie i.A. von `java.awt.Component` abgeleitet. Unter dieser Randbedingung des praktischen Einsatzes versteht man auch, welches Granulat für ein JavaBean als Komponente zu wählen ist. Die Spannbreite reicht dabei nämlich von einfachen AWT-Komponenten bis zu Spreadsheets, die innerhalb einer grafischen Oberfläche verwendet werden können.

Diese kurze Einführung in JavaBeans soll genügen, um einen Eindruck zu erhalten, was mit Komponententechnologie machbar ist und welche Voraussetzungen erfüllt sein müssen, um Komponenten wieder verwenden zu können.