# Java Server und Servlets

Professionelle Programmierung

PEARSON Education

ADDISON-WESLEY

Peter Roßbach / Hendrik Schreiber

# Java Server und Servlets

Portierbare Web-Applikationen effizient entwickeln

2., aktualisierte Auflage

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist
bei der Deutschen Bibliothek erhältlich.



Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt. Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.


ein Imprint der Pearson Education Deutschland GmbH
Martin-Kollar-Straße 10–12, D-81829 München/Germany


10 9 8 7 6 5 4 3 2 1
04 03 02 01 00
ISBN 3-8273-1694-4

*Einbandgestaltung*: vierviertel Gestaltung, Köln
*Lektorat*: Susanne Spitzer, sspitzer@pearson.de
*Korrektorat:* Friederike Daenecke, Zülpich
*Satz*: reemers publishing services gmbh, Krefeld
*Belichtung, Druck und Bindung*: Schoder, Gersthofen
*Produktion:* TYP*isch* Müller, Arcevia, Italien
Printed in Germany

Für Regina

P.R.

*»Life is Vegas.«*

H.S.

# Inhalt

# Vorwort

Ein Buch über Servlets sollte es werden.

Als wir im Herbst 1997 mit der Entwicklung unserer ersten servletbasierten Anwendung begannen, mußten wir feststellen, daß dazu keinerlei gedruckte Literatur existierte. Zwar fand sich im Netz und insbesondere in diversen Mailing-Listen schnell eine sehr aktive Gemeinde zusammen – Gedrucktes ließ aber weiterhin auf sich warten. Das Apache-JServ-Projekt entwickelte eine 0.9.x-Version nach der anderen (`http://java.apache.org`), Sun warf den Java Webserver auf den Markt und IBM zog nach mit ServletExpress, der frühen Grundlage des jetzigen Websphere. Gleichzeitig kamen Live Software mit ihrem Plug-In JRun groß raus. Aber immer noch gab es nichts Schwarz auf Weiß – außer ein paar kurzen Artikeln in den einschlägigen Fachzeitschriften [Heid 97, Roßbach/Schreiber 97].

Nach den ersten Gehversuchen wurde uns schnell klar, daß das reine Servlet-API zwar kleine und feine Lösungen für eben solche Probleme versprach, jedoch nicht zum großen Durchbruch verhalf. So wurde die ersten Idee zu einer Erweiterung des APIs geboren, die wir später *Servlet Method Invocation (SMI)* tauften. Diese Erweiterung ermöglichte es uns, aufbauend auf dem Servlet-API, robuste und konfigurierbare Anwendungen zu bauen.

Dabei kam eins zum anderen: Wir benötigten einen Konfigurationsmanager, ein objekt-relationale Abbildung für Datenbanken und schrieben zum Schluß unsere eigene Servlet-Engine jo!. Ein Framework war entstanden: das WebApp-Framework.

Während der Entwicklung legten wir Wert darauf, alle Komponenten so flexibel wie möglich zu halten. Nur wenige der Abhängigkeiten innerhalb des WebApp-Frameworks sind starr und stur kodiert, die meisten sind durch Schnittstellen entkoppelt. Mehr zu diesem Designprinzip lesen Sie in Teil Zwei dieses Buches.

Durch all diese Aktivitäten wurde die reine Servlet-Programmierung immer mehr in den Hintergrund gedrängt, zugunsten von Ideen zur Anwendungs- und Server-Architektur, einer leistungsfähigen Schicht zum Speichern von Objektnetzen sowie Anwendungen für das Framework.

So mag das Buch an einigen Stellen den Eindruck vermitteln, daß es aus vielen Bereichen etwas erklärt, aber nichts richtig. Diesem Eindruck halten wir entgegen, daß das

Buch ganz sicher eines leistet: Es erklärt, wie man webbasierte Anwendungen baut. Daß man dazu gewisse Zutaten wie Servlets, eine Servlet-Engine, eine Erweiterung wie SMI und einen Persistenz-Layer benötigt, ist nicht unsere Schuld.

Wir glauben, daß erst alle diese Teile zusammen effizientes Arbeiten ermöglichen. Was nutzt es, wenn Sie ein tolles Mail-Servlet schreiben, das leider nicht in Ihre Anwendung eingebunden werden kann? Was nutzt Ihnen Programmcode, in den praktisch unwartbare SQL-Anweisungen eingebaut sind? Was machen Sie, wenn Ihre Anwendung plötzlich über einen eigenen, komfortableren Client verfügen muß und nicht mehr nur mit Webbrowsern funktionieren soll? Auf alle diese Fragen können Sie gelassen antworten, wenn Sie sich zu Beginn der Entwicklung Ihrer Software mit einigen Themen auseinandergesetzt haben – und eben nicht nur mit naturbelassener Servlet-Programmierung und »rohem« JDBC.

Das mit dem Buch vorgestellte Framework ist sicher kein Allheilmittel. Es kann Ihnen jedoch bei der Entwicklung besserer Anwendungen helfen. Auch, wenn Sie es niemals einsetzen, bekommen Sie ein Gefühl für die zu bewältigenden Probleme.

Mittlerweile gibt es einige gute Bücher über Servlet-Programmierung [Moss 98, Hunter/Crawford 98]. Und wenn dieses Buch im ersten Teil auch die Programmierung von Servlets abdeckt, so ist es dennoch kein reines Servlet-Buch.

Es ist ein Buch über die Architektur und den Bau von servletbasierten Anwendungen.

## Software zum Buch

Da das Thema sich sehr schnell entwickelt und die Halbwertszeit von Software und den dazugehörigen Dokumentationen generell gegen einen geringen Grenzwert zu konvergieren scheint, haben wir uns dazu entschlossen, diesem Buch keine CD-ROM beizulegen. Statt dessen existiert eine Website zum Buch. Die Adresse lautet: `http://www.webapp.de`.

Wir empfehlen Ihnen dringend, sich das WebApp-Framework von der Website herunterzuladen, damit Sie die Beispielanwendungen selbst nachvollziehen können.

## Notation in UML

In allen drei Teilen des Buches verwenden wir Diagramme, um Design und Abläufe plastisch zu verdeutlichen. Die verwendete Notation ist UML 1.0 [Oestereich 98]. Zum Erstellen der Diagramme benutzten wir Rational Rose für Java.

## Fragen und Anregungen

Wir begrüßen den offenen Dialog über dieses Buch und die Servlet-Szene. Zu diesem Zweck haben wir die Mail-Adresse `book@webapp.de` eingerichtet. Bitte senden Sie Ihre Gedanken, Anregungen, Idee und Verbesserungsvorschläge. Sie sind herzlich willkommen.

## Danksagungen

Unser Dank gilt den Kollegen der FACTUM Projektentwicklung und Management GmbH Phillip Ghadir, Michael Jürgens, Wolfgang Neuhaus, Frank Peske, Henning Steiner und Axel Terfloth, für ihren Willen, immer wieder noch ein Kapitel zu korrigieren und zu kommentieren. Allen Mitgliedern des intraNEWS-Team, die mit uns innerhalb von 14 Monaten ein verteiltes Informationssystem auf Basis eines servletbasierten Applikationsservers realisiert haben, danken wir für ihre Ideen und ihren Einsatz. Letztendlich waren sie wichtiger Impulsgeber und Motivation für dieses Buch.

Ein ganz besonderer Dank gebührt Frank Wegmann für seine präzise und konstruktive Kritik. Sein Perfektionismus hat vieles erst ins rechte Licht gerückt.

Zudem gilt unser Dank Ulrich Büttgen, der mit Argusaugen frühe Versionen des Frameworks und des Buchs unter die Lupe nahm und uns wertvolle Hinweise gab.

Ferner danken wir unserer Lektorin Susanne Spitzer für die unkomplizierte Zusammenarbeit und die Freiheit, die sie uns bei der Realisierung dieses Buches gewährte. Es ist ein etwas anderes Java-Buch geworden und wir hoffen, daß der Erfolg sie dafür belohnen wird.

*Peter Roßbach*

Meiner Familie danke ich für die viele Zeit. Meiner Frau Regina Potthoff insbesondere für die Geduld und die aufmunternde Hilfe in schwierigen Phasen. Meinen Töchtern Josephine und Vivienne für die unschätzbare Freude und Abwechslung. Josephine hat uns ihren Namen für die Servlet-Engine jo! gegeben und ihre hilfreichen Worte des Trostes – »Alles wird Gut!« – haben mich zur Fertigstellung getrieben.

*Hendrik Schreiber*

Ich danke allen, die mir geholfen haben dieses Buch zu schreiben – insbesondere Barbara, Rolf, Marc, Bernd und Enke. Ohne ihre Unterstützung, ihr Verständnis und ihre Geduld sowie einigen Nächten in »Keller« und »Soundgarden« wäre dieses Buch nie zustande gekommen.

*Bochum, Dortmund 1999*

# Vorwort zur 2. Auflage

Viel schneller als erhofft war die erste Auflage vergriffen und mußte nachgedruckt werden, um die Zeit bis zur zweiten Auflage zu überbrücken. Wer also dachte, daß das Thema javabasierte Webapplikation eine Eintagsfliege sei, lag eindeutig falsch.

Natürlich hat sich technologisch einiges seit Erscheinen der ersten Auflage getan. So veröffentlichte Sun die Version 2.2 der Servlet- Spezifikation sowie die Version 1.1 der Java Server Pages. Für uns Grund genug, das Buch gründlich zu überarbeiten.

Doch nicht nur technologische Imperative forderten diese Überarbeitung. Der Erfolg im deutschsprachigen Markt war so groß, daß Addison-Wesley sich entschieden hat, eine englische Ausgabe des Buches international zu vermarkten. Die nun vorliegende 2. Auflage ist die überarbeitete Version der englischen Ausgabe.

Wir wünschen Ihnen viel Spaß mit dem Buch!

*Bochum/Dortmund im Juli 2000*

*Peter Roßbach, Hendrik Schreiber*

# Teil I
# Basis

Im ersten Teil dieses Buches widmen wir uns den Grundlagen der Server- und Servlet-Programmierung in Java. Kapitel 1 erklärt zunächst die einfachsten Grundlagen sowohl des World Wide Webs als auch von Java-Sockets und -Threads. Es soll Ihnen zu einem leichten Einstieg in die Materie verhelfen, Ihnen ein wenig Geschichte, die Basis-Begriffe, die Abläufe und Techniken näherbringen. Obwohl an einigen Stellen bei Null angefangen wird, eignet sich dieses Kapitel genauso wenig wie die folgenden als Einstieg in die Programmiersprache Java [Cornell/Horstmann 96, Campione/Walrath 97].

In Kapitel 2 zeigen wir die Entwicklung eines einfachen Webservers, anhand dessen ein tieferes Verständnis für die Abläufe zwischen Webbrowser und Webserver vermittelt wird. Es geht hier darum zu zeigen, wie einfach es ist, in Java Server zu programmieren. Dazu wird das Prinzip der Service-Handler-Architektur vorgestellt. Nebenbei erfahren Sie eine ganze Menge über HTTP, dem Übertragungsprotokoll des Webs.

In Kapitel 3 schließlich erläutern wir ausführlich das Servlet-API mit all seinen Möglichkeiten. Ausführlich wird auf die verschiedenen Schnittstellen und Klassen der Programmierschnittstelle eingegangen. Zudem werden Tips gegeben, wie man robuste Servlets baut. Zusätzlich zu den immer wieder eingestreuten kleineren Beispielen finden Sie am Ende des Kapitels 3 exemplarische Servlets für Standardanwendungen: Ein Mail-Servlet, ein Datenbank-Servlet und ein Gästebuch. Nach der Lektüre dieses Kapitels sollten Sie in der Lage sein, sicher und schnell Servlets zu programmieren. Diese Fähigkeit ist Grundlage für den zweiten und dritten Teil dieses Buches.

# 1 Grundlagen

Zu Beginn möchten wir kurz die wesentlichen technischen Grundlagen erläutern. Wer sich bereits bestens mit HTML, HTTP und CGI auskennt, kann die entsprechenden Kapitel getrost überspringen. Um die folgenden Kapitel verstehen zu können, müssen Sie sich gut mit Java-Threads und Sockets auskennen. Wenn das für Sie kein Problem ist, können Sie sich gleich ans Programmieren eines ersten Servers im nächsten Kapitel machen.

## 1.1 Hypertext Markup Language

Die Hypertext Markup Language (HTML) wurde von 1989 von Tim Berners-Lee am CERN in Genf entwickelt, um den dort arbeitenden Wissenschaftlern den Zugang zu Informationen zu erleichtern. Es sei einfacher, so Berners-Lee damals, etwas in einem nichtlinearen, nichthierarchischen Informationssystem wie dem Hypertext zu finden als in einem baumorientierten, hierarchisch gegliederten Katalog [Berners-Lee 89]. Konsequenterweise wurde am CERN bald HTML – im Grunde eine Anwendung der Standard Generalized Markup Language (SGML ISO 8879) – entwickelt.

Ein HTML-Dokument (Listing 1.1) besteht aus einem Kopf (Head) und einem Rumpf (Body). Der Kopf enthält den Titel und andere Meta-Informationen, der Rumpf den Dokumentinhalt. Auszeichnungen (Tags) dienen dazu, einem Abschnitt eine Eigenschaft zuzuweisen. So hat der Text zwischen dem Tag `<body>` und dem Tag `</body>` die Eigenschaft, der Körper des HTML-Dokuments zu sein. Daran sieht man bereits, daß meistens ein Anfangs-Tag und ein Ende-Tag existiert. Das Ende-Tag erkennen Sie an dem Schrägstrich nach dem Kleiner-Zeichen.

```
<html>
   <head>
      <title>Dieser Titel erscheint in der Titelzeile des Browsers
      </title>
   </head>
   <body>
      <h1>Dies ist eine Überschrift erster Ordnung</h1>
      An dieser Stelle könnte jetzt ein beliebiger Text stehen.
      Er könnte <em>betonte</em> Passagen enthalten,
```

```
      <b>fett gedruckte</b> oder auch <i>kursive</i>.
   </body>
</html>
```

*Listing 1.1: Exemplarisches HTML-Dokument*

`<body>` ist genau wie beispielsweise `<em>` (von emphasized, engl.: betont) eine logische Auszeichnung, d.h. sie schreibt nicht genau vor, wie der enthaltene Text dargestellt werden soll. So könnte ein mit `<em>` markierter Text von einem Browser unterstrichen, fett oder in einer anderen Farbe dargestellt werden. Wie der Text genau aussieht, ist egal, die Hauptsache ist, der Text wird hervorgehoben. Komplementär zu diesen inhaltlichen Auszeichnungen existieren auch physische. Beispiele dafür sind Fett, Unterstrichen oder Kursiv (`<b>`, `<u>` oder `<i>`).

Da das World Wide Web (WWW) ursprünglich lediglich für den einfachen Informationszugang und nicht als grafisches Medium gedacht war, bevorzugten die HTML-Kodierer die inhaltlichen Auszeichnungen. Doch schon bald entdeckten Layouter das WWW. Für die grafische Nutzung des Webs benötigten die Designer mehr Kontrolle. Die Firma Netscape gab sie ihnen nur allzu gern. Mit dem Browser Navigator führte Netscape im Laufe der Zeit Tabellen, Frames und andere Erweiterungen ein, die das genaue Layouten erleichterten. Das Web wurde zum grafischen Medium – sehr zum Leidwesen derer, die nicht über eine grafische Benutzeroberfläche verfügten und die Welt durch ein 80x25-Zeichen-Raster betrachten müssen.

Doch nicht nur die Layouter hatten ihre Wünsche, auch die HTML-Kodierer wünschten sich an so mancher Stelle ein bißchen mehr Funktionalität. Daher entwickelten die Programmierer von Netscape LiveScript, das später aus Marketinggründen in JavaScript umgetauft wurde. Microsoft wiederum baute in seine Browser Interpreter für die hauseigenen Skriptsprachen VBScript und JScript ein.

Der Wildwuchs an proprietären HTML-Erweiterungen nahm immer weiter zu. Gleichzeitig bemühte sich der Hüter des HTML-Standards, das World-Wide-Web-Konsortium (W3C), die Sprache flexibler zu gestalten und gleichzeitig die Trennung von logischer und physischer Auszeichnung zu fördern. Das Ergebnis sind die Cascading Style Sheets (CSS). Mit deren Hilfe können HTML-Programmierer ihren Code logisch auszeichnen und dennoch den Browser anweisen, einzelne Elemente einer Seite auf sehr spezifische Art und Weise zu formatieren. Leider ist der Grad der CSS-Unterstützung in den verschiedenen Browsern sehr unterschiedlich. Das gleiche gilt für das Document Object Model (DOM), das helfen soll, das Aussehen einer Seite über Skriptsprachen zu beeinflussen. Ebenso ist die Unterstützung der erweiterbaren Auszeichnungssprache XML (Extensible Markup Language) in Verbindung mit der XSL (Extensible Stylesheet Language) auf Browserseite noch alles andere als weit verbreitet.

Laut dem Web Standards Project (WASP, *http://www.webstandards.org*), einer Organisation, die für die Einhaltung von Standards kämpft, verursachen die Inkompatibilitäten zwischen verschiedenen Browsern und die Abweichungen von W3C-Standards etwa 25 Prozent Mehrkosten bei der Entwicklung einer Website.

Als Entwickler für das WWW trifft man mittlerweile drei Skriptsprachen in verschiedenen Versionen, die vierte HTML-Version und unterschiedlichste Unterstützung von CSS, DOM, XML und XSL an. Das sollte Grund genug dafür sein, sich stärker mit der serverseitigen Programmierung auseinanderzusetzen. Genau das wollen wir in diesem Buch tun.

## 1.2 Hypertext-Transfer-Protokoll

Das Hypertext-Transfer-Protokoll (HTTP) ist genau wie HTML 1989 von Tim Berners-Lee und seinen Mitstreitern am CERN entwickelt worden. Es dient im wesentlichen zur Übertragung von Daten, ist also nicht nur auf Texte beschränkt, wie man aus dem Namen schließen könnte. HTTP ist ein Protokoll auf Anwendungsebene, es ist zustandslos, in gewissem Maße objektorientiert und funktioniert nach einem simplen Anfrage/Antwort-Schema.

Die meisten einfachen HTTP-Server verwenden lediglich die Befehle `GET`, `HEAD` und `POST`. Mit `GET` und der Angabe eines Uniform Resource Identifiers (URI) werden die Ressourcen eines Servers angefordert. `POST` ermöglicht es, Daten an einen URI zu schicken, und `HEAD` liefert lediglich den Kopf einer Ressource, die sich hinter einem URI verbirgt. Daraus ergibt sich ein wesentliches Strukturmerkmal von HTTP-Antworten: Sie bestehen immer aus einem Kopf, auf den meistens ein Rumpf folgt.

In einem solchen Kopf wird unter anderem der Typ der per HTTP übertragenen Daten kodiert. Zu diesem Zweck werden MIME-Bezeichner (*Multipurpose Internet Mail Extensions*) verwendet. Sie sind in RFC 2045 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc2045.html`) definiert. HTML beispielsweise hat die Kennung »text/html«. GIFs (*Graphics Interchange Format*) werden mit »image/gif« gekennzeichnet. Andere wichtige Informationen, die sich im Kopf finden lassen, sind Cookies (Kapitel 3.8.4) oder das Datum der letzten Änderung eines Objekts.

Zur Zeit beherrschen die meisten Browser und Server lediglich die Version 1.0 des Protokolls [Berners-Lee et al. 96]. Einige Server, darunter auch der freie Server *Apache* und der *Java Webserver* von JavaSoft, beherrschen bereits die Nachfolgeversion HTTP 1.1 [Fielding et al. 99]. Einer der Hauptvorteile der neuen Version ist die Fähigkeit, über eine TCP-Verbindung mehr als eine Anfrage abzusetzen und über dieselbe Verbindung die entsprechenden Antworten zu erhalten. Unter HTTP 1.0 hingegen wird für jedes zu ladende Objekt eine neue Verbindung aufgebaut. Gerade bei Webseiten mit vielen Bildern oder Frames führt dies zur Ressourcenverschwendung.

Damit die Umstellung reibungslos vonstatten geht, ist HTTP abwärtskompatibel. Eine umfassende Beschreibung der Unterschiede zwischen HTTP 1.0 und HTTP 1.1 finden Sie in [Krishnamurthy et al. 98].

## 1.3 Common Gateway Interface

Das HTTP spezifiziert hauptsächlich den Transport von Objekten. Ein Mechanismus, mit dem die versandten Daten zu einer Anwendung gelangen, ist im HTTP jedoch nicht definiert. Lange Zeit war deshalb das von der *NCSA (National Center for Supercomputing Applications)* entwickelte *Common Gateway Interface (CGI)* der einzige Weg durch den HTTP-Server zu einem Programm.

Webserver, die das CGI benutzen, besitzen meist ein Verzeichnis mit dem Namen CGI-BIN (BIN = Binary, also ausführbar). In diesem Verzeichnis befinden sich oft Perl-Skripte oder kleine C-Programme. Wenn nun ein Client eine Anfrage an die URI eines dieser Programme sendet, reagiert der Webserver, indem er einen neuen Prozeß abspaltet und diesem Prozeß sämtliche Daten der Anfrage sowie einige Umgebungsvariablen zur Verfügung stellt. Die Ausgaben des gestarteten Programms werden als Antwort an den Client zurückgesendet, sofern sie in die Standardausgabe (stdout) geschrieben wurden.

Das Schöne an diesem Ansatz ist, daß der Anwendungsprozeß sauber vom Serverprozeß getrennt ist. Es handelt sich tatsächlich um zwei verschiedene, unabhängige Programme. Das bedeutet, daß CGI-Programme zum einen in jeder beliebigen Sprache geschrieben sein können und zum anderen den Server nicht zum Absturz bringen. Außerdem gibt es kein proprietäres API (*Application Programming Interface*, Anwendungsprogrammierschnittstelle), das sich von Serverversion zu Serverversion ändern könnte. Die Entwickler sind sogar weitgehend unabhängig vom Server selbst, sofern dieser ein CGI enthält.

Diesen Vorteilen steht jedoch ein großer Nachteil gegenüber: CGI hat einen riesigen Ressourcenhunger. Für jede Anfrage muß ein neuer Prozeß geschaffen werden, der zum Start initialisiert und nach dem Ausführen wieder beseitigt werden muß. In der Zwischenzeit belastet er das Betriebssystem, da Prozeßwechsel im Sinne von Rechenzeit recht teuer sind.

Die Konsequenzen können erdrückend sein: Stellen Sie sich einen gutbesuchten Server vor, dessen Hauptattraktion eine Anwendung ist, die über CGI gestartet wird. Nehmen wir an, diese Anwendung ist aufwendig und benötigt zur Ausführung fünf Sekunden auf einem unbelasteten System. Was passiert, wenn in einer Minute tausend Anfragen auf diesen Rechner einprasseln, dürfte klar sein.

Natürlich ist dieses Problem schon vor längerer Zeit erkannt worden. Deshalb wurde *FastCGI* entwickelt. Der wesentliche Vorteil besteht darin, daß für jede Anwendung lediglich einmal ein Prozeß gestartet wird. Dieser Prozeß lebt, solange der Webserver läuft. Ein anderer Vorteil ist die Art und Weise, wie der Anwendung die Anforderungsvariablen mitgeteilt werden. Während CGI die nötigen Daten als Umgebungsvariablen und über Pipes schickt, nutzt FastCGI eine einzige Vollduplexverbindung. Das hat nicht nur Geschwindigkeitsvorteile, sondern ermöglicht es auch, die FastCGI-Anwendung auf einem anderen Rechner als dem Webserver laufen zu lassen. Soll dies der Fall sein, dann wird eine TCP-Verbindung benutzt, andernfalls eine Vollduplex-Pipe. Im Grunde sind FastCGI-Anwendungen nichts anderes als Server.

## 1.4 Java

Bekannt wurde die Sprache Java 1996 durch kleine Programme, die sich in Webseiten einbinden lassen, sogenannte *Applets*. Mit ihnen begann der Siegeszug von Java: Keine Computerzeitschrift, keine Entwicklermesse, die nicht irgendwann Java zum Thema hatte. Der Hype beherrscht die Medien; Produkte für den Massenmarkt gibt es bis heute jedoch nur wenige. Zum großen Teil liegt das sicherlich auch daran, daß das erste *AWT (Abstract Windows Toolkit)*, die Klassenbibliothek zum Programmieren von Anwendungen mit grafischer Benutzeroberfläche (GUI), eher rudimentär ausgefallen ist. Mittlerweile hat Sun die Java-Foundation-Klassen nachgeschoben, die mit ihren Swing-Komponenten das API wesentlich verbessern.

Auch die geringe Ausführungsgeschwindigkeit der ersten Interpreter war sicherlich ein Hemmschuh. Inzwischen stehen jedoch gute JIT-Compiler (Just in Time) zur Verfügung, so daß, angesichts ständig wachsender Prozessorleistung, mangelnde Geschwindigkeit kein Argument mehr gegen den Einsatz von Java ist. Einen weiteren Geschwindigkeitsgewinn verspricht die in Java 2 völlig überarbeitete automatische Speicherverwaltung (Garbage Collection) sowie die Hotspot-Technologie von Sun.

Wenn sich also im Bereich der Java-Applikationen mit GUI bisher recht wenig getan hat, so gibt es um so mehr Applikationen, die sich insbesondere die eingebauten Netzwerk- und Multithreading-Fähigkeiten zunutze machen. Java ist gerade für Client/Server-Anwendungen und im besonderen Maße für Middleware-Technologien wie CORBA prädestiniert. Die wesentlichen Konzepte dafür sind Sockets und Threads.

### 1.4.1 Sockets

Um über ein IP-Netzwerk (Internet-Protokoll) kommunizieren zu können, benötigt man eine Beschreibung des Kommunikationsendpunktes. Im Falle von Java nennt man diesen Socket. Grundsätzlich stehen zwei Socketarten zur Verfügung: ein TCP-Socket

(Transmission Control Protocol) und ein UDP-Socket (User Datagram Protocol). Der wesentliche Unterschied zwischen den beiden Protokollen besteht in ihrer Verläßlichkeit.

TCP garantiert, daß die Datenpakete tatsächlich und sogar in der richtigen Reihenfolge ankommen. UDP hingegen stellt weder sicher, daß alle gesendeten Datenpakete (oder Datagramme) ankommen, noch, daß sie in der richtigen Reihenfolge eintreffen. Ein typisches Einsatzgebiet für UDP sind daher Echtzeitanwendungen wie zum Beispiel das Internetradio. Hier kommt es nicht so sehr darauf an, daß jedes einzelne Datenpaket ankommt, wichtiger ist, daß die meisten Pakete rechtzeitig ankommen. Da der Verwaltungsaufwand, der mit TCP einhergeht, wegfällt, gewinnt man ein wenig an Geschwindigkeit. Zum Übertragen von Dateien, bei denen es auf jedes Bit ankommt, ist dieses Verfahren weniger geeignet[1]. Hier kommt das verläßlichere TCP zum Zug.

In Java werden TCP-Sockets von der Klasse `java.net.Socket` abgebildet, UDP-Sockets von `java.net.DatagramSocket`.

Das Besondere an Java-Sockets ist ihre einfache Handhabung. Eine Verbindung mit dem Webserver von Yahoo läßt sich beispielsweise durch folgenden Einzeiler aufbauen:

```
Socket mySocket = new Socket ("www.yahoo.com", 80);
```

Als Parameter für den Konstruktor benötigen wir lediglich die Internet-Adresse des entfernten Rechners sowie den Port, an dem der Webserver auf eine Verbindung wartet. Per Konvention ist dies im Falle eines WWW-Servers der Port 80. Verschiedene Konstruktoren ermöglichen es, die Internet-Adresse entweder als Zeichenkette (`java.lang.String`) oder als Objekt vom Typ `java.net.InetAddress` anzugeben. Der Port ist in jedem Fall ein `int`.

Um über eine Instanz der Klasse `java.net.Socket` kommunizieren zu können, verfügt diese über die Methoden `getOutputStream()` und `getInputStream()`. Ist erst einmal eine Verbindung aufgebaut, kann ein Programm per `read()` und `write(int b)` aus den assoziierten Streams lesen und in sie schreiben, genau wie es das mit einer Datei tun würde.

Das folgende Beispielprogramm `SimpleWebClient` (Listing 1.2) nimmt als Startparameter die Adresse einer WWW-Seite entgegen, liest die Seite und schreibt sie in die Standardausgabe. Die Realisierung ist sehr einfach: Nach dem üblichen Parameterparsen wird ein Socket instantiiert. Mit dem assoziierten Output-Stream wird der entsprechende HTTP-Befehl abgesetzt. Über den Input-Stream wird die Antwort eingelesen und auf dem Bildschirm ausgegeben.

1. Obwohl es natürlich möglich ist, indem man selbst Fehlerkorrektur, erneutes Senden etc. implementiert. Aber wozu etwas neu erfinden, wenn es schon eine sehr gute Lösung mit Namen TCP gibt? Wenn Sie trotzdem sehen möchten, wie es gemacht wird, vgl. [Stevens 90], S. 465ff.

```
package de.webapp.Examples.HttpClient;

import java.net.Socket;
import java.io.OutputStream;
import java.io.InputStream;
import java.io.IOException;

public class SimpleWebClient {
public static void main(String args[]) {
   String host;
   String file;

   if (args.length != 1) {
      System.out.println("Nutzung : java SimpleWebClient <URL ohne Protokollnamen>");
      System.out.println("Beispiel: java SimpleWebClient www.yahoo.com/index.html");
      System.exit(0);
   }
   int delimiterIndex = args[0].indexOf('/');
   if (delimiterIndex == -1) {
      // falls kein '/' im Argument vorkommt,
      // annehmen, dass es sich nur um den
      // Hostnamen handelt, und als Datei
      // '/' anfordern
      host = args[0];
      file = new String("/");
   }
   else {
      // Host- und Dateinamen trennen.
      host = args[0].substring(0, delimiterIndex);
      file = args[0].substring(delimiterIndex);
   }
   try {
      // Verbindung zum Port 80 des Hosts herstellen.
      Socket socket = new Socket(host, 80);
      // OutputStream des Sockets besorgen.
      OutputStream out = socket.getOutputStream();
      // Befehl zusammenstellen.
      String command = new String("GET " + file + " HTTP/1.0\r\n\r\n");
      // HTTP-GET-Befehl schreiben.
      out.write(command.getBytes());
      // Sicherstellen, daß der Befehl tatsächlich
      // gesendet wird und nicht in einem
      // Zwischenspeicher liegenbleibt.
      out.flush();
      // InputStream des Sockets besorgen.
      InputStream in = socket.getInputStream();
      int aByte;
      // So lange aus dem InputStream lesen, bis
      // dessen Ende erreicht ist. Dabei jedes Zeichen
      // in die Standardausgabe schreiben.
      while ((aByte = in.read()) != -1) {
```

```
            System.out.write(aByte);
         }
         // Ressourcen freigeben.
         out.close();
         in.close();
         socket.close();
      }
      // Falls Fehler auftraten, diese in die
      // Standardausgabe ausgeben.
      catch (IOException ioe) {
         System.out.println(ioe.toString());
      }
   }
} // Ende der Klasse
```

*Listing 1.2: Die Klasse* `SimpleWebClient`

Nicht viel anders funktioniert übrigens das Programm Telnet. Der Unterschied besteht lediglich darin, daß bei Telnet alle Ein- und Ausgaben in die Standardein- und -ausgabe geleitet werden und es in der Lage ist, Terminal-Befehle zu interpretieren.

Wie oben gezeigt, eignet sich `SimpleWebClient` hervorragend dazu, eine Verbindung zu initiieren. Dazu benötigt man jedoch einen Kommunikationspartner. Im Beispiel ist dies ein beliebiger Webserver, der nur darauf wartet, daß ein Client versucht, eine Verbindung zu ihm aufzubauen. Glückt dies, nimmt er Befehle entgegen, führt diese aus und schickt das Ergebnis zurück.

Um dieses Verhalten in Java zu programmieren, benötigt man einen `java.net.ServerSokket`. Diesem weist man beim Instantiieren einen Port zu und ruft seine `accept()`-Methode auf. Das veranlaßt die `ServerSocket`-Instanz, auf eine Verbindungsanforderung zu warten. Kommt eine Verbindung zustande, gibt die `accept()`-Methode ein `Socket`-Objekt zurück. Wie alle `Socket`-Objekte verfügt auch dieses über die oben beschriebenen Eigenschaften. Ein in Java programmierter Webserver müßte sich also einen Eingabestrom vom gerade erlangten `Socket`-Objekt besorgen und versuchen, einen HTTP-Befehl zu lesen und diesen auszuführen. Dazu müßte er das Ergebnis in den Ausgabestrom des `Socket`-Objekts schreiben.

Natürlich ist das nicht alles, was einen Webserver ausmacht. Ein weiteres wesentliches Merkmal eines Webservers ist die Fähigkeit, mehrere Anfragen nahezu gleichzeitig bearbeiten zu können. Mehr dazu im nächsten Abschnitt.

### 1.4.2 Threads

Beim Programmieren von Netzwerkanwendungen kommt es häufig vor, daß man zwei oder mehr Aufgaben gleichzeitig erledigen möchte: beispielsweise ein Bild von einem Server laden (siehe `java.awt.Images`) – was recht lange dauern kann – und Benutzereingaben entgegennehmen. Wer beides umsetzen möchte, hat im wesentlichen vier Möglichkeiten. Sie können

- unübersichtliche Schleifenkonstrukte schreiben, in denen beides erledigt wird,
- in Ihr Programm eine ebenso unübersichtliche wie kryptische Interrupt-Steuerung oder ähnliches einbauen,
- einen Extraprozeß zum Laden des Bildes abspalten oder
- einen Thread zum Laden des Bildes abspalten.

Die Möglichkeiten Eins und Zwei können, sofern sie gut umgesetzt werden, die schnellsten sein. Sie sind aber eher unübersichtlich und deshalb schwierig zu erweitern. Ist die Ausführungszeit unkritisch, was beim Laden eines Bildes aus dem Netz der Fall sein dürfte, ist es sinnvoller, die Multitasking-Fähigkeiten des Betriebssystems zu nutzen. Genau das ist der Fall, wenn man einen Prozeß abspaltet. Im Grunde startet der Programmierer damit ein kleines Programm, das die gewünschte Aufgabe erfüllt und sich meldet, wenn es fertig ist. Bedient man sich nicht solcher Tricks wie geteilter Speicherräume, ist es etwas umständlich, das Bild dem aufrufenden Prozeß zukommen zu lassen. Zudem bedeutet ein Prozeßwechsel eine nicht unerhebliche Belastung für das Betriebssystem, da der gesamte Prozessorzustand des zu verlassenden Prozesses gesichert und der des zu startenden Prozesses geladen werden muß. Laufen zu viele Prozesse auf einem Rechner, ist der Rechner bald mehr mit der Verwaltung als mit der Ausführung der Prozesse beschäftigt.

Threads mildern all diese Probleme. Mit ihnen kann man Programmteile, die voneinander unabhängige Aufgaben erledigen sollen, in unterschiedlichen Codeblöcken (Threads) unterbringen. All diese Programmteile können quasi gleichzeitig ausgeführt werden, über ausgeklügelte Mechanismen miteinander kommunizieren und geteilte Speicherbereiche (bzw. Objekte oder Variablen) nutzen. Da alle Threads innerhalb eines Prozesses laufen, ist kein zeitaufwendiger Prozeßwechsel nötig. Threads bieten dem Programmierer die Möglichkeit, Multitasking innerhalb eines Programms zu verwirklichen, ähnlich wie Prozesse dies innerhalb eines Betriebssystems tun, jedoch ohne den für diese Anwendung überflüssigen Ballast. Oft werden Threads deshalb auch Leichtgewichtprozesse genannt.

Genau wie Sockets sind Threads keine Erfindung der Programmiersprache Java – nur lassen sich durch Java Threads einfach, effektiv und somit produktiv einsetzen. In Java wird ein neuer Thread erzeugt, indem man eine Instanz der Klasse `java.lang.Thread` erzeugt. Eine spezielle Funktionalität läßt sich einfach durch Vererbung dieser Basisklasse erreichen:

```
class MyApplication extends java.lang.Thread {
...
}
```

Die Aufgaben, die ein Thread erledigen soll, müssen in der Methode `public void run()` kodiert werden:

```
class MyApplication extends java.lang.Thread {
   ...
public void run() {
   ...
   }
}
```

Um einen Thread zu starten, ruft man einfach die Methode `public void start()` auf:

```
MyApplication myApp = new MyApplication();
myApp.start();
```

Innerhalb von `start()` wird automatisch die `run()`-Methode des Threads aufgerufen. Ein Thread ist dann beendet, wenn die `run()`-Methode vollständig ausgeführt worden ist. Enthält die `run()`-Methode eine Endlosschleife, terminiert der Thread erst, wenn die *Virtuelle Maschine (VM)* gestoppt wird.

Man kann einen Thread auch mit `stop()` stoppen. Diese Methode ist allerdings so brachial, daß sie in Java 2 mißbilligt (also »deprecated«) wurde. Das bedeutet, Sun rät von der Nutzung dieser Methode ab. Der Grund liegt darin, daß sämtliche Monitore, die zuvor von Objekten innerhalb der `run()`-Methode gehalten wurden, plötzlich aufgegeben werden. Dies führt dazu, daß Objekte, die sich in einer durch `synchronized` geschützten Transaktion befanden, diesen Schutz verlieren und in einem inkonsistenten Zustand für andere Objekte sichtbar werden.

#### *Thread-Zustände*

Grundsätzlich können sich Threads in einem von vier Zuständen befinden (Abbildung 1.1):

- neu
- lauffähig
- blockiert
- tot

Während ein neuer Thread ein Thread ist, der noch nicht gestartet wurde, und ein toter Thread einer, dessen `run()`-Methode terminierte, sind die beiden anderen Zustände nicht ganz so einfach zu erklären.

Man bezeichnet einen Thread als lauffähig, wenn er gestartet wurde. Lauffähig bedeutet dabei nicht, daß der Thread auch tatsächlich ausgeführt wird. Es ist Sache des Betriebssystems und der Virtuellen Maschine, einen Thread tatsächlich auszuführen. Auch, wenn er ausgeführt wird, befindet er sich immer noch im Zustand »lauffähig«. In diesem Zustand verharrt ein Thread, bis er entweder stirbt oder blockiert wird.

Eine Blockade kann dabei dreierlei Ursachen[2] haben:

- Die `wait()`-Methode wurde aufgerufen.
- Die `sleep()`-Methode wurde aufgerufen.
- Der Thread ruft eine Methode auf, die die Ein- oder Ausgabe blockiert.

Der wesentliche Unterschied zwischen `wait()` und `sleep()` besteht darin, daß bei `wait()` sämtliche Monitore abgegeben werden, während bei einem Aufruf von `sleep()` der schlafende Thread alle Monitore behält. Bei der Rückkehr beider Methoden beziehungsweise ihrer Unterbrechung wird der Thread wieder lauffähig. Gleiches gilt für den Aufruf einer blockierenden Ein-/Ausgabemethode.

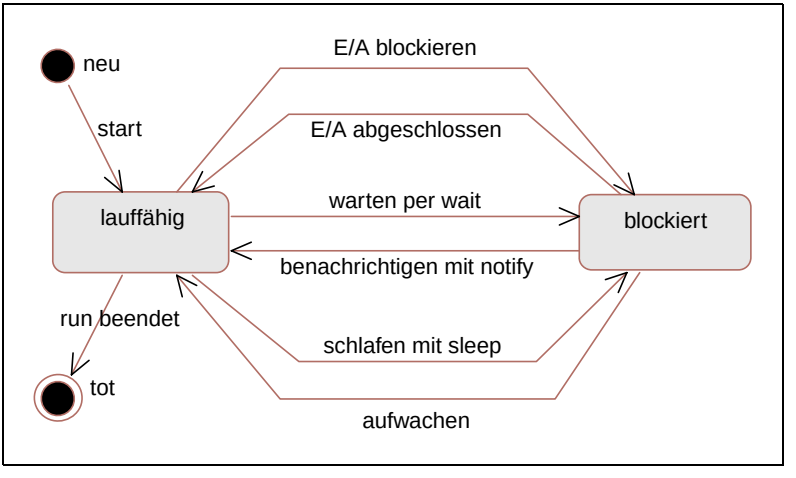


*Abbildung 1.1: Zustandsübergangsdiagramm der Klasse `java.lang.Thread`*

### *Runnable-Interface*

Gelegentlich möchte man Klassen wie einen Thread ausführen, kann sie aber nicht mehr von `Thread` ableiten, da sie schon von einer anderen Klasse erben. In einem solchen Fall sollte man die Schnittstelle `java.lang.Runnable` implementieren. Dazu muß zu der Klasse lediglich eine `run()`-Methode hinzugefügt werden.

```
class MyRunnableApplication extends BeliebigeKlasse
implements java.lang.Runnable {
   ...
public void run() {
```

2. In Java 2 wurden die Methoden `suspend()` und `resume()` wegen der ihnen innewohnenden Gefahr eines Deadlocks mißbilligt (*deprecated*). Daher gehen wir an dieser Stelle nicht näher auf sie ein.

```
   ...
   }
}
```

Um eine Instanz dieser Klasse wie einen Thread auszuführen, muß sie einem Thread im Konstruktor übergeben werden. Dieser wird anschließend wie jeder andere Thread gestartet. Der einzige Unterschied besteht darin, daß er statt der eigenen `run()`-Methode die des `Runnable`-Objekts aufruft.

```
class MyRunnableApplication extends BeliebigeKlasse
implements java.lang.Runnable {
public static void main (String args[]) {
   MyRunnableApplication myApp = new MyRunnableApplication();
   Thread myThread = new Thread(myApp);
   MyThread.start();
}
...
public void run() {
...
}
}
```

Damit sind die Grundlagen der Verwendung von Java-Threads erklärt. Wir wollen in diesem Buch nicht auf die speziellen Probleme der Entwicklung nebenläufiger Java-Programme eingehen, sondern werden an geeigneter Stelle lediglich einige Sonderfälle betrachten. Sollten Sie tiefer in die fraglos sehr interessante Materie eintauchen wollen, sei Ihnen das Buch von Doug Lea [Lea 96] empfohlen.

# 2 HTTP-Server

Im folgenden wollen wir einen einfachen HTTP-Server programmieren. Dabei wird es sich nicht um eine vollständige oder elegante Realisierung des HTTP handeln. Es geht uns in erster Linie darum, ein grundsätzliches Verständnis für die Struktur eines Servers sowie für HTTP zu schaffen.

## 2.1 Die Minimalimplementierung

Unser erster Server wird lediglich einen HTTP-Befehl verstehen, nämlich `GET`, und er wird ihn auch nur einmal ausführen können. Zugegeben, für einen Server ist das keine besondere Leistung, als erstes Beispiel ist es jedoch gut geeignet.

Nach dem Start horcht unser Server an dem vorgegebenen Port 8080, bis ein Client zu diesem eine Verbindung aufbaut. Natürlich horchen HTTP-Server normalerweise an Port 80. Dieser ist auf Unix-Systemen jedoch dem Systemadministrator vorbehalten. Damit Sie dieses Beispiel auch auf einem Unix-Rechner problemlos nachvollziehen können, haben wir Port 8080 gewählt.

Sie können den Server mit einem normalen Webbrowser wie Netscape Navigator, Microsoft Internet Explorer, Opera oder Lynx testen, indem Sie den URL `http://<hostname>:8080/` eingeben. Damit der Server auch etwas zurückschicken kann, sollte sich in Ihrem Arbeitsverzeichnis eine Datei mit dem Namen `index.html` befinden, da Anfragen relativ zum aktuellen Verzeichnis interpretiert werden.

Hat Ihr Browser mit dem Server eine Verbindung aufgebaut, gibt die Methode `accept()` einen Socket zurück. Von diesem Socket besorgen wir uns einen Eingabestrom, auf den wir über einen `Reader` zugreifen. Den Ausgabestrom des Sockets umgeben wir mit einem `BufferedOutputStream`, da in Java gepufferte Ströme Daten wesentlich effizienter übertragen. Anschließend lesen wir die erste Zeile aus dem Eingabestrom. In dieser Zeile sollte sich die Anforderung befinden. Damit wir nachvollziehen können, was geschieht, geben wir die Zeile aus.

Um die Anfrage auszuführen, teilen wir sie mit einem `java.util.StringTokenizer` in drei Teile: den Befehl, den URI und das verwendete Protokoll. Handelt es sich nicht um drei Tokens oder ist das erste Token kein `GET`-Befehl, geben wir eine Fehlermeldung aus und

brechen ab. Geht alles gut, lesen wir den Namen der angeforderten Datei aus Token Nummer zwei und hängen ihn an den Pfad des Dokumenten-Basisverzeichnisses (`docRoot`) unseres Servers. Da der angeforderte Dateiname gemäß RFC 2616 (`http://www.rfc-editor.org/rfc/rfc2616.txt`) immer mit einem Schrägstrich beginnt, ist das ohne Probleme möglich. Bleibt jetzt nichts oder nur ein Verzeichnisname übrig, hängen wir als Standardwert den Dateinamen »index.html« an. Die Methode `sendDocument()` schließlich kopiert die verlangte Datei in den Ausgabestrom des Sockets.

```
package de.webapp.Examples.HttpServer;

import java.io.*;
import java.net.*;
import java.util.*;

public class OneShotHttpd {

public final static int HTTP_PORT = 8080;

public static void main(String argv[]) {
   try {
      ServerSocket listen = new ServerSocket(HTTP_PORT);
      Socket client = listen.accept();
      BufferedReader is = new BufferedReader(
         new InputStreamReader(client.getInputStream()));
      DataOutputStream os = new DataOutputStream(client.getOutputStream());
      String request = is.readLine();
      System.out.println("Anfrage: " + request);
      StringTokenizer st = new StringTokenizer(request);
      if ((st.countTokens()==3) && st.nextToken().equals("GET")) {
         request = st.nextToken().substring(1);
         if (request.endsWith("/") || request.equals(""))
            request += "index.html";
         sendDocument(os, request);
      }
      else
         System.err.println ("400 Bad Request");
      is.close();
      os.close();
      client.close();
   }
   catch (IOException ioe) {
      System.err.println("Fehler: " + ioe.toString());
   }
}

public static void sendDocument(DataOutputStream out, String file)
      throws IOException {
   try {
      BufferedInputStream in = new BufferedInputStream (new FileInputStream(file));
      byte[] buf = new byte[1024];
```

```
      int len;
      while ((len = in.read(buf, 0, 1024)) != -1) {
         out.write(buf, 0, len);
      }
      in.close();
   }
   catch (FileNotFoundException fnfe) {
      System.err.println ("404 Not Found");
   }
}
} // Ende der Klasse
```

*Listing 2.1: Die Klasse* `OneShotHttpd`

Wenn Sie die Anfrage an `OneShotHttpd` (Listing 2.1) durch den Browser wiederholen, erscheint eine Fehlermeldung, daß der Server nicht gefunden wurde. Um diese Situation zu verbessern, wollen wir das Programm um die Fähigkeit erweitern, mehrere Anfragen bedienen zu können. Für diese Erweiterung sind lediglich minimale Änderungen erforderlich.

## 2.2 Simultanbedienung

Im Gegensatz zu `OneShotHttpd` erbt `SimpleHttpd` (Listing 2.2) von `java.lang.Thread` statt von `java.lang.Object`. In der statischen `main()`-Methode wird zwar immer noch am Port 8080 gehorcht, sobald aber eine Verbindung zustande kommt, wird `SimpleHttpd` mit dem erlangten Socket als Argument instantiiert. Wegen einer Endlosschleife wird dieser Vorgang bei jeder Anfrage erneut ausgeführt. Er dient einzig und allein zur Annahme der Verbindung. Die eigentliche Arbeit wird von den erzeugten Instanzen der Klasse `SimpleHttpd` erledigt, an die der Socket weitergereicht wird (Abbildung 2.1). Somit fungiert die statische `main`-Methode als Akzeptor und jede Instanz der Klasse `SimpleHttpd` als Handler.

Im Konstruktor wird zunächst der Instanzvariablen `s` der aktuelle Socket zugewiesen. Anschließend wird der `SimpleHttpd`-Thread mit `start()` gestartet. In der `run()`-Methode schließlich finden wir große Teile des Codes wieder, der sich vorher in der `main()`-Methode befand. Die Methode `sendDocument()` bleibt völlig gleich. Aus `OneShotHttpd` wird so `SimpleHttpd`.

```
package de.webapp.Examples.HttpServer;

import java.io.*;
import java.net.*;
import java.util.*;


public class SimpleHttpd extends Thread {
```

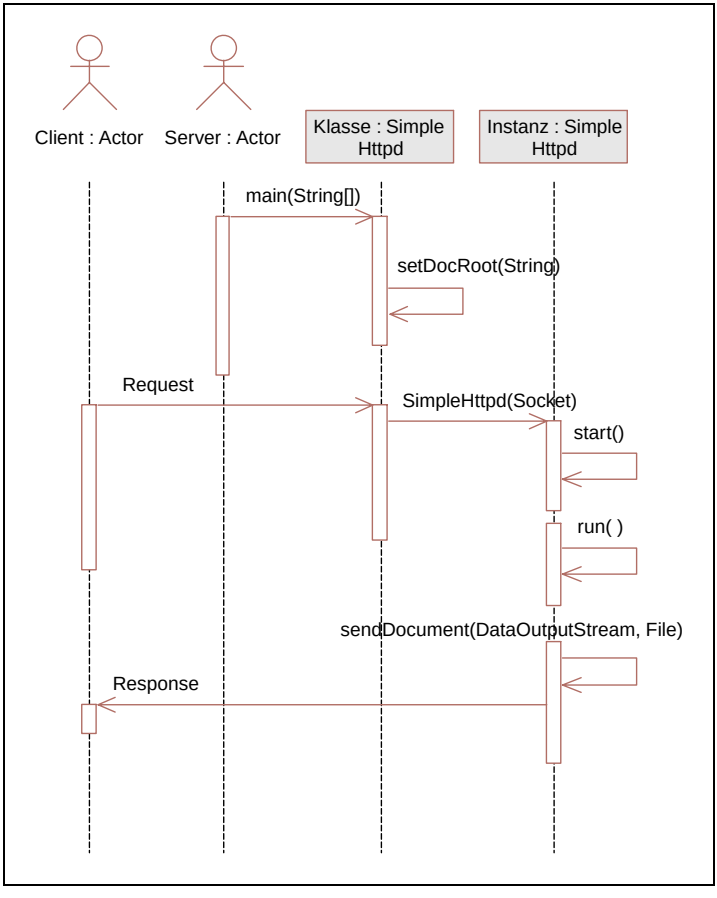


*Abbildung 2.1: Sequenzdiagramm von* `SimpleHttpd`

```
protected Socket s = null;
public final static int HTTP_PORT = 8080;

public static void main(String argv[]) {
   try {
      ServerSocket listen = new ServerSocket(HTTP_PORT);
      while (true) {
         SimpleHttpd aRequest = new SimpleHttpd(listen.accept());
      }
   }
   catch (IOException e) {
      System.err.println("Fehler: " + e.toString());
   }
}
public SimpleHttpd(Socket s) {
   this.s = s;
```

```
   start();
}

public void run() {
   try {
      BufferedReader is = new BufferedReader(
         new InputStreamReader(s.getInputStream()));
      DataOutputStream os = new DataOutputStream(s.getOutputStream());
      String request = is.readLine();
      System.out.println("Request: " + request);
      StringTokenizer st = new StringTokenizer(request);
      if ((st.countTokens()==3) && st.nextToken().equals("GET")) {
         request = st.nextToken().substring(1);
         if (request.endsWith("/") || request.equals(""))
            request += "index.html";
         sendDocument(os, request);
      }
      else
         os.write ("400 Bad Request\r\n".getBytes());
      is.close();
      os.close();
      s.close();
   }
   catch (IOException ioe) {
      System.err.println("Fehler: " + ioe.toString());
   }
}
protected void sendDocument(DataOutputStream out, String request)
      throws IOException {
   ...
}
} // Ende der Klasse
```

*Listing 2.2: Die Klasse* `SimpleHttpd`

## 2.3 Etwas mehr Sicherheit, bitte

Mit `SimpeHttpd` steht uns ein sehr einfacher, multithreaded Webserver zur Verfügung, der den `GET`-Befehl beherrscht. Er ist in der Lage, gleichzeitig und nacheinander beliebig viele Anfragen zu beantworten. Leider beantwortet er fast jede Anfrage – auch jene, die er besser unbeantwortet lassen sollte. Eine Anforderung folgenden Formats sollte besser zu einer Fehlermeldung führen:

```
http://<hostname>:8080/../../../../../etc/passwd
```

Sonst könnte man von jedem Webbrowser aus jede Datei des Rechners erreichen. Um dies zu verhindern, müssen wir sicherstellen, daß keine Datei ausgeliefert wird, die sich unterhalb des Dokumenten-Basisverzeichnisses befindet. Recht praktisch geht das, indem man sämtliche relativen Referenzen im Pfad der verlangten Datei (`/pfad/../`

`datei.dat` entspricht `/datei.dat`) auflöst und den absoluten Pfad sowohl vom Dokumenten-Basisverzeichnis als auch von der angeforderten Datei berechnet. Diese kanonischen Pfade sind pro Dateisystem eindeutig. Beginnt der Pfad der angeforderten Datei also nicht mit dem Pfad des Dokumenten-Basisverzeichnisses, handelt es sich um eine verbotene Anfrage. Daher müssen wir an unserem Server zwei kleine Änderungen vornehmen.

Da der kanonische Pfad des Dokumenten-Basisverzeichnisses immer gleich bleibt, berechnen wir ihn nur einmal und speichern ihn in einer Klassenvariablen. Nützlicherweise verfügt die Klasse `java.io.File` über die Methode `getCanonicalPath()`, die genau den Pfad berechnet, den wir benötigen.

```
...
protected static String canonicalDocRoot;
...
public static void main(String argv[]) {
   ...
      docRoot = new File(".");
      canonicalDocRoot = docRoot.getCanonicalPath();
      ServerSocket listen = new ServerSocket(HTTP_PORT);
   ...
}
```

Nun muß bei jedem Aufruf noch verglichen werden, ob der angeforderte Dateipfad mit dem kanonischen Dokumenten-Basisverzeichnis beginnt. Dies wird in der `run()`-Methode erledigt:

```
public void run() {
   ...
         if (filename.endsWith("/") || filename.equals(""))
            filename += "index.html";
         File file = new File(filename);
         if (file.getCanonicalPath().startsWith(canonicalDocRoot))
            sendDocument(os, file);
         else
            System.err.println("403 Forbidden");
   ...
}
```

Wird nun eine Datei unterhalb des Dokumenten-Basisverzeichnisses angefordert, schreibt der Server eine Fehlermeldung in die Standardfehlerausgabe (stderr).

## 2.4 Statuscodes

Unser Server funktioniert jetzt schon recht gut – leider ist er aber noch weit davon entfernt, HTTP-konform zu sein. Das Protokoll sieht nämlich vor, daß die erste Zeile jeder Antwort die HTTP-Version, einen Statuscode und eine kurze Meldung (Reason-

Phrase) enthält. Die drei Teile müssen jeweils durch Leerzeichen getrennt sein. Außerdem sollten noch ein Wagenrücklauf- und ein Zeilenvorschubzeichen (CRLF) angehängt werden, zum Beispiel:

```
HTTP/1.0 200 OK\r\n
```

Der Einfachheit halber schreiben wir die Protokollversion in eine Klassenvariable, ebenso legen wir einige Konstanten für die Statuscodes an. Außerdem ist es sinnvoll, den aktuellen Statuscode in einer Instanzvariablen (`statusCode`) zu halten, die über entsprechende `get`- und `set`-Methoden angesprochen werden kann. Auf diese Weise können wir `statusCode` mit der Konstante `SC_OK`, `"200 OK"` initialisieren und im Falle einer Änderung des Codes einfach mit `setStatusCode(String)` überschreiben.

Die Methode `sendStatusLine(DataOutputStream)` sendet den Protokollnamen, den aktuellen Statuscode und hängt noch die Konstante `CRLF` (`"\r\n"`) an.

```
public final static String CRLF = "\r\n";
public final static String PROTOCOL = "HTTP/1.0 ";
public final static String SC_OK = "200 OK";
public final static String SC_BAD_REQUEST = "400 Bad Request";
public final static String SC_FORBIDDEN = "403 Forbidden";
public final static String SC_NOT_FOUND = "404 Not Found";
protected String statusCode = SC_OK;

protected void setStatusCode(String statusCode) {
   this.statusCode = statusCode;
}

protected String getStatusCode() {
   return statusCode;
}

protected void sendStatusLine(DataOutputStream out)
      throws IOException {
   out.writeBytes(PROTOCOL + getStatusCode() + CRLF);
}
```

Zum Abschluß müssen wir nur noch einen entsprechenden Aufruf in der Methode `sendDocument` unterbringen:

```
protected void sendDocument(DataOutputStream os, File file)
      throws IOException {
   try {
      BufferedInputStream in = new BufferedInputStream (new FileInputStream(file));
      sendStatusLine(os);
      out.writeBytes(CRLF);
      byte[] buf = new byte[1024];
      ...
}
```

Bewußt rufen wir `sendStatusLine` erst nach dem Öffnen der Datei auf, damit im Fehlerfall auch noch ein anderer Statuscode als `SC_OK` gesendet werden kann. Anschließend schreiben wir `CRLF` in den `OutputStream`, da laut HTTP-Spezifikation zwischen Kopf und Rumpf der Antwort ein extra `CRLF` stehen muß.

Mit den nun vorhandenen Statuscodes ist es leicht, die Fehlerbehandlung zu verbessern. Schließlich will der Client auch von Fehlern unterrichtet werden. Zu diesem Zweck schreiben wir die Methode `sendError`.

```
protected void sendError(String statusCode, DataOutputStream out)
      throws IOException {
   setStatusCode(statusCode);
   sendStatusLine(out);
   out.writeBytes(
         CRLF
      + "<html>"
      + "<head><title>" + getStatusCode() + "</title></head>"
      + "<body><h1>" + getStatusCode() + "</h1></body>"
      + "</html>"
   );
   System.err.println(getStatusCode());
}
```

Jetzt müssen noch die `System.err.println`-Aufrufe durch entsprechende `sendError`-Aufrufe ersetzt werden.

## 2.5 Kopfdaten

Neben der Statuszeile können und sollten im Kopf der Antwort noch zusätzliche Informationen an den Client zurückgesandt werden. Zum einen sind das allgemeine Daten wie der Name des Servers und die aktuelle Zeit, zum anderen Daten über das angeforderte Objekt wie dessen Größe oder letzte Änderung. In jedem Fall aber handelt es sich um Schlüssel-Wert-Paare. Eine Hashtabelle drängt sich daher als Datenstruktur geradezu auf. Neben der Datenstruktur benötigen wir noch Methoden, um auf diese zuzugreifen, sowie eine Methode, die sämtliche Kopfdaten in der korrekten Form ausgibt. In unserem Server übernehmen diese Aufgaben die `getHeader()`- und `setHeader()`-Methoden sowie `sendHeader()`.

```
protected Hashtable myHeaders = new Hashtable();
protected void setHeader(String key, String value) {
   myHeaders.put(key, value);
}
protected String getHeader(String key) {
   return (String)myHeaders.get(key);
}

protected void sendHeader(DataOutputStream out)
```

```
      throws IOException {
   String line;
   String key;
   Enumeration e = myHeaders.keys();
   while (e.hasMoreElements()) {
      key = (String)e.nextElement();
      out.writeBytes(key + ": " + myHeaders.get(key) + CRLF);
   }
}
```

Die Kopfdaten müssen an den Client geschickt werden, bevor das angeforderte Objekt gesendet wird. Daher erfolgt der Aufruf von `sendHeader()` vor dem Schreiben der Datei:

```
protected void sendDocument(DataOutputStream os, File file)
      throws IOException {
   try {
      BufferedInputStream in = new BufferedInputStream (new FileInputStream(file));
      sendStatusLine(os);
      setHeader("Content-Length", (new Long(file.length())).toString());
      setHeader("Content-Type", guessType(file.getPath));
      ...
      sendHeader(os);
      os.writeBytes(CRLF);
      ...
}

protected static Properties typeMap = new Properties();

public String guessType(String filename) {
   String type = null;
   int i = filename.lastIndexOf(".");
   if (i > 0)
      type = typeMap.getProperty(filename.substring(i));
   if (type == null)
      type = "unknown/unknown";
   return type;
}
```

Einer der wichtigsten Headerwerte ist der Content-Type, da er dem Browser bei der Entscheidung hilft, wie ein Objekt angezeigt werden soll. HTML-Objekte werden anders dargestellt als PDF-Dateien. Im Content-Type ist daher der MIME-Typ der angeforderten Daten kodiert. Zumeist läßt sich dieser aus der Endung des Dateinamens erraten. Genau das tun wir mit der Methode `guessType(String)`, in der wir das Suffix als Schlüssel zu einem `java.util.Properties`-Objekt benutzen. Dieses `Properties`-Objekt sollte zuvor in der `main`-Methode geladen werden. Es muß Schlüssel-Wert-Paare folgenden Formats beinhalten:

```
.html=text/html
.htm=text/html
.txt=text/plain
```

```
.gif=image/gif
.jpeg=image/jpeg
.jpg=image/jpeg
...
```

Durch die Entwicklung dieses einfachen Servers sind die Grundlagen des Hypertext-Transfer-Protokolls sicherlich klar geworden. Der Server erfüllt ein paar Funktionen mehr, als in HTTP 0.9 definiert sind, ist jedoch von der Umsetzung des HTTP 1.0 noch weit entfernt. Tatsächlich kann er nur Dateien ausliefern; alle anderen HTTP-Befehle werden ignoriert. Im nächsten Kapitel werden wir uns damit beschäftigen, wie man mit Servlets die Funktionalität eines Servers weit über das bloße Abliefern von Dateien hinaus erweitern kann.

Um noch einmal einen kompletten Überblick über den Server zu geben, folgt hier das gesamte Listing:

```
package de.webapp.Examples.HttpServer;

import java.io.*;
import java.net.*;
import java.util.*;

public class SimpleHttpd2 extends Thread {

protected Socket s = null;
protected static File docRoot;
protected static String canonicalDocRoot;

public final static int HTTP_PORT = 8080;
public final static String CRLF = "\r\n";
public final static String PROTOCOL = "HTTP/1.0 ";

public final static String SC_OK = "200 OK";
public final static String SC_BAD_REQUEST = "400 Bad Request";
public final static String SC_FORBIDDEN = "403 Forbidden";
public final static String SC_NOT_FOUND = "404 Not Found";

protected static Properties typeMap = new Properties();
protected String statusCode = SC_OK;
protected Hashtable myHeaders = new Hashtable();

public static void main(String argv[]) {
   try {
      typeMap.load(new FileInputStream("mime.types"));
      docRoot = new File(".");
      canonicalDocRoot = docRoot.getCanonicalPath();
      ServerSocket listen = new ServerSocket(HTTP_PORT);
      while (true) {
         SimpleHttpd2 aRequest = new SimpleHttpd2(listen.accept());
      }
```

```
   }
   catch (IOException e) {
      System.err.println("Fehler: " + e.toString());
   }
}

public SimpleHttpd2(Socket s) {
   this.s = s;
   start();
}

public void run() {
   try {
      setHeader("Server", "SimpleHttpd2");
      BufferedReader is = new BufferedReader(
         new InputStreamReader(s.getInputStream()));
      DataOutputStream os = new DataOutputStream(s.getOutputStream());
      String request = is.readLine();
      System.out.println("Request: " + request);
      StringTokenizer st = new StringTokenizer(request);
      if ((st.countTokens()==3) && st.nextToken().equals("GET")) {
         String filename = docRoot.getPath() + st.nextToken();
         if (filename.endsWith("/") || filename.equals(""))
            filename += "index.html";
         File file = new File(filename);
         if (file.getCanonicalPath().startsWith(canonicalDocRoot))
            sendDocument(os, file);
         else
            sendError(SC_FORBIDDEN, os);
      }
      else {
         sendError(SC_BAD_REQUEST, os);
      }
      is.close();
      os.close();
      s.close();
   }
   catch (IOException ioe) {
      System.err.println("Fehler: " + ioe.toString());
   }
}

protected void sendDocument(DataOutputStream os, File file)
      throws IOException {
   try {
      BufferedInputStream in = new BufferedInputStream (new FileInputStream(file));
      sendStatusLine(os);
      setHeader("Content-Length", (new Long(file.length())).toString());
      setHeader("Content-Type", guessType(file.getPath()));
      sendHeader(os);
      os.writeBytes(CRLF);
```

```
      byte[] buf = new byte[1024];
      int len;
      while ((len = in.read(buf, 0, 1024)) != -1) {
         os.write(buf, 0, len);
      }
      in.close();
   }
   catch (FileNotFoundException fnfe) {
      sendError(SC_NOT_FOUND, os);
   }
}

protected void setStatusCode(String statusCode) {
   this.statusCode = statusCode;
}

protected String getStatusCode() {
   return statusCode;
}

protected void sendStatusLine(DataOutputStream out)
      throws IOException {
   out.writeBytes(PROTOCOL + getStatusCode() + CRLF);
}

protected void setHeader(String key, String value) {
   myHeaders.put(key, value);
}

protected void sendHeader(DataOutputStream out)
      throws IOException {
   String line;
   String key;
   Enumeration e = myHeaders.keys();
   while (e.hasMoreElements()) {
      key = (String)e.nextElement();
      out.writeBytes(key + ": " + myHeaders.get(key) + CRLF);
   }
}

protected void sendError(String statusCode, DataOutputStream out)
      throws IOException {
   setStatusCode(statusCode);
   sendStatusLine(out);
   out.writeBytes(
      CRLF
      + "<html>"
      + "<head><title>" + getStatusCode() + "</title></head>"
      + "<body><h1>" + getStatusCode() + "</h1></body>"
      + "</html>"
   );
```

```
   System.err.println(getStatusCode());
}

public String guessType(String filename) {
   String type = null;
   int i = filename.lastIndexOf(".");
   if (i > 0)
      type = typeMap.getProperty(filename.substring(i));
   if (type == null)
      type = "unknown/unknown";
   return type;
}

} // Ende der Klasse
```

*Listing 2.3: Die Klasse* `SimpleHttpd2`

# 3 Servlets

Da die Dienste konventioneller Webserver größtenteils auf das Ausliefern von Dateien beschränkt sind, benötigt man zur Realisierung darüber hinausgehender Anwendungen eine leistungsfähige Schnittstelle zwischen der eigentlichen Anwendung und dem HTTP-Server. Im wesentlichen gibt es dazu drei verschiedene Ansätze:

- Common Gateway Interface (CGI): Die vom *NCSA* (*National Center for Supercomputing Applications*) entwickelte Schnittstelle funktioniert nach einem simplen Schema. Ein Client fordert ein Objekt vom Webserver an. Dieser startet ein Programm als externen Prozeß und schickt dessen Ausgabe an den Client zurück (Kapitel 1.3).
- Proprietäres Server-API: Netscape und Microsoft bieten mit *NSAPI (Netscape Server Application Programming Interface)* und *ISAPI (Internet Server Application Programming Interface)* zu ihren Webservern jeweils ein eigenes API an. Über diese Programmierschnittstelle kann man Programme direkt in den Server einbinden. Sie werden beim Hochfahren des Servers initialisiert. Da sie zumeist im selben Prozeß wie der Server laufen, entfallen teure Prozeßwechsel zur Ausführungszeit. Im Vergleich zu CGI führt dies zu einem erheblichen Geschwindigkeitsvorteil. Leider hat diese Technik auch einen großen Nachteil: Wenn auch nur ein einziges eingebundenes Programm abstürzt, reißt es den gesamten Server mit in den Abgrund. Microsoft hat dieses Problem erkannt und bietet mit dem *Internet Information Server (IIS)* 4.0 die Möglichkeit, webbasierte Applikationen in separaten Prozessen auszuführen.
- Servlet-API: Sun entwickelte diese Anwendungsprogrammierschnittstelle, um serverseitigen Java-Programmen eine einfache und flexible Grundlage zu geben. Über das API kann der Programmierer ähnlich wie in CGI auf Umgebungsvariablen zugreifen und in einen Strom die Antwort (Response) auf die Anfrage (Request) schreiben. Darüber hinaus bietet das API Cookie-Unterstützung und Session-Management. Dabei wird offengelassen, ob die serverseitige Applikation in demselben oder einem anderen Prozeß als der Server läuft. Will man die Prozesse trennen, kann man das in jedem Fall leicht mit CORBA, RMI oder der Einbindung von *Enterprise Java Beans (EJB)* erreichen.

In diesem Kapitel werden wir uns intensiv der Servlet-Programmierung widmen. Die notwendigen Klassen sind in den Paketen `javax.servlet` und `javax.servlet.http` enthal-

ten, die sich als *Java Servlet Development Kit (JSDK)* von Suns Webserver herunterladen lassen (`http://java.sun.com/products/servlet/index.html`). Ebenfalls dort befindet sich das *JavaServer Web Development Kit (JSWDK)*, das zur Drucklegung dieses Buches in der Version 1.0 vorlag. Anders als das JSDK beinhaltet das JSWDK zusätzlich eine JavaServer-Pages-Implementierung (JSP) und stellt somit eine komplette Entwicklungsplattform dar.

## 3.1 Grundlagen

Zunächst einmal wollen wir uns den Kommunikationsprozeß zwischen dem Nutzer und der Webapplikation verdeutlichen (Abbildung 3.1). Wenn ein Nutzer im WWW surft, klickt er mit der Maus auf Links. Dies veranlaßt den Browser, eine Anfrage (Request) an einen Server zu schicken. Der *Uniform Resource Locator (URL)*, also die Adresse des verlangten Objekts, ist dabei im HTML-Code enthalten:

```
<a href="http://www.meineSite.de/eineDatei.html">Link</a>
```

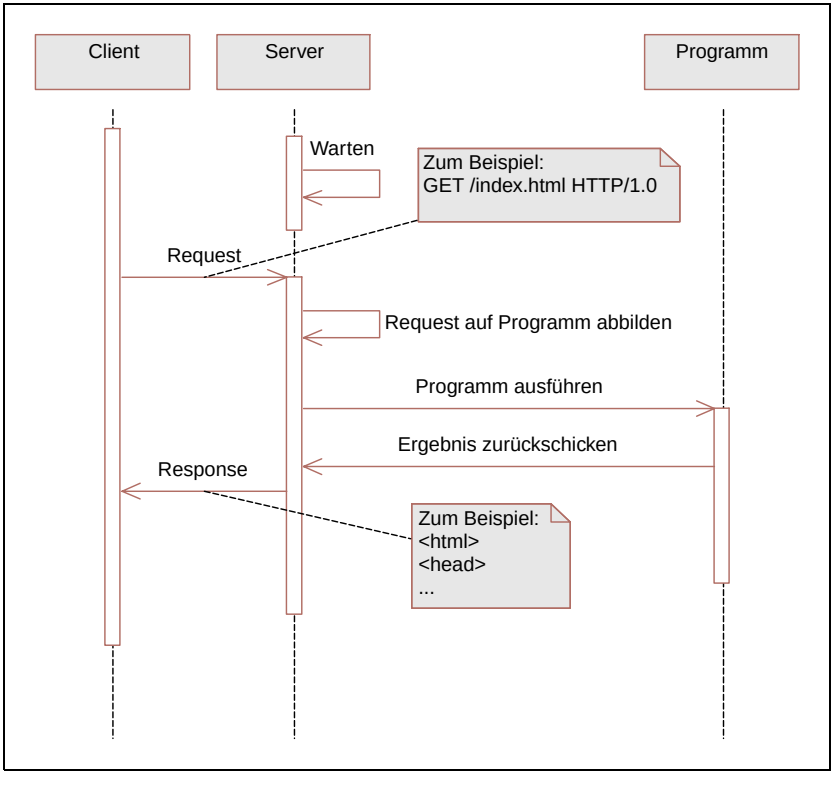


*Abbildung 3.1: Interaktion zwischen Client und Server*

Wenn auf einen solchen Link geklickt wird, sendet der Browser einen HTTP-`GET`-Befehl an den Server, um das verlangte Objekt vom Server in den Browser zu laden und anzuzeigen. Soviel zur Sicht des Clients.

Für den Server stellt sich der Vorgang wie folgt dar: Ein Client baut eine Verbindung mit dem Port auf, an dem der Server auf HTTP-Befehle wartet. Der Server liest den Befehl, versucht, ihn auszuführen, und schickt entweder das Ergebnis oder, im Falle eines Fehlers, eine entsprechende Fehlermeldung zurück. Sämtliche Meldungen sind in der HTTP-Spezifikation RFC 2616 (`http://www.rfc-editor.org/rfc/rfc2616.txt`) definiert.

Interessant für uns ist nur die Befehlsausführung. Handelt es sich bei dem per `GET` verlangten Objekt um eine Datei, muß der Server diese lediglich lesen und – mit entsprechenden Kopfdaten (einem Header) versehen – an den Browser schicken. Verbirgt sich hinter dem URL jedoch ein Programm, so muß der Server dieses ausführen und seine Ausgaben an den Client senden. Falls eine Servlet-Engine im Server enthalten ist, bietet diese die Möglichkeit, eigenen Java-Code an genau dieser Stelle einzuklinken.

## 3.2 Servlet-Engines

Mittlerweile gibt es fast zu jedem Server eine passende Servlet-Engine. Um die meisten Beispiele in diesem Buch nachvollziehen zu können, reicht die im WebApp-Framework (`http://www.webapp.de/`) enthaltene Engine jo! aus. Möchten Sie eine Engine in Ihren bereits laufenden Webserver einklinken, empfehlen wir Ihnen die Engines JServ der Apache Group (`http://www.apache.org`, `http://java.apache.org/`) und JRun von Allaire (`http://www.allaire.com/`). Beide sind im Netz verfügbar und werden ständig weiterentwickelt. Während JServ nur zusammen mit dem Apache-Webserver funktioniert, ist JRun für eine ganze Reihe von Servern verfügbar. Dazu zählen ebenfalls der Apache-Webserver (NT und Unix), Microsoft Personal Webserver 4.x, Microsoft Internet Information Server 3.x/4.x, Netscape FastTrack und Enterprise Server 3.x (NT und Unix) sowie der StarNine WebSTAR 3.x (Mac).

Neben diesen beiden Engines gibt es noch eine Reihe anderer kommerzieller und freier Engines beziehungsweise Server, die eine Engine beinhalten, deren genaue Eigenschaften wir hier aber nicht aufführen wollen. Um die wichtigsten zu nennen, hier eine kurze Liste:

- Java Webserver von JavaSoft (`http://jserv.java.sun.com/products/webserver/`)
- Jigsaw vom World Wide Web Consortium (`http://www.w3.org/Jigsaw/`)
- Websphere (früher ServletExpress) von IBM (`http://www.software.ibm.com/webservers/`)
- WebExpress von WebLogic (`http://weblogic.beasys.com/`)

- ServletExec von Unify (`http://www.unify.com/`)
- Jetty von Mortbay (`http://www.mortbay.com/software/Jetty.html`)

Eine ständig aktualisierte Liste befindet sich auf der Website von Sun unter der Adresse `http://java.sun.com/products/servlet/runners.html`.

## 3.3 Hello World

Die folgende Wendung ist zwar abgenutzt und arg strapaziert, und die Servlet-Technologie ist noch nicht einmal eine neue Programmiersprache, sondern nur ein API – dennoch wollen wir auf den Klassiker zum Einstieg nicht verzichten: Unser erstes Servlet soll »Hello World« im Browser des Nutzers anzeigen.

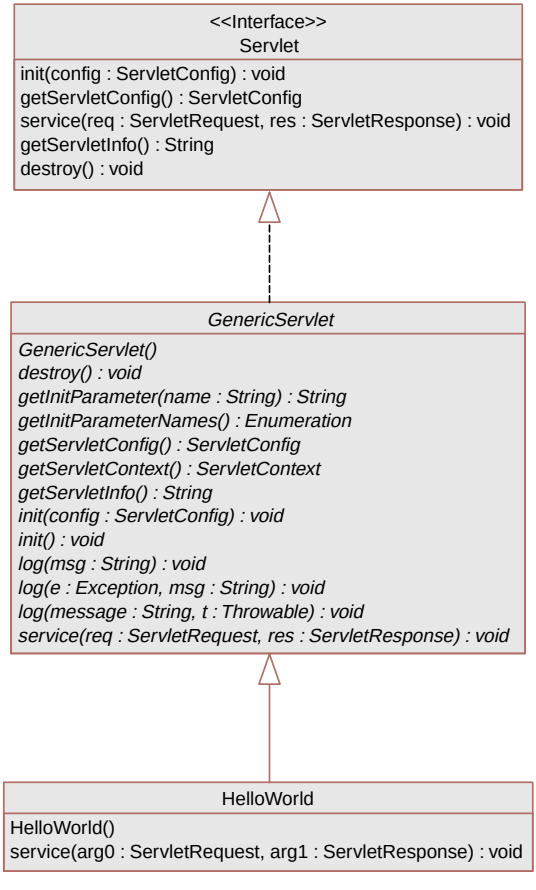


*Abbildung 3.2: `HelloWorld` erbt von `GenericServlet`, das wiederum die Schnittstelle `Servlet` implementiert.*

Wie Sie in Abbildung 3.2 sehen können, erbt unsere Klasse `HelloWorld` von `javax.servlet.GenericServlet`. Das ist nicht unbedingt notwendig, aber bequem. Servlets werden nicht durch eine Oberklasse, sondern durch die Schnittstelle `javax.servlet.Servlet` definiert. Diese Schnittstelle fordert, daß sich nur eine Klasse, die die Methoden `init(ServletConfig)`, `service(ServletRequest, ServletResponse)`, `destroy()`, `getServletConfig()` und `getServletInfo()` besitzt, Servlet nennen darf. `GenericServlet` realisiert all diese Methoden und noch ein paar mehr. Es liegt also nahe, von dieser Klasse zu erben und, falls notwendig, Methoden zu überschreiben. Das ist zumindest für die `service()`-Methode der Fall, da sie in `GenericServlet` als `abstract` deklariert worden ist. Um unser erstes Beispiel nicht unnötig kompliziert zu machen, haben wir uns in `HelloWorld` darauf beschränkt, genau diese Methode zu überschreiben.

### 3.3.1 Das Herz jedes Servlets: die service()-Methode

Jedesmal, wenn beim Server eine Anfrage für ein Servlet eintrifft, veranlaßt dieser die Servlet-Engine dazu, die `service()`-Methode des Servlets aufzurufen. Logischerweise ist diese Methode deshalb der ideale Ort, um unseren Gruß an die Welt loszuwerden. Behilflich ist uns dabei das Objekt `ServletResponse`, das der Methode beim Aufruf von der Engine übergeben wird. `ServletResponse` verfügt neben anderen über die zentralen Methoden `setContentType(String)` und `getOutputStream()`.

Da wir »Hello World« HTML-kodiert in einem Browser darstellen wollen, müssen wir uns an die Spielregeln halten und dem Browser mitteilen, daß es sich bei unserer Antwort um HTML handelt. Dies geschieht, indem wir den MIME-Typ der Antwort mittels `setContentType("text/html")` gleich »text/html« setzen. Das veranlaßt die Servlet-Engine dazu, im Kopf die Zeile »Content-type: text/html« einzufügen. Da der Kopf als erstes zum Browser gesendet wird, sollte `setContentType()` aufgerufen werden, bevor irgendeine Ausgabe erfolgt. Von dieser Regel dürfen Sie nur abweichen, wenn die von Ihnen benutzte Engine die Ausgabe zunächst in einen Zwischenspeicher schreibt, bevor sie die Daten tatsächlich an den Client übermittelt. Doch dazu mehr im Kapitel 3.5.4.

Um überhaupt etwas ausgeben zu können, müssen wir uns einen `OutputStream` vom `ServletResponse`-Objekt besorgen. Dies geschieht mit der Methode `getOutputStream()`. Bei dem Strom, den wir erhalten, handelt es sich jedoch nicht um einen normalen `java.io.OutputStream`, sondern um einen Abkömmling, einen `javax.servlet.ServletOutputStream`.

Die ohnehin abstrakte Klasse `OutputStream` wurde dahingehend erweitert, daß `ServletOutputStream` für primitive Datentypen (`boolean`, `char`, `double`, `float`, `int` und `long`) sowie `String`-Objekte jeweils `print`- und `println`-Methoden bereitstellt, die den Umgang mit dem Strom erheblich vereinfachen. Insbesondere wird der resultierende HTML-Code durch die beim Aufruf von `println` angefügten Wagenrücklauf- und Zeilenvorschubzeichen (Carriage Return und Line Feed, `CRLF`) leichter lesbar.

Nachdem wir also einen `ServletOutputStream` erhalten haben, rufen wir seine `println`-Methode auf und drucken »Hello World«.

Wie aus Listing 3.1 ersichtlich, verzichten wir darauf, einen möglichen Puffer des Stroms explizit zu leeren (engl.: flush) sowie den Strom zu schließen. Das wäre zwar möglich, ist jedoch nicht nötig, da die Servlet-Engine dies für uns erledigt.

```
package de.webapp.Examples.Servlet;
import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletResponse;
import javax.servlet.ServletRequest;
import javax.servlet.ServletOutputStream;
import java.io.IOException;
public class HelloWorld extends javax.servlet.GenericServlet {
public void service(ServletRequest req, ServletResponse res)
      throws ServletException, IOException {
   // MIME-Typ gleich "text/html" setzen.
   res.setContentType("text/html");
   // ServletOutputStream des ServletResponse-Objekts
   // besorgen.
   ServletOutputStream out = res.getOutputStream();
   // "Hello World" in den ServletOutputStream schreiben.
   out.println("Hello World");
}
} // Ende der Klasse
```

*Listing 3.1: Hello-World-Servlet*

### 3.3.2 Übersetzen, installieren, testen

Zum schnellen Testen von Servlets enthält das JSDK 2.1 einen einfachen servletfähigen Server. Er kann über die Datei `default.cfg` konfiguriert werden. Die entsprechenden Parameter können Sie Tabelle 3.1 entnehmen.

| Parameter | Bedeutung |
|---|---|
| `server.port` | Port, den der Server benutzen soll. Voreingestellt ist 8080. |
| `server.docbase` | Verzeichnis, in dem sich die Dateien und Servlets befinden. Dieser Parameter kann relativ zum aktuellen Verzeichnis oder als komplette URL angegeben werden. |
| `server.hostname` | Name des Servers. Dieser Parameter ist optional. |
| `server.inet` | Internet-Adresse des Servers. Dieser Parameter ist optional. |
| `server.tempdir` | Verzeichnis für temporäre Dateien. |
| `server.webapp.<webapp-Name>.mapping` | Definiert die Abbildung eines URI auf eine Web-Applikation. Beginnt der URI eines Requests mit diesem Parameter, wird der Request der angegebenen Web-Applikation zugeordnet. |
| `server.webapp.<webapp-Name>.docbase` | Basis-Verzeichnis einer Web-Applikation. Hier kann ein Verzeichnis relativ zum aktuellen Pfad oder eine komplette URL angegeben werden. |

*Tabelle 3.1: Konfigurationsparameter für den Server des JSDK 2.1*

Vor dem Start der Engine müssen wir noch ein paar Dinge erledigen. Zunächst einmal wollen wir `HelloWorld` übersetzen lassen. Stellen Sie dazu sicher, daß sich die jar-Datei `servlet.jar` im Klassenpfad befindet. Ist dies der Fall, lassen sich Servlets wie alle anderen Klassen auch mit dem Java-Compiler `javac` übersetzen, so auch `HelloWorld`.

Zum Start der Engine müssen nun noch mindestens zwei Konfigurationsdateien bearbeitet werden. In `servlets.properties` wird das Servlet bei der Engine registriert. In `mappings.properties` wird die Abbildung von URIs auf Servlets definiert. Beide Dateien befinden sich im Unterverzeichnis `/webpages/WEB-INF/` des JSDK beziehungsweise JSWDK.

In `servlets.properties` müssen Klassennamen und bei Bedarf Initialisierungsargumente aller Servlets angegeben werden, die von der Engine gestartet werden sollen (Listing 3.2). Dazu wird jedem Servlet ein Name zugeordnet, der innerhalb der Datei `servlets.properties` eindeutig sein muß. Er dient bei einigen Engines zudem als Teil des URL, unter dem das Servlet nach dem Start der Engine aufgerufen werden kann.

```
# "servlets.properties"-Beispieldatei
#
# <Name>.code=<Klassenname>(Klasse oder Klasse.class)
# <Name>.initparams=durch Kommata
#    getrennte Liste von <Name=Wert>-Paaren,
#    auf die vom Servlet durch
#    API-Aufrufe zugegriffen werden kann.
#

# HelloWorld-Servlet
HelloWorld.code=de.webapp.Examples.Servlet.HelloWorld

# Anderes Servlet
Anders.code=AnderesServlet
Anders.initArgs=argument1=arg1, argument2=arg2
```

*Listing 3.2: `servlets.properties`-Beispieldatei*

Nachdem wir unser Servlet registriert haben, müssen wir noch die Abbildung eines URI auf unser Servlet festlegen. Dazu dient die Datei `mappings.properties`. Sie enthält Zuordnungen von Dateierweiterungen wie beispielsweise `.jsp` zu einem zuvor in `servlets.properties` definierten Servlet (Listing 3.3).

```
# "mappings.properties"-Beispieldatei
#
# <Pfad>=<servletname>
# <Datei-Erweiterung>=<servletname>
#

# ordne /HelloWorld dem Servlet HelloWorld zu
```

```
/HelloWorld=HelloWorld
# ordne URIs mit der Endung .jsp dem Servlet JSPServlet zu
.jsp=JSPServlet
```

*Listing 3.3: `mappings.properties`-Beispieldatei*

Nachdem wir alle wichtigen Parameter definiert haben, können wir nun den Server mit dem Befehl `startserver` starten.

Das `HelloWorld`-Servlet läßt sich nun unter dem URL `http://<ServerAdresse>:8080/HelloWorld` mit einem beliebigen WWW-Browser aufrufen. `<ServerAdresse>` muß dabei durch den Namen oder die IP-Adresse des Rechners ersetzt werden, auf dem der Server gestartet wurde.

Um die Engine des WebApp-Frameworks jo! zu starten, muß zuvor eventuell die Datei `/webapp/projects/jo/etc/server.cfg` angepaßt werden. Tragen Sie hier den Port ein, unter dem jo! erreichbar sein soll. Beachten Sie, daß bei Unix-Systemen der Port 80 den Administratoren vorbehalten ist.

Die Datei `servlet.properties` folgt größtenteils dem Aufbau der gleichnamigen Datei des JSWDK. Beachten Sie, daß hier die Abbildung von URIs auf Servlets ebenfalls in der Datei `servlet.properties` erfolgt.

Gestartet wird jo! über ein Skript im Verzeichnis `/webapp/projects/jo/bin` namens `jostart` beziehungsweise `jostart.bat` auf Windows-Systemen. Falls der Klassenpfad nicht ohnehin schon korrekt gesetzt wurde, kann dies im Skript nachgeholt werden.

Wie Sie schon hier erkennen, ist die Konfiguration von Servlets leider nicht bei allen Engines einheitlich. Abhilfe verschafft der mit der Version 2.2 des Servlet-API eingeführte Deployment-Deskriptor – die Beschreibung aller Konfigurationsdaten in einem definierten Format. Die kommentierte DTD des Deployment-Deskriptors befindet sich in Anhang C.

## 3.4 Lebenszyklus von Servlets

Nachdem wir unser erstes Servlet programmiert und ausprobiert haben, wollen wir uns den Lebenszyklus von Servlets genauer ansehen. Aus der Dokumentation von `javax.servlet.Servlet` folgt, daß ein Servlet, nachdem es von der Servlet-Engine instantiiert wurde, drei Lebensabschnitte durchläuft (Abbildung 3.3):

1. Initialisierung durch Aufruf der Methode `init(ServletConfig)`
2. beliebig viele Aufrufe von `service(ServletRequest, ServletResponse)`
3. Freigabe durch `destroy()`

Aufgerufen werden diese Methoden von der Servlet-Engine. Aufwendige und einmalige Aufgaben sollten in der `init(ServletConfig)`-Methode untergebracht werden. `service(ServletRequest, ServletResponse)` sollte sämtliche Programmteile enthalten, die bei jedem Aufruf des Servlets durchgeführt werden. In `destroy()` schließlich muß man eventuell notwendige Aufräumarbeiten erledigen. Es bietet sich an, dort Datenbankverbindungen zu schließen oder persistente Daten zu speichern.

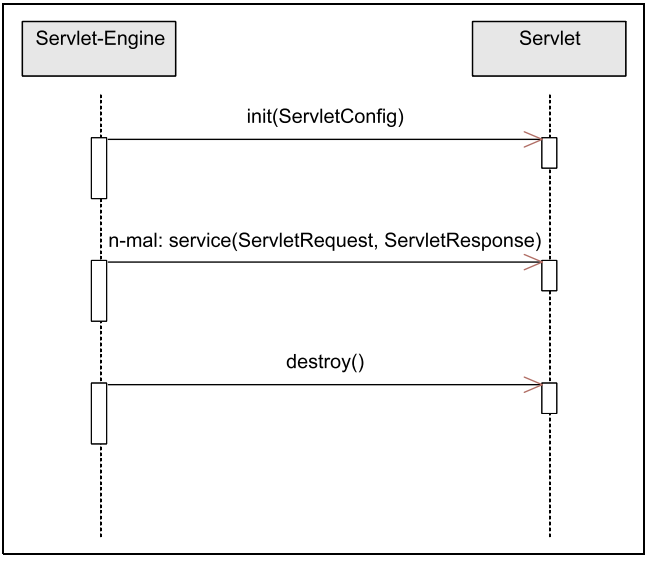


*Abbildung 3.3: Lebenszyklus eines Servlets*

### 3.4.1 Erbsenzähler

Als Beispiel wollen wir ein Zähler-Servlet schreiben (Listing 3.4). Zur Initialisierung soll unser Servlet den Zählerstand aus einer Datei lesen. Bei jedem Aufruf des Servlets wird der Wert um eins erhöht und ausgegeben. Wenn es die Servlet-Engine für nötig hält, unser Servlet aus dem Speicher zu entfernen, soll der Zählerwert in einer Datei gespeichert werden.

```
package de.webapp.Examples.Servlet;
import javax.servlet.GenericServlet;
import javax.servlet.ServletException;
import javax.servlet.ServletResponse;
import javax.servlet.ServletRequest;
import javax.servlet.ServletOutputStream;
import javax.servlet.ServletConfig;
import java.io.IOException;
import java.io.FileInputStream;
import java.io.FileOutputStream;
```

```
import java.io.DataInputStream;
import java.io.DataOutputStream;

public class PersistentCounter extends javax.servlet.GenericServlet {
protected String myFilename;
protected int myCount = 0;
public void init(ServletConfig conf) throws ServletException {
   super.init(conf);
   // Dateinamen lesen
   myFilename = getInitParameter("counterDatei");
   // Default-Dateinamen setzen
   if (myFilename == null) {
      myFilename = new String("counter.txt");
   }
   // Zählerstand lesen
   try {
      FileInputStream fileIn = new FileInputStream(myFilename);
      DataInputStream dataIn = new DataInputStream(fileIn);
      myCount = dataIn.readInt();
      dataIn.close();
   }
   catch (IOException e) {
      // Nichts machen.
   }
}
public void service(ServletRequest req, ServletResponse res)
      throws ServletException, IOException {
   // MIME-Typ gleich "text/html" setzen.
   res.setContentType("text/html");

   // ServletOutputStream des ServletResponse-Objektes
   // besorgen.
   ServletOutputStream out = res.getOutputStream();
   // Zähler inkrementieren...
   myCount++;
   // ... und aktuellen Stand ausgeben.
   out.println("Dieses Servlet wurde " + myCount + " mal aufgerufen. ");
}
public void destroy() {
   // Zählerstand schreiben.
   try {
      FileOutputStream fileOut = new FileOutputStream(myFilename);
      DataOutputStream dataOut = new DataOutputStream(fileOut);
      dataOut.writeInt(myCount);
      dataOut.close();
   }
   catch (IOException e) {
      // Eventuell Fehlermeldung ausgeben.
      System.err.println("Konnte Zählerstand nicht speichern: " + e.toString());
   }
```

```
super.destroy();
}
} // Ende der Klasse
```

*Listing 3.4: Die Klasse* `PersistentCounter`

Um das Servlet zu starten, muß die Datei `servlets.properties` folgenden Eintrag enthalten:

```
...
# Erbsenzähler-Servlet
erbsenzaehler.code=de.webapp.Examples.Servlet.PersistentCounter
erbsenzaehler.initparams=counterDatei=erbsenzaehler.txt
...
```

Die Datei `mappings.properties` sollte folgenden Eintrag enthalten:

```
...
# Erbsenzähler-Servlet
/erbsenzaehler=erbsenzaehler
...
```

Nach dem Start Ihrer Servlet-Engine sollte das Servlet unter der Adresse `http://<hostname>:<port>/erbsenzaehler` zur Verfügung stehen.

### 3.4.2 Initialisierung

Während der Initialisierung besorgen wir uns mittels `getInitParameter(String)` den Dateinamen der Datei, aus der wir den Zählerstand lesen wollen. `getInitParameter(String)` greift dabei auf die Initialisierungsargumente zurück, die in der Datei `servlet.properties` angegeben wurden. Gibt die Methode `null` zurück, bedeutet dies, daß das Initialisierungsargument nicht angegeben wurde. Damit wir das Servlet dennoch benutzen können, setzen wir einen Standarddateinamen (`counter.txt`).

Um später noch auf den Dateinamen zugreifen zu können, speichern wir ihn in der Instanzvariable `myFilename`. Anschließend lesen wir einen Integer aus der Datei und ordnen ihn der Instanzvariablen `myCount` zu. Der Einfachheit halber ignorieren wir sämtliche `IOExceptions`.

Das Speichern des Zählerwertes in einer Instanzvariablen setzt voraus, daß nur eine Instanz dieses Servlets existiert. Dies muß laut Servlet-Spezifikation nur der Fall sein, wenn es sich nicht um eine verteilte Umgebung handelt. Zur Zeit arbeiten jedoch die meisten Servlet-Engines mit nur einer Instanz pro Servlet. Zudem müssen seit dem Servlet-API 2.2 für verteilte Umgebungen optimierte Servlets als solche gekennzeichnet werden.

### 3.4.3 Service

Im Vergleich zu HelloWorld sind die Änderungen minimal. Unsere Zählervariable wird um eins erhöht, und der bedeutsame Satz »Dieses Servlet wurde X-mal aufgerufen.« wird ausgegeben.

### 3.4.4 Freigabe

Um den Zählerstand auch über das Ableben unseres Servlets hinaus zu retten, sichern wir den Stand in der Datei, deren Namen wir während der Initialisierung herausgefunden haben. Sollte dabei etwas schiefgehen, schreiben wir eine kurze Meldung in die Standardfehlerausgabe. Um die `destroy()`-Methode testweise aufzurufen, können Sie beim JSDK/JSWDK das Skript `stopserver` aufrufen.

## 3.5 Ausführungsumgebung

Bis jetzt haben wir uns auf die »reine« Servlet-Programmierung beschränkt und uns wenig um die Umgebung gekümmert. Lediglich mit der Methode `getInitParameter(String)` haben wir einen kurzen Blick über das Servlet hinaus gewagt. Das soll sich nun ändern.

### 3.5.1 Konfiguration mit ServletConfig

Wie wir schon beschrieben haben, wird zur Initialisierung eines Servlets die `init(ServletConfig)`-Methode aufgerufen. Dabei wird ein Objekt des Typs `javax.servlet.ServletConfig` als Argument übergeben. Die Schnittstelle `ServletConfig` bietet Methoden an, die den Zugriff auf die Initialisierungsparameter und ein `ServletContext`-Objekt ermöglichen (Tabelle 3.2).

| Methode | Bedeutung |
|---|---|
| `String getInitParameter(String name)` | Gibt das entsprechende Initialisierungsargument aus der Datei `servlet.properties` zurück. |
| `Enumeration getInitParameterNames()` | Liefert eine Aufzählung (siehe Java-API-Dokumentation: `java.util.Enumeration`) aller Parameternamen. |
| `ServletContext getServletContext()` | Gibt das `ServletContext`-Objekt des Servlets zurück (mehr dazu im nächsten Abschnitt). |
| `String getServletName()` | Gibt den Namen zurück, unter dem das Servlet bei der Servlet-Engine registriert wurde. |

*Tabelle 3.2: Methoden der Schnittstelle `ServletConfig`*

Zwangsweise muß jede Servlet-Engine jedem Servlet ein `ServletConfig`-Objekt zur Verfügung stellen – sonst wäre der korrekte Aufruf unmöglich. Zusätzlich bietet es sich an, die Schnittstelle auch in eigenen Servlets zu implementieren, um den Zugriff auf Initialisierungsparameter und den Kontext zu vereinfachen. Dies ist in `GenericServlet` verwirklicht.

### 3.5.2 Schnittstelle zur Engine: ServletContext

Genau wie bei `ServletConfig` handelt es sich bei `javax.servlet.ServletContext` um eine Schnittstelle, welche von einer Klasse der Servlet-Engine implementiert werden muß. Der Kontext ist die Umgebung, in der ein Servlet ausgeführt wird. Seit dem Servlet-API 2.2 wird diese Umgebung auch *Web-Applikation* genannt. Jede Servlet-Instanz besitzt genau einen Kontext, auf den sie über das `ServletConfig`-Objekt zugreifen kann. Tabelle 3.3 gibt einen Überblick über die Methoden der Schnittstelle.

Über `ServletContext` kann man alle Eigenschaften der Umgebung erfahren sowie einige umgebungsspezifische Dienste in Anspruch nehmen. Es ist wichtig, sich klarzumachen, was genau diese Umgebung ist beziehungsweise wie sie begrenzt ist.

Schon immer boten Servlet-Engines, die ihre Dienste für verschiedene Domänen zur Verfügung stellen (virtuelle Hosts), jeder Domäne einen eigenen Kontext. Seit dem Erscheinen des Servlet-API 2.1 werden `ServletContexte` nun nicht mehr nur für verschiedene Hosts benutzt, sondern auch auf den URI-Raum eines einzelnen Hosts abgebildet. So ist es möglich, daß ein Servlet mit dem URI `/context1/meinServlet` einen anderen Kontext besitzt als ein Servlet mit dem URI `/context2/meinServlet` (Abbildung 3.4). Die Spezifikation stellt weiterhin klar, daß die Granularität der Kontexte mindestens ebenso groß ist wie die virtueller Hosts. Ein Server mit mehreren virtuellen Hosts muß also für jeden Host mindestens einen eigenen Kontext zur Verfügung stellen. Der abgedeckte URI-Raum eines Kontextes darf somit nicht größer sein als der seines Hosts.

Prinzipiell ist es möglich, über die Methode `getContext(String URIPath)` von einem Kontext auf einen anderen Kontext zuzugreifen, sofern der Kontext oder das Servlet nicht besonderen Restriktionen unterliegt. Dies ist beispielsweise der Fall, wenn ein Servlet in einer speziellen Sicherheitsumgebung (Sandbox) ausgeführt wird, die den Zugriff auf Ressourcen des Servers einschränkt. Entspricht dem URI-Pfad kein Kontext, oder ist dem Servlet der Zugriff zum Kontext verwehrt, gibt die Methode `null` zurück.

Nachdem nun klar ist, wie ein Kontext aussieht, wollen wir seine Dienste näher betrachten. Ein sehr wichtiger Dienst ist der Zugriff auf Ressourcen. Die Methode `getResource(String URI)` gibt ein URL-Objekt zurück, das den Zugriff auf Ressourcen unabhängig von einem Dateisystem erlaubt. Es ist Aufgabe der Engine, den angegebenen URI auf einen URL abzubilden. Dabei spielt es keine Rolle mehr, wo sich die Ressource tatsächlich befindet beziehungsweise ob es sich dabei um eine Datei auf einem entfernten Server in einem Java-Archiv (Jar) oder um etwas vollständig anderes handelt. Es zählt einzig und allein die Tatsache, ob sich der Ort mit einem URL beschreiben läßt oder nicht.

Um den Komfort im Umgang mit Ressourcen etwas zu erhöhen, bietet der Kontext zusätzlich die Methode `getResourceAsStream(String URI)` an. Benutzt man diese Methode, gehen jedoch Meta-Informationen wie Länge und Typ der Ressource verloren.

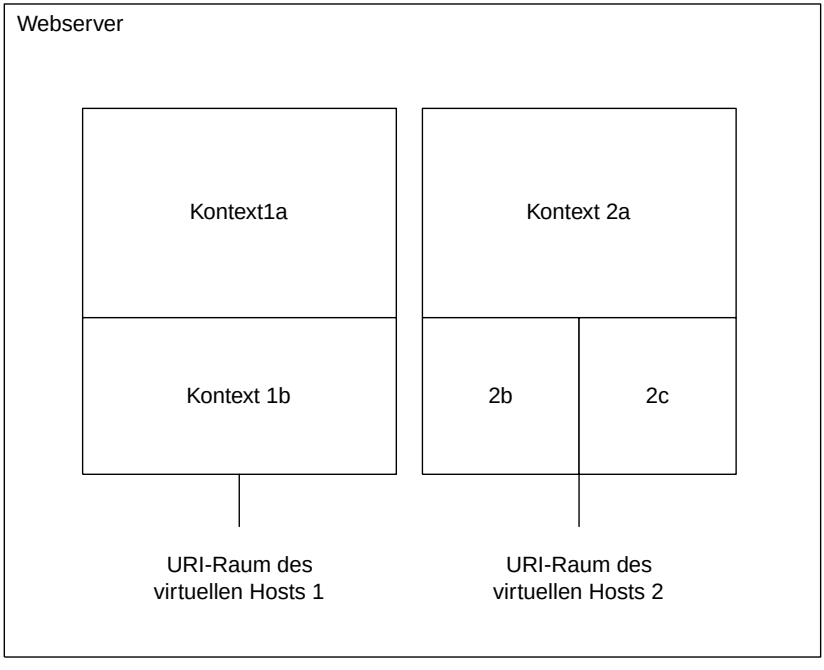


*Abbildung 3.4: Ein virtueller Host kann mehrere disjunkte* `ServletContexte` *enthalten; ein* `ServletContext` *kann sich aber nicht über mehrere Hosts erstrecken.*

Beide Methoden wurden erst in Version 2.1 eingeführt, um die lokal beschränkte Methode `getRealPath()` sinnvoll zu erweitern. Klassisch angewendet werden alle drei Methoden in File-Servlets. File-Servlets analysieren die Anfrage und liefern – ganz in Webserver-Manier – einfach eine Datei aus dem Dateisystem.

Eine weitere wichtige Methode des Kontextes, die erst in der Version 2.1 eingeführt wurde, ist `getRequestDispatcher(String URI)`. Mittels dieser Methode kann man andere Server-Ressourcen aufrufen beziehungsweise einbinden. Dabei darf es sich um beinahe beliebige Ressourcen handeln, die vom Server zur Verfügung gestellt werden. Hierzu zählen unter anderem Servlets, JavaServer-Pages (JSP), Dateien oder CGI-Skripten. Wir beschreiben den `RequestDispatcher` ausführlicher in Abschnitt 3.9.

`getRequestDispatcher()` ermöglicht den Ersatz eines Verfahrens, das vor der Version 2.0 üblich war. Über die nun veralteten Methoden `getServlet()`, `getServlets()` und `getServletNames()` ließ sich eine Referenz auf ein Servlet erlangen. Anschließend konnte dessen `service()`-Methode aufgerufen werden. Dieses Vorgehen birgt einige ernstzunehmende Gefahren, da es keine Möglichkeit gibt festzustellen, in welchem Zustand sich das erlangte Servlet befindet. Somit war es möglich, die `service()`-Methode bereits gelöschter oder noch gar nicht initialisierter Servlets aufzurufen.

Oft wurde über denselben Mechanismus auch auf Instanzvariablen anderer Servlets zugegriffen, um so Objekte gemeinsam zu nutzen. Dieses waghalsige Verfahren ist seit der Version 2.1 des API ebenfalls obsolet. Zusätzlich zu `getAttribute(String)` wurden die Methoden `setAttribute(String, Object)` und `removeAttribute(String)` eingeführt. Somit ist jetzt die problemlose, gemeinsame Nutzung von Objekten durch Gruppen von Servlets möglich.

Zumindest eines der Kontext-Attribute wird übrigens bereits von der Servlet-Engine gesetzt. Dabei handelt sich um ein temporäres Verzeichnis, das sich als `java.io.File`-Objekt hinter dem Schlüssel `javax.servlet.context.tempdir` verbirgt.

Als letzte Eigenschaft des Kontextes sei noch der zentrale Logdienst erwähnt. Über `log(String)` und `log(String, Throwable)` (früher: `log(Exception, String)`) kann man sehr leicht in eine zentrale Logdatei schreiben. Beide Methoden sind zusätzlich bequem über `GenericServlet` (Abbildung 3.2) zugänglich.

| Methode | Bedeutung |
|---|---|
| `String getInitParameter(String name)` | Gibt einen Initialisierungsparameter dieses Kontextes zurück. |
| `Enumeration getInitParameterNames()` | Gibt eine Aufzählung aller Schlüssel für Initialisierungsparameter zurück. |
| `ServletContext getContext(String URI)` | Gibt einen passenden Kontext zum URI zurück. Falls kein passender Kontext gefunden wurde oder das Servlet kein Recht hat, ihn zu erlangen, gibt die Methode `null` zurück. |
| `int getMajorVersion()` | Gibt die Hauptversion des Servlet-API zurück, das vom Server unterstützt wird. Für Version 2.1 wird also 2 zurückgegeben. |
| `int getMinorVersion()` | Gibt die Unterversion des Servlet-API zurück, das vom Server unterstützt wird. Für Version 2.1 wird also 1 zurückgegeben. |
| `String getMimeType(String dateiname)` | Stellt den MIME-Typ einer Datei fest und gibt diesen oder `null` zurück, falls er nicht bekannt ist. |
| `String getRealPath(String virtuellerpfad)` | Gibt den realen Pfad einer Datei zurück. |
| `URL getResource(String URI)` | Gibt ein URL-Objekt zu einem URI zurück. |
| `InputStream getResourceAsStream(String URI)` | Gibt einen Stream zu einer Ressource zurück. |

*Tabelle 3.3: Die Schnittstelle `ServletContext`*

| Methode | Bedeutung |
|---|---|
| `RequestDispatcher getRequestDispatcher(String URI)` | Gibt einen passenden `RequestDispatcher` zu einem URI oder `null` zurück, falls es keinen gibt. Der URI muß relativ zur Basis des ServletContextes angegeben werden und mit einem Schrägstrich beginnen. Siehe auch ServletRequest.getRequestDispatcher(). |
| `RequestDispatcher getNamedDispatcher(String name)` | Gibt einen passenden `RequestDispatcher` zu einem Servletnamen oder `null` zurück, falls es kein Servlet oder keine registrierte Java-ServerPage mit dem angegebenen Namen gibt. |
| `String getServerInfo()` | Gibt Name und Version des Servers zurück. |
| `void log(String nachricht)` | Schreibt eine Nachricht in die zentrale Logdatei. |
| `void log(String nachricht, Throwable t)` | Schreibt eine Nachricht und ein `Throwable`-Objekt beziehungsweise dessen Stacktrace in die zentrale Logdatei. |
| `void log(Exception e, String nachricht)` *deprecated* | Schreibt eine Nachricht und ein `Exception`-Objekt beziehungsweise dessen Stacktrace in die zentrale Logdatei. |
| `Object getAttribute(String name)` | Liefert Attribute der Servlet-Engine, die nicht explizit durch Methoden der Schnittstelle `ServletContext` abgedeckt sind. |
| `void setAttribute(String name, Object o)` | Setzt ein Attribut dieses Kontextes. |
| `void removeAttribute(String name)` | Löscht ein Attribut dieses Kontextes. |
| `Servlet getServlet(String name)` *deprecated* | Gibt eine Referenz auf ein Servlet zurück. Diese Methode sollte nicht mehr benutzt werden. Neuere Engines geben `null` zurück. |
| `Enumeration getServletNames()` *deprecated* | Gibt eine Aufzählung aller Servletnamen zurück. Neuere Engines liefern ein leeres `Enumeration`-Objekt. |
| `Enumeration getServlets()` *deprecated* | Gibt eine Aufzählung aller Servlets zurück. Neuere Engines liefern ein leeres `Enumeration`-Objekt. |

*Tabelle 3.3: Die Schnittstelle `ServletContext` (Fortsetzung)*

### 3.5.3 Anfrageobjekt ServletRequest

Neben der Konfiguration des Servlets mit `ServletConfig` und dem Zugriff auf den Server beziehungsweise die Servlet-Engine mit `ServletContext` bietet das Servlet-API mit der Schnittstelle `ServletRequest` die Möglichkeit, auf die Eigenschaften einer Anfrage zuzugreifen. Viele der Methoden, die `ServletRequest` bereitstellt, liefern Werte, die klassischen CGI-Variablen entsprechen. Sie sind in der folgenden Tabelle aufgeführt.

| Methode | CGI-Äquivalent | Bedeutung |
|---|---|---|
| `int getContentLength()` | `CONTENT_LENGTH` | Gibt die Länge der Anfragedaten an bzw. `-1`, falls diese nicht bekannt ist. |
| `String getContentType()` | `CONTENT_TYPE` | Entspricht dem MIME-Typ der Anfrage, bzw. `null`, falls dieser nicht bekannt ist. |
| `String getProtocol()` | `SERVER_PROTOCOL` | Protokoll mit Versionsnummer. Dabei wird folgendes Format benutzt: `<Protokoll>/<Hauptversion>.<Unterversion>` |
| `String getScheme()` | - | Gibt das Schema an, nach dem der URL, der dieser Anfrage zugrundeliegt, geformt ist. Beispiele: `http`, `ftp` oder `https`. |
| `String getServerName()` | `SERVER_NAME` | Hostname des Servers |
| `int getServerPort()` | `SERVER_PORT` | Portnummer des Servers |
| `String getRemoteAddr()` | `REMOTE_ADDR` | IP-Adresse des Clients |
| `String getRemoteHost()` | `REMOTE_HOST` | Hostname des Clients |

*Tabelle 3.4: Methoden mit CGI-Äquivalent*

Hinzu kommen einige Methoden, die zum Arbeiten mit Servlets unerläßlich sind (Tabelle 3.5). Zu allererst ist `ServletStream getInputStream()` zu nennen. Der `InputStream`, den diese Methode zurückgibt, ist zum Lesen vornehmlich binärer Daten gedacht, wie sie zum Beispiel beim Übertragen von Dateien mittels FTP (File-Transfer-Protokoll) oder beim File-Upload anfallen. Wenn Sie sicher sind, daß nur Textdaten übertragen werden, sollten Sie sich lieber mittels `Reader getReader()` einen Reader anstelle des Stroms besorgen. Die mit dem JDK 1.1 eingeführten Reader und Writer (siehe JDK-Dokumentation, package `java.io`) sind im Gegensatz zu Strömen in der Lage, Zeichensatzkodierungen korrekt zu handhaben.

Egal, ob man nun zu Strom oder Reader greift – hat man das eine getan, ist das andere tabu. Ruft man dennoch beide Methoden auf, wird eine `IllegalStateException` ausgelöst.

Neben dem rohen Zugriff mittels Strom oder Reader stellt jeder `ServletRequest` zudem noch Methoden zum Zugriff auf Parameter zur Verfügung. Unter Parametern werden dabei Schlüssel-Wert-Paare verstanden, die über einen Anfrage-String per `GET` oder über ein Formular per `POST` an den Server geschickt werden können. Die entsprechenden Methoden zum Zugriff auf Parameter heißen `String[] getParameterValues()`, `String getParameter(String name)` und `Enumeration getParameterNames()`. Mit Vorsicht zu genießen ist dabei `String getParameter(String name)`. Sind nämlich mehrere Werte einem Schlüssel zugeordnet, erhält man mit `getParameter(String name)` nur einen Wert. Daher sollte diese Methode auch aus dem Servlet-API verschwinden. Der Protest einiger Entwickler hat jedoch dazu geführt, daß Sun die Methode wieder eingeführt hat. Ist man sich nämlich sicher, daß nur ein Wert übermittelt werden soll, ist es sehr viel komfortabler, direkt

mit einem String zu arbeiten, anstatt immer auf ein Array zuzugreifen. Werden mehrere Parameter gleichen Namens übermittelt, liefert `getParameter(String name)` den ersten. Doch Vorsicht – dies ist erst seit Version 2.2 so definiert. Ebenfalls erst seit Version 2.2 ist festgelegt, daß ein Aufruf von `String[] getParameters()` die per GET übergebenen Parameter vor den per POST übermittelten einordnet. Erfolgt also eine POST-Anfrage mit einem Anfrage-String `Gruss=Hallo` und dem Rumpf `Addressat=Welt`, so gibt `String[] getParameterValues()` als ersten Parameterwert `Hallo` und als zweiten `Welt` zurück.

Listing 3.5 zeigt exemplarisch, wie man Parameter und ihre Werte ausgeben kann:

```
public void showParameters(Servletrequest req, ServletResponse res)
      throws ServletException, IOException {
   // Zunächst einen OutputStream,
   ServletOutputStream out = res.getOutputStream();
   // dann die Parameternamen besorgen.
   Enumeration e = req.getParameterNames();
   while (e.hasMoreElements()) {
      String name = (String)e.nextElement();
      // Jeden Namen ausgeben...
      out.println(name + ":");
      String[] values = req.getParameterValues(name);
      // ...und falls vorhanden, alle seine Werte ausgeben.
      if (values != null) {
         for (int i=0; i < values.length; i++)
            out.println(values[i]);
      }
   }
}
```

*Listing 3.5: Codeausschnitt Parameterausgabe*

Zusätzlich zu den Parametern kann ein Request-Objekt noch weitere Attribute besitzen. Diese lassen sich über die Methode `Object getAttribute(String name)` in Erfahrung bringen. Gewöhnlich handelt es sich um Attribute, die requestspezifisch von der Servlet-Engine gesetzt werden. Beispielsweise verbirgt sich hinter dem Attribut `javax.net.ssl.cipher_suite` der Name der benutzten SSL-Chiffrier-Suite, falls SSL-Klassen (Secure Socket Layer: ein Verfahren, um den Netzverkehr sicher zu verschlüsseln) von Sun benutzt wurden. Es gibt noch weitere dieser Attribute. Ihnen ist gemeinsam, daß sie für ihre Namen als Präfix üblicherweise den Paketnamen derjenigen Klassen benutzen, über die sie Auskunft erteilen.

Seit der Version 2.1 des API gibt es auch eine Methode `Enumeration getAttributeNames()`, die eine Aufzählung aller Attributnamen zurückgibt. Vorher tappte man unweigerlich im Dunkeln. Zudem wurde die Methode `void setAttribute(String name, Object o)` eingeführt. Diese ist insbesondere dann sinnvoll einzusetzen, wenn der Request mittels eines `RequestDispatchers` weitergeleitet wird und zuvor noch einige Zusatzinformationen eingefügt werden sollen.

| Methode | Bedeutung |
|---|---|
| `Object getAttribute(String name)` | Gibt den Wert eines anfragespezifischen Attributs zurück, das nicht schon von den übrigen Methoden abgedeckt wird. Existiert kein Attribut mit dem angegebenen Namen, gibt die Methode `null` zurück. |
| `void setAttribute(String name, Object o)` | Setzt ein Objekt als Attribut dieser Anfrage. Dies ist besonders bei geschachtelten Servlets nützlich (siehe `RequestDispatcher`). Attribute, deren Schlüssel mit `java`, `javax`, `sun` oder `com.sun` beginnen, sind reserviert und sollten nicht benutzt werden. |
| `String getCharacterEncoding()` | Gibt die Zeichensatzkodierung des Anfragerumpfes zurück oder `null`, falls keine kodiert ist. |
| `ServletInputStream getInputStream() throws IOException` | Gibt einen Stream zum Lesen von Binärdaten des Anfragerumpfes zurück. |
| `String getParameter(String name)` | Gibt einen einzelnen Wert eines Parameters zurück oder `null`, falls dieser nicht existiert. |
| `Enumeration getParameterNames()` | Gibt eine Aufzählung aller Parameternamen dieser Anfrage zurück. |
| `String[] getParameterValues(String name)` | Liefert ein Array mit allen Werten dieses Parameters oder `null`, falls der Parameter nicht existiert. |
| `BufferedReader getReader() throws IOException` | Liefert einen Reader, um Text aus dem Anfragerumpf zu lesen. |
| `Locale getLocale()` | Gibt die bevorzugte Sprache des Clients an. Diese Information wird gewöhnlich aus dem Header Accept-Language gewonnen. |
| `Enumeration getLocales()` | Gibt eine Aufzählung der bevorzugten Sprachen in absteigender Reihenfolge Ihrer Akzeptanz zurück. |
| `boolean isSecure()` | Zeigt an, ob der Request über eine sichere Verbindung wie SSL übertragen wurde. Ist dies der Fall und handelt es sich um einen Server, der auf Basis von Java 2 läuft, befindet sich das zugehörige Zertifikat im Attribut mit dem Schlüssel `javax.servlet.request.X509Certificate`. |
| `RequestDispatcher getRequestDispatcher(String URI)` | Gibt einen passenden `RequestDispatcher` zu einem URI oder `null` zurück, falls es keinen gibt. Der URI darf relativ zur aktuell ausgeführten Ressource sein und muß nicht mit einem Schrägstrich beginnen. Siehe auch `ServletContext.getRequestDispatcher()`. |

*Tabelle 3.5: Methoden ohne CGI-Äquivalent*

| Methode | Bedeutung |
|---|---|
| `String getRealPath(String virtuellerpfad)` *deprecated* | Wandelt einen virtuellen in einen realen Pfad um. Diese Methode wurde durch die gleichnamige Methode in `ServletContext` ersetzt. |

*Tabelle 3.5: Methoden ohne CGI-Äquivalent (Fortsetzung)*

### 3.5.4 Antwortobjekt ServletResponse

Die Schnittstelle `ServletResponse` ist das Gegenstück zu `ServletRequest`. Dort sind Methoden gekapselt, die es erlauben, Daten an den Client zurückzuschreiben (Tabelle 3.6). Wie wir schon in einigen Beispielen gesehen haben, kann man auch hier, genau wie in `ServletRequest`, einen Stream erhalten. Die entsprechende Methode heißt `ServletOutputStream getOutputStream()`. Ebenso gibt `PrintWriter getWriter()` einen entsprechenden Writer zurück. Dieser sollte zum Schreiben von Texten benutzt werden. Wiederum gilt, daß man nicht beide Methoden benutzen darf. Darüber hinaus verfügt `ServletResponse` über einige Methoden, um Einfluß auf den Ausgabe-Puffer und Header zu nehmen.

Falls bekannt, kann mit `setContentLength(int length)` die Länge der Antwort gesetzt werden. Dies ist zwar in den meisten Fällen nicht unbedingt nötig, aber aus Performancegründen auf jeden Fall empfehlenswert. Nur wenn dieser Header gesetzt wurde, können mit Browsern, die noch nicht HTTP/1.1 implementiert haben, persistente Verbindungen aufgebaut werden. Der Netscape Navigator 4.61 beispielsweise beherrscht HTTP/1.1 noch nicht.

Aber selbst wenn der Client HTTP/1.1 spricht, ist nicht unbedingt gewährleistet, daß Sie es sich sparen können, die Länge der Antwort zu setzen. Dies ist nur unter zwei Bedingungen ohne Zeitstrafe möglich: Entweder Ihre Servlet-Engine puffert die Antwort, um so die Länge festzustellen, oder sie zerstückelt von sich aus die Antwort in sogenannte Chunks. Das Chunked-Encoding ist in HTTP/1.1 definiert [Fielding et al. 99, S. 24f] und erlaubt es, persistente Verbindungen ohne Wissen der Content-Länge aufrechtzuerhalten.

Um zu testen, ob Ihre Engine das Chunked-Encoding beherrscht oder puffert, verbinden Sie sich einfach via Telnet mit Ihrem Server und fordern das in Abschnitt 3.3 beschriebene `HelloWorld`-Servlet folgendermaßen an:

```
telnet <servername> <port><enter>
GET /HelloWorld HTTP/1.1<enter>
Host: <servername>:<port><enter>
<enter>
```

An der Antwort können Sie leicht erkennen, welches Protokoll Ihr Server verwendet, ob er `Content-Length` oder `Transfer-Coding` und das Schlüsselwort »`chunked`« gesetzt hat.

Seit Version 2.2 des API kann man das Puffern der Ausgabedaten durch das Programm steuern. Mit `void setBufferSize(int size)` läßt sich die Größe des Puffers setzen, und mit `int getBufferSize()` können Sie sie in Erfahrung bringen. Dabei ist zu beachten, daß die Servlet-Engine nicht unbedingt den Wert als Puffergröße setzt, den Sie wünschen. Vielmehr dient Ihre Angabe wirklich nur als Wunsch. Die tatsächliche Puffergröße gibt die Methode `int getBufferSize()` als Rückgabewert zurück. Diese kann unter Umständen größer sein als der gewünschte Wert. Außerdem ist zu beachten, daß Sie die Puffergröße nur setzen dürfen, wenn Sie noch keine Daten mittels `ServletOutputStream` oder `Writer` geschrieben haben. Andernfalls wird eine `IllegalStateException` ausgelöst.

Durch das Verwenden eines Puffers ergibt sich ein gewisser Komfort. So können bereits erfolgte Ausgaben und gesetzte Headerwerte durch den Aufruf von `reset()` wieder gelöscht werden – zumindestens solange `isCommitted()` falsch ist. Wie lange das der Fall ist, hängt wiederum von der Puffergröße und bereits erfolgten Ausgaben ab. `flushBuffer()` erzwingt schließlich das sofortige Schreiben aller gepufferten Daten.

`setContentType(String type)` muß in den meisten Fällen explizit aufgerufen werden. Geschieht dies nicht, wird der Basis-MIME-Typ »text/plain« verwendet. Wichtig ist, den MIME-Typ zu setzen, bevor irgendeine Ausgabe endgültig an den Client geschickt wurde. Falls nicht explizit ein Puffer benutzt wird, ist dies in der Regel der Fall, wenn die erste Ausgabe erfolgt.

Mit `setLocale(Locale)` können Sie den Header `Content-Language` gemäß der Sprache Ihrer Ausgabedaten setzen. Gleichzeitig wird dadurch auch ein der Sprache gerechter Zeichensatz gesetzt. Mehr Informationen zu Lokalisierung finden Sie im Abschnitt 3.10. Schließlich gibt `String getCharacterEncoding()` die benutzte Zeichenkodierung (zum Beispiel ISO-8859-1) zurück.

| **Methode** | **Bedeutung** |
|---|---|
| `String getCharacterEncoding()` | Gibt die verwendete Zeichensatzkodierung zurück. |
| `ServletOutputStream getOutputStream() throws IOException` | Liefert einen `ServletOutputStream`. |
| `PrintWriter getWriter() throws IOException` | Gibt einen `PrintWriter` zum Schreiben von Text zurück. |
| `void setContentLength(int laenge)` | Setzt die Länge der Antwort. |
| `void setContentType(String typ)` | Setzt den MIME-Typ der Antwort. |
| `int getBufferSize()` | Gibt die Größe des benutzten Puffers zurück. |
| `void setBufferSize(int groesse)` | Setzt die Puffergröße. |
| `boolean isCommitted()` | Zeigt an, ob die gepufferten Ausgabedaten bereits endgültig an den Client gesendet wurden. |
| `void flushBuffer()` | Leert den Puffer und sendet die Daten an den Client. |

*Tabelle 3.6: Die Schnittstelle `ServletResponse`*

| | |
|---|---|
| `void reset()` | Löscht alle bereits geschriebenen Ausgaben und Headerwerte, sofern `isCommitted()` `false` zurückgibt. Ist das nicht der Fall, wird eine `IllegalStateException` ausgelöst. |
| `void setLocale(Locale loc)` | Setzt den Header `Content-Language` gemäß dem angegebenen Locale. |

*Tabelle 3.6: Die Schnittstelle `ServletResponse` (Fortsetzung)*

## 3.6 Servlet-Ausnahmen

Natürlich sind auch Servlets nicht gegen Fehler oder fehlerhafte Konfigurationen gefeit. Um dennoch ein verläßliches Verhalten zu garantieren, können sowohl die `init`- als auch die `service`-Methode neben der `IOException` eine `javax.servlet.ServletException` auslösen.

Dies sollte immer dann geschehen, wenn ein Servlet ein Problem nicht selbst lösen kann. Meist ist das verursachende Problem selbst eine Ausnahme, die in einer `ServletException` gekapselt wird. Zu diesem Zweck existieren entsprechende Konstruktoren, die ein `Throwable`-Objekt als Argument akzeptieren:

- public ServletException()
- public ServletException(String nachricht)
- public ServletException(String nachricht, Throwable grund)
- public ServletException(Throwable grund)

Um auf das auslösende `Throwable` zugreifen zu können, kennt die Ausnahme zudem die Methode `Throwable getRootCause()`.

Neben der normalen `ServletException` kennt das API noch eine andere Ausnahme, die `UnavailableException`. `UnavailableException` basiert auf `ServletException` und signalisiert der Servlet-Engine, daß das Servlet entweder zeitweise oder dauerhaft nicht zur Verfügung steht.

Ein dauerhafter Ausfall sollte gemeldet werden, wenn das Servlet auf keinen Fall in der Lage ist, den Dienst wieder aufzunehmen, ohne daß der Server neu gestartet wird. Ein zeitweiser Ausfall wird angezeigt, wenn das auftretende Problem auch zur Laufzeit gelöst werden kann. Zu dieser Klasse von Problemen zählt zum Beispiel der Ausfall dritter Dienste, auf die ein Servlet angewiesen ist. Darunter fallen unter anderem Datenbanken.

Die `UnavailableException` verfügt in der Version 2.2 über zwei neue Konstruktoren:

- public UnavailableException(String nachricht)
- public UnavailableException(String nachricht, int sekunden)

Auf die in ihnen spezifizierten Daten kann man über die Methoden aus Tabelle 3.7 zugreifen. Die als *deprecated* gekennzeichneten Methoden und Konstruktoren sollten seit Version 2.2 des API nicht mehr benutzt werden.

| Methode | Bedeutung |
|---|---|
| `int getUnavailableSeconds()` | Gibt die Zeit an, die das Servlet erwartungsgemäß nicht zur Verfügung steht. Steht das Servlet permanent nicht mehr zur Verfügung, gibt diese Methode `-1` zurück. |
| `boolean isPermanent()` | Zeigt an, ob das Servlet dauerhaft nicht mehr zur Verfügung steht. |
| `Servlet getServlet()` ***deprecated*** | Gibt das betroffene Servlet zurück. Diese Methode sollte nicht mehr benutzt werden. Seit API 2.2 gibt sie `null` zurück. |
| `UnavailableException(Servlet servlet, String nachricht)` ***deprecated*** | Konstruktor, der nicht mehr benutzt werden sollte. |
| `UnavailableException(int sekunden, Servlet servlet, String nachricht)` ***deprecated*** | Konstruktor, der nicht mehr benutzt werden sollte. |

*Tabelle 3.7: Methoden und Konstruktoren von `UnavailableException`*

## 3.7 Thread-Sicherheit

Wer beim Programmieren von Servlets alle Fäden in der Hand behalten will, muß sich darüber im klaren sein, daß die `service()`-Methode seines Servlets unter Umständen von mehreren Threads gleichzeitig aufgerufen wird. Es existieren also gewöhnlich weniger Servlet-Instanzen als Threads, die dieses Servlet gleichzeitig ausführen. Tatsächlich wird ein Servlet in der Regel nur *einmal* instantiiert. Das bedeutet, daß ungeschützte Klassen- und Instanzvariablen während der Ausführung durch Thread `A` von Thread `B` verändert werden können. Dieses Verhalten kann leicht zu inkonsistenten Zuständen führen.

### 3.7.1 Probleme mit der Nebenläufigkeit

Ein Beispiel: Das Servlet `ConcurrentCalculation` (Listing 3.6) berechnet zunächst die Summe zweier Instanzvariablen, anschließend eine Differenz. Zwischen den beiden Rechenschritten ruft es eine Methode auf, die etwa drei Sekunden wartet. Zwischendurch schreibt es die Werte der Instanzvariablen `a` und `b` in die Logdatei. Ruft man nun das Servlet auf, wartet auf die Ausgabe und ruft es dann noch einmal auf, so ergibt sich folgender Log-Eintrag:

```
Init
Service Start
a=10
b=0
Service Ende
Service Start
a=16
b=16
Service Ende
destroy
```

Wartet man dagegen nicht auf die Ausgabe des ersten Aufrufs und ruft das Servlet gleich noch einmal auf, so ergibt sich dieser Log-Eintrag:

```
init
Service Start
a=10
Service Start
a=16
b=6
Service Ende
b=10
Service Ende
destroy
```

Nach jeweils zwei Aufrufen befindet sich das Servlet also in unterschiedlichen Zuständen, nämlich (16,16) oder (16,10). Die einfachste Möglichkeit, dieses unberechenbare Verhalten zu umgehen, bestände darin, die `service()`-Methode als `synchronized` zu definieren. Dadurch würde verhindert werden, daß zwei Threads gleichzeitig auf sie zugreifen. Davon ist aber *dringend* abzuraten.

```
package de.webapp.Examples.Servlet;

import javax.servlet.*;
import java.io.IOException;
import java.io.PrintWriter;

public class ConcurrentCalculation extends javax.servlet.GenericServlet {
int a = 4;
int b = 10;

public void service(ServletRequest req, ServletResponse res)
      throws ServletException, IOException {
   log("Service Start");
   // Erste Berechnung
   a = a + 6;
   log("a=" + a);
   // warten...
   waitABit();
   // Zweite Berechnung
   b = a - b;
```

```
    log("b=" + b);
    log("Service Ende");
    // Ausgabe im Browser
    res.setContentType("text/html");
    PrintWriter out = res.getWriter();
    out.println("Berechnung beendet.");
}

public void waitABit() {
    long waitTill = System.currentTimeMillis() + 3000;
    //drei Sekunden warten
    while (System.currentTimeMillis() < waitTill) {}
}
} // Ende der Klasse
```

*Listing 3.6: Das Servlet* `ConcurrentCalculation`

Wie so oft, ist die einfachste Möglichkeit leider nicht die beste – in diesem Fall sogar die schlechteste. Wird `service()` für `synchronized` erklärt, ist das zwar sicher, verlangsamt aber die Ausführung erheblich. Besser ist es, nur diejenigen Programmteile zu synchronisieren, in denen es wirklich zu Problemen kommen kann. Dies sind die Code-Passagen, in denen Ressourcen genutzt und manipuliert werden, die mehreren Threads zur Verfügung stehen; insbesondere also die Instanz- und Klassenvariablen von Servlets, Dateien, Datenbank- oder Netzwerkverbindungen. Lokale Daten oder Parameter sollten dagegen nicht synchronisiert werden, da dies die Ausführung des Servlets unnötig verlangsamt.

### 3.7.2 Einer nach dem anderen: SingleThreadModel

Eine weitere Möglichkeit, Thread-Sicherheit herzustellen, bietet die Schnittstelle `javax.servlet.SingleThreadModel`. Es handelt sich dabei um ein Interface, das lediglich zum Markieren einer Klasse dient und keinerlei Methoden spezifiziert. Implementiert ein Servlet diese Schnittstelle, so wird der Servlet-Engine damit signalisiert, daß niemals mehr als ein Thread die `service()`-Methode dieses Servlets ausführen darf. Um dennoch kurze Antwortzeiten zu garantieren, legen Servlet-Engines meist einen Pool von Instanzen des Servlets an. Eingehende Anfragen können so immer von einer Instanz bedient werden, deren `service()`-Methode gerade nicht von einem anderen Thread ausgeführt wird.

Der Nachteil dieser Methode ist der mit dem Poolmanagement einhergehende höhere Verwaltungsaufwand und ihr wesentlich höherer Speicherverbrauch. Wer ohnehin speicherhungrige Servlets schreibt, sollte sich deshalb lieber mit echter Synchronisierung beschäftigen. Außerdem können `SingleThreadModel`-Servlets ihre Daten keinesfalls über Instanzvariablen teilen. Der in Abschnitt 3.4.1 beschriebene Zähler würde also nicht funktionieren. Zudem sollten Sie sicherstellen, daß Ihre Servlet-Engine die Schnittstelle auch beachtet, bevor Sie sich darauf verlassen.

### 3.7.3 Sichere Freigabe

Wenn eine Servlet-Engine ein Servlet freigibt (zum Beispiel, um Speicherplatz freizuräumen), ruft sie normalerweise `destroy()` auf, nachdem alle `service()`-Aufrufe beendet wurden, spätestens aber, nachdem eine konfigurationsabhängige Zeit vergangen ist. Das bedeutet, daß Servlets, die sehr zeitaufwendige Aufgaben erledigen, selbst dafür sorgen müssen, daß sie nicht entladen werden, bevor nicht alle Aufgaben beendet worden sind, beziehungsweise belegte Ressourcen wie zum Beispiel Datenbankverbindungen wieder freigegeben werden.

```
package de.webapp.Examples.Servlet;

import javax.servlet.*;
import java.io.IOException;

public class LongRunner extends javax.servlet.GenericServlet {
private int serviceCounter = 0;
private boolean shuttingDown = false;
protected long _PollInterval = 500;

protected void beforeService() {
   serviceCounter++;
}

protected void afterService() {
   serviceCounter--;
}

protected int getNumServices() {
   return serviceCounter;
}

protected void setShuttingDown(boolean flag) {
   shuttingDown = flag;
}

protected boolean getShuttingDown() {
   return shuttingDown;
}

public void service(ServletRequest req, ServletResponse res)
      throws ServletException, IOException {
   beforeService();
   try {
      // Hier den eigentlichen
      // Service-Code einfügen
      // bzw. super.service(req, res);
   }
   finally {
      afterService();
```

```
    }
}

public void destroy() {
    setShuttingDown(true);
    while (getNumServices() > 0) {
        synchronized (this) {
            try {
                wait(_PollInterval);
            }
            catch (InterruptedException e) {
                // evtl. Ausnahmebehandlung einfügen
            }
        }


    }
    super.destroy();
}

} // Ende der Klasse
```

*Listing 3.7: Die Klasse* `LongRunner`

In unserem Beispiel `LongRunner` (Listing 3.7) speichern wir dazu die Anzahl der `service()`-Aufrufe in der privaten Instanzvariablen `serviceCounter`. `serviceCounter` wird in den Methoden `beforeService()` und `afterService()` erhöht beziehungsweise heruntergezählt. Wie ihre Namen schon andeuten, werden diese Methoden vor und nach dem eigentlichen Service-Code aufgerufen. Diesen haben wir in einem `try`-Block gekapselt, um mittels `finally` sicherstellen zu können, daß `afterService()` auch im Falle einer Ausnahmesituation aufgerufen wird.

Damit im Service-Code festgestellt werden kann, ob die Servlet-Engine versucht, das Servlet zu entladen, führen wir außerdem das Flag `boolean shuttingDown` ein. Es wird mit `false` initialisiert und beim Aufruf von `destroy()` auf `true` gesetzt. Über `boolean getShuttingDown()` können ausführende Threads feststellen, ob `destroy()` aufgerufen wurde, und gegebenenfalls entsprechende Maßnahmen ergreifen.

Nachdem `shuttingDown` gleich `true` gesetzt worden ist, wird mittels `int getNumServices()` überprüft, ob einer oder mehrere Threads `service()` ausführen. Ist dies nicht der Fall, wird sofort `super.destroy()` aufgerufen. Ansonsten wird eine durch `long _PollInterval` gesetzte Zeit gewartet und danach erneut überprüft, ob `service()` von mindestens einem Thread ausgeführt wird. So wird verfahren, bis die Anzahl der ausführenden Threads null ist.

Praktischerweise erlaubt es die Struktur von `LongRunner`, sein Verhalten auch nachträglich auf bestehende Servlets anzuwenden. Dazu muß lediglich `LongRunner` von den betreffenden Servlets erben und im `try`-Block der `service()`-Methode `super.service`

(ServletReqest, ServletResponse) aufgerufen werden. Einziger Nachteil: getShuttingDown() ist von der Oberklasse aus natürlich nicht zugänglich.

## 3.8 HTTP-spezifische Servlets

Die bisher besprochenen Beispiele waren alle recht allgemein und kaum HTTP-spezifisch. Um HTTP wirklich nutzen zu können, muß der Request, der vom Browser an ein Servlet gesendet wird, vom Servlet analysiert werden. Uns interessiert zunächst, welcher HTTP-Befehl überhaupt gesendet wurde. Anschließend kümmern wir uns um die Umgebungsvariablen und das Session-Tracking.

### 3.8.1 HttpServlet

Mit der Klasse javax.servlet.http.HttpServlet stellt Sun dem Entwickler eine komfortable Ausgangsbasis für eigene, HTTP-spezifische Servlets zur Verfügung. Obwohl sie ein Abkömmling von GenericServlet ist, hat die service()-Methode hier eine andere Funktion und sollte nicht überschrieben werden. In ihr wird nämlich der vom Client benutzte HTTP-Befehl festgestellt und eine entsprechende Methode aufgerufen. Standardmäßig werden die Befehle GET, POST, HEAD, PUT, DELETE, OPTIONS und TRACE unterstützt. Die entsprechenden Methoden heißen wie die HTTP-Befehle, sind jedoch mit einem »do«-Präfix versehen (also: doGet(), doPost() etc.). Mit Ausnahme von OPTIONS und TRACE senden alle Methoden die HTTP-Fehlermeldung »400 – BadRequest« zurück an den Client, falls sie nicht überschrieben wurden.

Um also ein Servlet zu schreiben, das auf den GET-Befehl reagiert, muß die Methode doGet() überschrieben werden. Gleiches gilt für POST, PUT und DELETE. doHead(), doOptions() und doTrace() können zwar auch überschrieben werden, zumeist ist die Standardrealisierung jedoch völlig ausreichend. Insbesondere gilt dies für doHead(). Sobald nämlich doGet() überschrieben wird, versteht das HttpServlet auch HEAD. Das liegt daran, daß HEAD im Grunde das gleiche macht wie GET, mit dem Unterschied, daß der Dokumentrumpf nicht mitgeschickt wird.

Listing 3.8 zeigt ein simples Servlet, das die Methoden GET, HEAD, POST, OPTIONS und TRACE unterstützt. Explizit überschrieben werden mußten dazu nur die doGet()- und die doPost()-Methode. Beim Aufruf über GET oder POST erscheint die Bestätigung, welche Methode aufgerufen wurde, im Fenster des Browsers.

```
package de.webapp.Examples.Servlet;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.IOException;
import java.io.PrintWriter;

public class SimpleHttpServlet extends javax.servlet.http.HttpServlet {
```

```
/** doGet()-Methode */
public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   res.setContentType("text/html");
   PrintWriter out = res.getWriter();
   out.println("Sie haben dieses Servlet über die Get-Methode aufgerufen. ");
}

/** doPost()-Methode */
public void doPost(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   res.setContentType("text/html");


   PrintWriter out = res.getWriter();
   out.println("Sie haben dieses Servlet über die Post-Methode aufgerufen. ");
}
} // Ende der Klasse
```

*Listing 3.8: Die Klasse* `SimpleHttpServlet`

Sieht man sich `SimpleHttpServlet` aufmerksam an, fällt auf, daß statt des normalen `javax.servlet.ServletRequest`- und des `javax.servlet.ServletResponse`-Objekts ein `javax.servlet.http.HttpServletRequest`- und ein `javax.servlet.http.HttpServletResponse`-Objekt übergeben werden. Darauf gehen wir in den nächsten beiden Abschnitten genauer ein.

### 3.8.2 HttpServletRequest

Wie der Name schon vermuten läßt, ist `javax.servlet.http.HttpServletRequest` die HTTP-spezifische Erweiterung eines `ServletRequests`. Genau wie `ServletRequest` ist auch `HttpServletRequest` eine Schnittstelle, die von einer Klasse der Servlet-Engine realisiert werden muß. Zusätzlich zu den Methoden aus `ServletRequest` besitzt sie hauptsächlich Methoden, um die spezifischen Eigenschaften eines HTTP-Requests abzufragen. Genau wie in `ServletRequest` werden CGI-Programmierern die meisten Methoden sehr bekannt vorkommen, da sie den CGI-Umgebungsvariablen nahezu entsprechen. Tabelle 3.8 zeigt eine Übersicht der zur Verfügung stehenden HTTP-spezifischen Methoden mit CGI-Äquivalent, Tabelle 3.9 führt die übrigen Methoden auf.

| Methode | CGI-Äquivalent | Bedeutung |
|---|---|---|
| `String getAuthType()` | `AUTH_TYPE` | Gibt den Typ der Authentifikation oder `null` zurück. |
| `String getMethod()` | `REQUEST_METHOD` | Gibt die Anfrage-Methode an, also zum Beispiel `GET` oder `POST`. |

*Tabelle 3.8: Methoden mit CGI-Äquivalent*

| Methode | CGI-Äquivalent | Bedeutung |
|---|---|---|
| `String getPathInfo()` | `PATH_INFO` | Gibt den optionalen Teil des Anfragepfades zwischen Servletpfad (siehe `getServletPath()`) und Anfrage-String oder `null` zurück, falls dem Servletpfad keine Extrainformation folgt. Beispiel: `http://host/servlets/meinServlet/optionalerPfad?QueryString` würde `/optionalerPfad` ergeben. |
| `String getPathTranslated()` | `PATH_TRANSLATED` | Macht aus dem virtuellen Pfad `PATH_INFO` einen echten Pfad. Beispiel: `http://host/servlets/meinServlet/optionalerPfad?QueryString` könnte `/usr/local/htdocs/optionalerPfad` ergeben.<br>Ist kein optionaler Pfad angegeben, gibt diese Methode `null` zurück. |
| `String getQueryString ()` | `QUERY_STRING` | Liefert den Anfrage-String-Teil des Request-URI oder `null`. |
| `String getRemoteUser()` | `REMOTE_USER` | Der Name des Nutzers, falls er sich über HTTP-Authentifikation ausgewiesen hat, oder `null`. Siehe auch `getRemoteUserPrincipal()` und `isRemoteUserInRole()`. |
| `String getServletPath()` | `SCRIPT_NAME` | Gibt den Teil des URI zurück, der sich auf das Servlet bezieht. Wenn ein Servlet auf `/servlet/meinServlet` abgebildet wird und der aufgerufene URI `/servlet/meinServlet/extraInfo` lautet, ist der Servletpfad gleich `/servlet/meinServlet`. Wird das Servlet auf die Dateiendung `.shtml` abgebildet und lautet der URI `/pfad/datei.shtml`, so lautet der Servletpfad ebenso. Dies gilt jedoch nur, wenn der Kontextpfad leer ist – andernfalls muß dieser noch abgezogen werden. |

*Tabelle 3.8: Methoden mit CGI-Äquivalent (Fortsetzung)*

Für die wichtigsten Pfad-Methoden gilt folgende zentrale Gleichung:

```
getRequestURI() = getContextPath() + getServletPath() + getPathInfo()
```

Dabei ist zu beachten, daß die `RequestURI` URL-kodiert sein kann. Darüber hinaus gilt:

```
getPathTranslated() = getRealPath(getPathInfo())
```

Das Interface vereinfacht zudem den Zugriff auf den Header eines HTTP-Requests über entsprechende Methoden. So liefert `Enumeration getHeaderNames()` eine Aufzählung aller Headernamen. `Enumeration getHeaders()` gibt eine Aufzählung aller Headerfelder

mit gleichem Headernamen zurück. `String getHeader(String)`, `int getIntHeader(String)` und `long getDateHeader(String)` geben das erste referenzierte Headerfeld als String-Objekt oder Skalar zurück. Weiterhin existieren Methoden, um auf Cookies und Sessions bzw. SessionIDs zuzugreifen. Diese werden wir in einem späteren Abschnitt behandeln.

| Methode | Bedeutung |
|---|---|
| `Cookie[] getCookies()` | Gibt eine Liste aller Cookies dieser Anfrage zurück. Existieren keine Cookies, wird ein leeres Array zurückgegeben. |
| `long getDateHeader(String name)` | Gibt den Wert des verlangten Headerfeldes in Millisekunden seit dem 1. Januar 1970 GMT zurück. Wenn das Headerfeld nicht existiert, gibt diese Methode `-1` zurück. |
| `Enumeration getHeaders(String name)` | Gibt alle Werte mit gleichem Headernamen in Form einer Aufzählung zurück. Der Header Cache-Control kann beispielsweise mehrfach vorkommen. |
| `String getHeader(String name)` | Gibt den ersten Wert des verlangten Headerfeldes oder `null` zurück, falls das Feld nicht existiert. |
| `Enumeration getHeaderNames()` | Gibt eine Aufzählung der Kopffeldnamen dieser Anfrage zurück. |
| `int getIntHeader(String name)` | Gibt den Wert des verlangten Kopffeldes als skalaren Integer zurück. Wenn das Kopffeld nicht existiert, gibt diese Methode `-1` zurück. |
| `String getContextPath()` | Pfad des aktuellen Kontextes. |
| `boolean isRemoteUserInRole(String role)` | Gibt an, ob ein authentifizierter Nutzer eine bestimmte Rolle innehat. Siehe auch `getRemoteUser()`. |
| `Principal getRemoteUserPrincipal()` | Gibt das `Principal`-Objekt eines authentifizierten Nutzers zurück. |
| `String getRequestedSessionId()` | Gibt die `SessionID` dieses Requests zurück. |
| `String getRequestURI()` | Gibt den in der Anfragezeile enthaltenen URI zurück. Ein eventuell vorhandener Anfragestring ist darin nicht enthalten. |
| `HttpSession getSession()` | Gibt die assoziierte Session zurück, beziehungsweise erzeugt sie, falls sie noch nicht besteht. |
| `HttpSession getSession(boolean create)` | Gibt die assoziierte Session zurück, beziehungsweise erzeugt sie, falls sie noch nicht besteht und `create` wahr ist. |
| `Boolean isRequestedSessionIdValid()` | Zeigt an, ob die mit dieser Anfrage assoziierte Session gültig ist. |

*Tabelle 3.9: Methoden des `HttpServletRequest` ohne CGI-Äquivalent*

| Methode | Bedeutung |
|---|---|
| `isRequestedSessionIdFromCookie()` | Zeigt an, ob die verlangte `SessionID` aus einem Cookie stammt. |
| `isRequestedSessionIdFromURL()` | Zeigt an, ob die verlangte SessionID aus einem URL stammt. |
| `isRequestedSessionIdFromUrl()` *deprecated* | Diese Methode wurde einer einheitlichen Schreibweise wegen zugunsten von `isRequestedSessionIdFromURL()` mißbilligt. |

*Tabelle 3.9: Methoden des `HttpServletRequest` ohne CGI-Äquivalent (Fortsetzung)*

Unser Beispiel `RequestInfoServlet` (Listing 3.9) gibt bei einer Anfrage alle verfügbaren Umgebungsvariablen (auch aus `ServletRequest`) aus. Ein ähnliches Servlet (`SnoopServlet`) ist auch als Standardbeispiel im JSDK enthalten.

```
package de.webapp.Examples.Servlet;

import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestInfoServlet extends HttpServlet {
public void doPost (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   // Der Einfachheit halber rufen wir hier doGet auf...
   doGet(req, res);
}

public void doGet (HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   // MIME-Typ der Antwort setzen
   res.setContentType("text/html");
   // Writer besorgen.
   Writer out = res.getWriter();
   // HTML-Header drucken.
   out.write("<html>");
   out.write("<head><title>RequestInfoServlet</title></head>");
   out.write("<body>");
   out.write("<h2>RequestInfoServlet</h2>");
   out.write("<b>Angeforderte URL: </b>");
   out.write (HttpUtils.getRequestURL (req).toString () + "<br>");
   out.write("<h2>Umgebungsvariablen</h2>");
   out.write("<b>Methode: </b>" + req.getMethod() + "<br>");
   out.write("<b>Protokoll: </b>" + req.getProtocol() + "<br>");
   out.write("<b>Anfrage-URI: </b>" + req.getRequestURI() + "<br>");
   out.write("<b>Servlet-Pfad: </b>" + req.getServletPath() + "<br>");
   out.write("<b>Pfad-Info: </b>" + req.getPathInfo() + "<br>");
   out.write("<b>Übersetzter Pfad: </b>" + req.getPathTranslated() + "<br>");
```

```
   out.write("<b>Anfrage String: </b>" + req.getQueryString() + "<br>");
   out.write("<b>Länge: </b>" + req.getContentLength() + "<br>");
   out.write("<b>MIME-Typ: </b>" + req.getContentType() + "<br>");
   out.write("<b>Servername: </b>" + req.getServerName() + "<br>");
   out.write("<b>Serverport: </b>" + req.getServerPort() + "<br>");
   out.write("<b>Nutzer: </b>" + req.getRemoteUser() + "<br>");
   out.write("<b>Clientadresse: </b>"+ req.getRemoteAddr() + "<br>");
   out.write("<b>Clienthostname: </b>" + req.getRemoteHost() + "<br>");
   out.write("<b>Authentifizierungsschema: </b>" + req.getAuthType() + "<br>");
   out.write("<h2>Kopfdaten</h2>");
   Enumeration e = req.getHeaderNames();
   if (e.hasMoreElements()) {
      while (e.hasMoreElements()) {
         String name = (String)e.nextElement();
         out.write("<b>" + name + ": </b> " + req.getHeader(name) + "<br>");
      }
   }
   out.write("<h2>Servlet-Parameter</h2>");
   e = req.getParameterNames();
   while (e.hasMoreElements()) {
      String name = (String)e.nextElement();
      String vals[] = (String []) req.getParameterValues(name);
      if (vals != null) {
         out.write("<b>" + name + ": </b>");
         for (int i = 0; i<vals.length; i++)
            out.write(vals[i] + " ");
         out.write("<br>");
      }
   }
   out.write("<p></body></html>");
}
} // Ende der Klasse
```

*Listing 3.9: Die Klasse* `RequestInfoServlet`

### 3.8.3 HttpServletResponse

Neben zahllosen Status- und Fehlercodes (RFC 2616, Abschnitt 10) besitzt die Schnittstelle `HttpServletResponse` nur wenige Methoden. Vor allem geht es darum, genau diese Codes an den Client zurückzuschicken sowie Kopfdaten zu setzen und abzufragen. In Tabelle 3.10 finden Sie einen Überblick.

Bei den Statuscodes handelt es sich ausnahmslos um `public static final int`-Variablen, deren Namen mit »SC_« für Statuscode beginnen. Angehängt ist eine HTTP-Statusmeldung. Praktischerweise sind alle diese Variablen mit dem HTTP-Statuscode als Wert belegt.

Beispiele:

```
public static final int SC_OK = 200;
public static final int SC_NOT_FOUND = 404;
```

Will man eine Statusmeldung an den Client zurückschicken, die von der üblichen 200 abweicht (zum Beispiel: »302 MOVED TEMPORARILY«), kann man dies mit der Methode `setStatus(int sc)` tun. Es bietet sich an, als Statuscode eine der vordefinierten Konstanten zu verwenden – früher konnte auch noch ein Meldungstext angegeben werden. Die entsprechende Methode ist jedoch mißbilligt worden. `setStatus()` sollte nur verwendet werden, wenn kein Fehler vorliegt. Statuscodes, deren erste Ziffer eine 4 (Client-Fehler) oder 5 (Server-Fehler) ist, sollten mit den Methoden `sendError(int sc)` und `sendError(int sc, String msg)` verschickt werden. Die setzen nämlich nicht nur den Statuscode, sondern senden in jedem Fall auch eine HTML-Seite mit einer kurzen Fehlerbeschreibung an den Client.

Zusätzlich kann man zu jeder Antwort an den Client noch andere Attribute hinzufügen. Dies geschieht mit den Methoden `setHeader(String key, String value)`, `addHeader(String key, String value)`, `setIntHeader(String key, int value)`, `addIntHeader(String key, int value)`, `setDateHeader(String name, long date)` und `addDateHeader(String name, long date)`. Ob ein Headerwert bereits gesetzt wurde, läßt sich über `boolean containsHeader(String name)` feststellen. Das erneute Setzen eines Headerwertes überschreibt den alten.

Des weiteren bietet `HttpServletResponse` mit der Methode `sendRedirect(String location)` die Möglichkeit, den Browser zu einem anderen URL zu schicken. Zu diesem Zweck werden intern einfach folgende zwei Zeilen ausgeführt:

```
setStatus(SC_MOVED_TEMPORARILY);
setHeader("Location", location);
```

Dies veranlaßt den Browser dazu, den in `location` enthaltenen URL zu laden. Dazu noch eine Anmerkung: Sowohl `sendRedirect()` als auch `sendError()` haben den Nebeneffekt, daß sämtliche vorher erfolgten Ausgaben gelöscht, entsprechend ersetzt und abgeschickt werden. Dies ist natürlich nur möglich, wenn `isCommitted()` noch falsch zurückgibt. Ist dies nicht der Fall, wird daher auch eine `IllegalStateException` ausgelöst. Falls Sie nach Aufruf einer dieser Methoden noch Daten ausgeben, werden diese ignoriert.

Neben den bereits genannten Methoden besitzt `HttpServletResponse` noch weitere Methoden zur Handhabung von Cookies und Sessions, auf die wir in den nächsten Abschnitten eingehen wollen.

| Methode | Bedeutung |
|---|---|
| `void addCookie(Cookie cookie)` | Setzt einen Cookie. |
| `boolean containsHeader(String name)` | Gibt an, ob die Antwort ein bestimmtes Kopffeld enthält. |
| `String encodeRedirectURL(String url)` | Kodiert eine SessionID in einem URL, der für die Methode `sendRedirect()` gedacht ist. |

*Tabelle 3.10: Methoden von `HttpServletResponse`*

| Methode | Bedeutung |
|---|---|
| `String encodeRedirectUrl(String url)` *deprecated* | Diese Methode wurde einer einheitlichen Schreibweise wegen zugunsten von `encodeRedirectURL(String url)` mißbilligt. |
| `String encodeURL(String url)` | Kodiert eine SessionID in einem URL. |
| `String encodeUrl(String url)` *deprecated* | Diese Methode wurde einer einheitlichen Schreibweise wegen zugunsten von `encodeURL(String url)` mißbilligt. |
| `void sendError(int statusCode) throws IOException` | Sendet eine Fehlermeldung an den Client. |
| `void sendError(int statusCode, String nachricht) throws IOException` | Sendet eine Fehlermeldung an den Client. Die Nachricht wird als Rumpf der Antwort mitgesandt. |
| `void sendRedirect(String url)` | Sendet den HTTP-Befehl `Moved Temporarily` an den Client. Der angegebene URL muß seit Version 2.2 nicht mehr absolut sein. |
| `void setDateHeader(String name, long datum)` | Setzt das benannte Headerfeld. Das Datum sollte in Millisekunden seit dem 1. Januar 1970 GMT angegeben werden. |
| `void addDateHeader(String name, long datum)` | Fügt das benannte Headerfeld zum Header hinzu. Das Datum sollte in Millisekunden seit dem 1. Januar 1970 GMT angegeben werden. |
| `void setHeader(String name, String wert)` | Setzt ein Headerfeld. Falls schon ein Header gleichen Namens gesetzt wurde, wir dieser überschrieben. |
| `void addHeader(String name, String wert)` | Fügt dem Header ein Headerfeld hinzu, ohne eventuell schon vorhandene Werte zu überschreiben. |
| `void setIntHeader(String name, int wert)` | Setzt einen skalaren Integer als Headerfeld. |
| `void addIntHeader(String name, int wert)` | Fügt einen skalaren Integer als Headerfeld hinzu. |
| `void setStatus(int statusCode)` | Setzt den Statuscode dieser Antwort. |
| `void setStatus(int statusCode, String nachricht)` *deprecated* | Setzt den Statuscode dieser Antwort. Die Nachricht wird als Rumpf der Antwort an den Client zurückgeschickt. |

*Tabelle 3.10: Methoden von `HttpServletResponse` (Fortsetzung)*

### 3.8.4 Cookies

Gerade in Anwendungen wie Online-Shops ist es notwendig, nutzerspezifische Informationen während des Einkaufens zu sammeln. Anders wären virtuelle Einkaufswagen gar nicht möglich (Kapitel 12). Leider ist HTTP ein verbindungs- und zustandsloses Protokoll. Daher fällt es schwer, zwei aufeinanderfolgende Anforderun-

gen eindeutig einem Nutzer zuzuordnen. Zwar könnte man die IP-Adresse des Clients auswerten, doch besteht die Gefahr, daß mehrere Nutzer sich eine IP-Adresse teilen. Gründe hierfür sind zum Beispiel das IP-Masquerading oder die dynamische IP-Adressen-Vergabe durch einen Internet-Service-Provider. Abhilfe versprechen Cookies. Darunter versteht man Informationshäppchen, die ein Client auf Anweisung eines Servers speichert und bei jedem weiteren Aufruf einer Webseite dieses Servers an den Server zurückschickt. Auf diese Weise kann sich ein Client halbwegs verläßlich gegenüber einem Server ausweisen. Das beschriebene Verfahren ist jedoch keinesfalls sicher! Vielfältigen Formen von Betrug sind ohne ein besseres Authentifizierungsverfahren beziehungsweise ohne eine sichere Verbindung – beispielsweise über SSL – Tür und Tor geöffnet.

Das Servlet-API unterstützt die Verwendung von Cookies gemäß RFC 2109 (`http://www.cis.ohio-state.edu/htbin/rfc/rfc2109.html`) recht komfortabel. So läßt sich mit der Methode `addCookie(Cookie aCookie)` der Schnittstelle `HttpServletResponse` ein Cookie setzen. Die Methode `Cookie[] getCookies()` der Schnittstelle `HttpServletRequest` liefert ein Array aller vom Client gesandten Cookies. Die Klasse `javax.servlet.http.Cookie` selbst ist eigentlich nur eine Datenstruktur, deren Attribute sich mittels `get`- und `set`-Methoden setzen und abfragen lassen. Die wohl wichtigsten dieser Attribute sind `Name` und `Value` – sie müssen auch im Konstruktor `Cookie(String name, String value)` angegeben werden.

Neben dem Namen und dem Wert lassen sich noch einige andere Attribute setzen. Normalerweise werden Cookies von Clients nur an die Server zurückgesandt, die sie auch gesetzt haben. Mittels `setDomain(String domain)` läßt sich der Kreis der Server, an die Cookies gesandt werden, auf eine Domain erweitern. Ebenso kann man Cookies mit Kommentaren versehen, ihre Lebensdauer beeinflussen sowie ein Sicherheits-Flag setzen, das anzeigt, daß das Cookie nur über eine sichere Verbindung an den Server zurückgesendet werden darf.

In unserem Beispiel `CookieCounterServlet` (Listing 3.10) besorgen wir uns zuerst ein Array der gesandten Cookies. Wurden mit der Anforderung überhaupt keine Cookies geschickt, ist dieses Array gleich `null`. Ist dies nicht der Fall, gehen wir der Reihe nach die Cookie-Namen durch und suchen unser Cookie. Falls wir es finden, speichern wir es in der Variable `myCookie`. Finden wir es nicht oder wurden keine Cookies gesandt, instantiieren wir ein neues Cookie. Anschließend lesen wir mit `getValue()` den Wert unseres Cookies und erhöhen ihn um eins. Nun müssen wir das neue Cookie nur noch mittels `addCookie(myCookie)` setzen und den Zählerstand mit `write(...)` ausgeben.

```
package de.webapp.Examples.Servlet;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
```

```
public class CookieCounterServlet extends HttpServlet {
public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   // MIME-Typ gleich "text/html" setzen.
   res.setContentType("text/html");

   // Writer besorgen.
   Writer out = res.getWriter();

   // Cookies besorgen.
   Cookie[] cookies = req.getCookies();
   Cookie myCookie = null;

   // Das richtige Cookie finden.
   int i = 0;
   if (cookies != null) {
      while (myCookie == null && i < cookies.length) {
         if (cookies[i].getName().equals("CookieCounterServlet.anzahl"))
            myCookie = cookies[i];
         i++;
      }
   }

   // Falls unser Cookie nicht gefunden wurde,
   // neues instantiieren.
   if (myCookie == null)
      myCookie = new Cookie("CookieCounterServlet.anzahl", "0");
   // Zählerstand herausfinden und erhöhen.
   Integer anzahl = new Integer(new Integer(myCookie.getValue()).intValue() + 1);
   // Neuen Zählerstand im Cookie setzen.
   myCookie.setValue(anzahl.toString());

   // Das Cookie in der Antwort setzen.
   res.addCookie(myCookie);

   // Meldung an den Client schreiben.
   out.write("Sie haben dieses Servlet " anzahl.toString() + " mal aufgerufen.");
}
} // Ende der Klasse
```

*Listing 3.10: Die Klasse `CookieCounterServlet`*

Wenn wir das Servlet mit einem Browser aufrufen, erscheint wie gewünscht der Satz:

```
Sie haben dieses Servlet 1 mal aufgerufen.
```

Bei jedem weiteren Aufruf erhöht sich dieser Wert um eins. Beendet man nun den Browser und startet ihn erneut, fängt der Zähler wieder bei eins an. Das liegt daran, daß Browser normale Cookies bei Programmende löschen. Nur wenn die Lebensdauer über `setMaxAge(int expiry)` explizit größer als null gesetzt wurde, kann ein Cookie den Browser überleben. `expiry` gibt dabei die maximale Lebensdauer des Cookies in Sekun-

den an. Setzt man einen negativen Wert, verhält sich das Cookie standardmäßig. Setzt man eine Null, wird es gelöscht.

Zusätzlich zu den schon genannten Einschränkungen gelten für Cookies noch folgende Begrenzungen:

- Ein Browser kann nicht mehr als 300 Cookies speichern.
- Ein Cookie darf nur 4 kByte groß sein.
- Je Domain oder Server dürfen nur 20 Cookies verwendet werden. Das bedeutet, daß an einen Server maximal 40 Cookies (20 pro Domain plus 20 pro Server) gesendet werden können.

Trotz dieser Beschränkungen ist die Datenmenge, die in Cookies hin- und hergeschickt werden kann, nicht zu unterschätzen. Immerhin sind Datenvolumina von 160 kByte pro Request möglich. Im Extremfall kann deshalb bei einer HTML-Seite mit mehreren Bildern die transportierte Datenmenge in den Megabyte-Bereich wachsen. Daher muß man bezweifeln, ob es überhaupt sinnvoll ist, umfangreichere Daten in Cookies abzulegen. Effizienter ist es, eine Referenz auf Daten, die auf dem Server liegen, im Cookie abzuspeichern. Alle Daten, die ein Cookie enthalten kann, sind dem Server ohnehin bekannt – warum also das Netz unnötig mit Daten belasten, wenn ein Zeiger völlig ausreicht? Dieser Zeiger ist sehr häufig auch eine SessionID.

### 3.8.5 Sessions

Unter einer Session versteht man eine zusammenhängende Abfolge von Anforderungen und Antworten zwischen Client und Server. Da HTTP ein verbindungsloses Protokoll ist, muß bei jeder Frage und Antwort eine Identifikation mitgeschickt werden, um mehrere Anforderungen als zusammenhängend erkennen zu können. Wie oben beschrieben, ist dies mit Cookies möglich. Eine andere Möglichkeit besteht darin, alle in einer Webseite enthaltenen URLs so umzuschreiben, daß sie einen Identifikator – eine SessionID – enthalten. Dieses Verfahren wird URL-Rewriting genannt.

#### *Grundlegende Session-Unterstützung*

Beide Möglichkeiten werden vom Servlet-API unterstützt. Entsprechende Methoden finden sich sowohl in der Schnittstelle `HttpServletRequest` als auch in `HttpServletResponse`. Zudem bietet das API noch die Schnittstelle `javax.servlet.http.HttpSession`, die ((Bezug: Schnittstelle?)) Methoden zum komfortablen Umgang mit Sessions enthält.

```
package de.webapp.Examples.Servlet;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import java.util.Date;
```

```
public class SessionServlet extends HttpServlet {
/**
Gibt Werte, die mit der Session zusammenhängen, aus.
 */
public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   // Session besorgen. Falls nicht vorhanden,
   // wird eine neue instantiiert.
   HttpSession session = req.getSession(true);

   // MIME-Typ gleich "text/html" setzen.
   res.setContentType("text/html");

   // Writer besorgen.
   Writer out = res.getWriter();
   // Titel ausgeben.
   out.write("<h2>SessionServlet</h2>");

   if (session == null) {
      // Melden, falls keine Session erlangt werden konnte.
      out.write("Konnte keine Session erlangen.");
      return;
   }
   // SessionID ausgeben.
   out.write("<b>Session-ID: </b>" + session.getId() + "<br>");
   // Wurde die Session gerade erst angelegt?
   out.write("<b>Session ist neu: </b>" + session.isNew() + "<br>");
   // Wann wurde die Session erstellt?
   out.write("<b>Session wurde erstellt: </b>"
      + new Date(session.getCreationTime()) + "<br>");
   // Wann wurde zum letztenmal auf die Session zugegriffen?
   out.write("<b>Letzter Zugriff: </b>"
      + new Date(session.getLastAccessedTime()) + "<p>");
   // Mit welchem URL wurde auf dieses Servlet zugegriffen?
   String requestURL = HttpUtils.getRequestURL(req).toString();
   out.write("<b>Angeforderte URL: </b>" + requestURL + "<br>");
   // Wie sieht dieser URL kodiert aus?
   String encodedURL = res.encodeUrl(requestURL);
   out.write("<b>Kodierte URL: </b>" + encodedURL + "<br>");
   // Wie sieht dieser URL kodiert aus, wenn er als
   // Umleitung dienen soll?
   String redirectURL = res.encodeRedirectUrl(requestURL);
   out.write("<b>Kodierter Umleitungs-URL: </b>" + redirectURL + "<p>");

   if (req.isRequestedSessionIdFromCookie()) {
      // Wenn die Session aus einem Cookie gelesen wurde,
      // muß der Reload-URL nicht kodiert werden.
      out.write("Diese SessionID wurde aus einem Cookie gelesen.<p>");
      out.write("<a href='" + requestURL + "'>Reload</a>");
   }
   else {
```

```
      if (req.isRequestedSessionIdFromUrl())
      // Hinweis nur ausgeben, wenn die SessionID
      // auch tatsächlich aus dem URL rekonstruiert wurde.
      out.write("Diese SessionID wurde aus dem URL rekonstruiert.<p>");
      // Jedoch auf jeden Fall einen kodierten URL ausgeben.
      out.write("<a href='" + encodedURL + "'>Reload</a>");
   }
}

} // Ende der Klasse
```

*Listing 3.11: Die Klasse* `SessionServlet`

Unser `SessionServlet` (Listing 3.11) erlaubt einen Einblick in die wichtigsten Session-Mechanismen. Zunächst wird mit `HttpSession getSession(true)` ein `HttpSession`-Objekt besorgt. Durch das Argument `true` wird veranlaßt, daß ein neues Objekt instantiiert wird, falls nicht schon eins vorhanden ist. Den gleichen Effekt hat die Methode `HttpSession getSession()`, sie ist aber erst ab der API-Version 2.1 realisiert. Mußte ein neues Objekt erzeugt werden, gibt die Methode `boolean isNew()` den Wert `true` zurück, ansonsten `false`. Dabei ist zu beachten, daß es zwei Situationen gibt, die dazu führen, daß eine Session als »neu« deklariert wird. Zum einen kann es sich um die erste Anforderung eines Nutzers handeln. Zum anderen ist es aber auch möglich, daß ein Client keine Cookies unterstützt und kein URL-Rewriting verwendet wurde. In diesem Fall erscheint der Servlet-Engine jede Anfrage des Clients als neu, da ihr nie eine SessionID zurückgesandt wird. Für die Praxis bedeutet dies entweder den konsequenten Einsatz von URL-Rewriting, der sehr aufwendig sein kann, oder die dringende Empfehlung, umfangreichere Ressourcen nicht an neue Sessions zu binden. Hinzu kommt die Empfehlung, zumindest bei öffentlichen Websites eine Servlet-Engine einzusetzen, die es ermöglicht, die maximale Anzahl aktiver Sessions zu begrenzen. Potentielle Angreifer können sonst die ausführende VM leicht ins Reich der `OutOfMemoryException` schicken. Eine andere, etwas aufwendigere Möglichkeit, sich zu schützen, besteht darin, selbst die Anzahl der erzeugten Sessions nachzuhalten und bei einer entsprechenden Attacke einfach keine neuen Sessions zu erzeugen.

Eine Bedingung für den ordnungsgemäßen Verlauf einer Session ist übrigens, daß die Methode `HttpSession getSession(boolean create)` mindestens einmal aufgerufen wird, bevor etwas in den Ausgabe-Stream geschrieben wird.

Das erhaltene Session-Objekt verfügt über Eigenschaften wie Erstellungsdatum, letzter Zugriff etc. (Tabelle 3.11), die wir ausgeben. Zudem zeigt unser Servlet noch an, wie der aufgerufene URL aussah. Um einen Einblick zu erhalten, wie ein umgeschriebener URL aussieht, lassen wir uns diesen ausgeben (Abbildung 3.5).

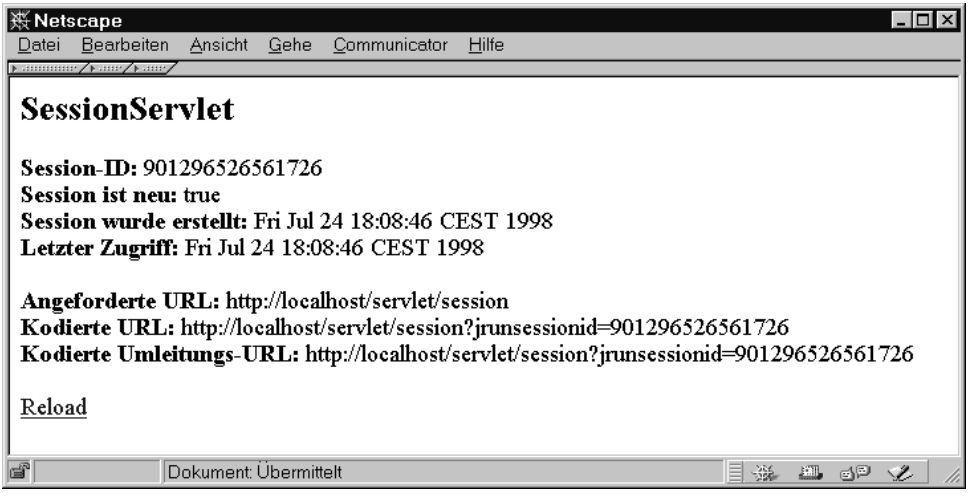


*Abbildung 3.5: SessionServlet-Ausgabe, erstellt mit JRun von Allaire*

### *Session als Datenstruktur*

Wirklich gewinnbringend kann man eine Session erst nutzen, wenn man ihr Objekte zuordnet. Allein mit der ID kann schließlich niemand etwas anfangen. Zu diesem Zweck verfügt `HttpSession` über die Methoden `Object getAttribute(String name)`, `Enumeration getAttributeNames()`, `setAttribute(String name, Object value)` und `removeAttribute(String name)`. Übrigens hießen diese Methoden vor dem Servlet-API 2.2 noch `Object getValue(String name)`, `String[] getValueNames()`, `putValue(String name, Object value)` und `removeValue(String name)`.

Somit ist eine Session eigentlich eine zeitlich befristet gültige Datenstruktur, ähnlich einer Hashtabelle mit Verfallsdatum. Wird eine Session längere Zeit nicht benutzt und das Verfallsdatum überschritten, wird sie ungültig. Das bedeutet, daß auf das `HttpSession`-Objekt und seine Werte nicht mehr zugegriffen werden kann. Dies ist sinnvoll, da es keinen impliziten Weg gibt, auf dem der Session ihr Ende mitgeteilt werden könnte; und auf ein explizites Abmelden des Nutzers sollte und kann man sich nicht verlassen. Schließlich könnte auch das Netz zusammenbrechen oder der Client-Rechner abstürzen – die Servlet-Engine würde es nicht merken. Daher ist ein Timeout-Mechanismus notwendig. Üblicherweise liegt die maximale Überlebensdauer einer nicht aktiven Session bei 30 Minuten. Ob eine Session für ungültig erklärt werden muß, überprüft regelmäßig ein Thread mit niedriger Priorität.

| Methode | Bedeutung |
|---|---|
| `long getCreationTime()` | Gibt an, wann diese Session begann. Der Wert wird in Millisekunden seit dem 1.1.1970 angegeben. |
| `String getId()` | Gibt die vom Server vergebene Session-ID zurück. |
| `long getLastAccessedTime()` | Gibt an, wann vor dem aktuellen Request das letzte Mal auf die Session zugegriffen wurde. Ist die Session neu, gibt diese Methode `-1` zurück. |
| `boolean isNew()` | Gibt an, ob diese Session neu ist. |
| `int getMaxInactiveInterval()` | Gibt an, wie lange diese Session gültig bleibt, ohne daß auf sie zugegriffen wird, bevor sie automatisch von der Servlet-Engine für ungültig erklärt wird. Die Zeit wird dabei in Sekunden angegeben. |
| `void setMaxInactiveInterval(int sekunden)` | Setzt die Zeit, die eine Session gültig bleibt, ohne daß auf sie zugegriffen wird, bevor sie automatisch von der Servlet-Engine für ungültig erklärt wird. Die Zeit wird dabei in Sekunden angegeben. |
| `Object getAttribute(String name)` | Gibt ein in der Session gespeichertes Objekt zurück. |
| `Enumeration getAttributeNames()` | Gibt eine Aufzählung der Namen aller in einer Session hinterlegten Attribute zurück. |
| `void setAttribute(String name, Object wert)` | Hinterlegt ein Objekt unter dem angegebenen Namen in der Session. |
| `void removeAttribute(String name)` | Entfernt das Attribut mit dem angegebenen Namen, sofern es vorhanden ist. |
| `void invalidate()` | Markiert die Session als ungültig. Alle folgenden Zugriffe auf ihre Methoden führen zu `IllegalStateExceptions`. |
| `Object getValue(String name)` *deprecated* | Alte Version von Object `getAttribute(String name)` |
| `String[] getValueNames()` *deprecated* | Alte Version von `Enumeration getAttributeNames()` |
| `void putValue(String name, Object wert)` | Alte Version von `void setAttribute(String name, Object wert)` |
| `void removeValue(String name)` | Alte Version von `void removeAttribute(String name)` |
| `HttpSessionContext getSessionContext()` *deprecated* | Gibt ein Dummy-Objekt zurück, das über keinerlei Funktionalität verfügt. |

*Tabelle 3.11: Methoden von `HttpSession`*

Für den Fall, daß eine Session während ihres Gebrauchs verfällt, gibt es die Möglichkeit, mit der Methode `isRequestedSessionIdValid()` des `HttpServletRequest`-Objekts zu überprüfen, ob die gesandte SessionID noch gültig ist.

### *Kein Bund fürs Leben*

Möglicherweise ist es hilfreich, wenn den an die Session gebundenen Objekten mitgeteilt wird, wenn sie gebunden und entbunden werden. Nützlich ist dies vor allem für begrenzte oder geteilte Ressourcen wie Datenbankverbindungen sowie für stark speicherverbrauchende Objektnetze. Letztere können auf diese Weise explizit freigegeben werden, wenn man sie nicht mehr benötigt. Um dies zu ermöglichen, können Objekte die Schnittstelle `javax.servlet.http.HttpSessionBindingListener` implementieren. Tun sie dies, wird die Methode `valueBound(HttpSessionBindingEvent event)` aufgerufen, wenn das Objekt gebunden wird, und die Methode `valueUnbound(HttpSessionBindingEvent event)`, wenn das Objekt entbunden wird. Das dabei übergebene Ereignisobjekt `HttpSessionBindingEvent` enthält neben der Quelle auch noch den Namen der Quelle sowie die Session, von der der `HttpSessionBindingListener` ge- oder entbunden wird.

```
package de.webapp.Examples.Servlet;

import javax.servlet.http.*;
import javax.servlet.*;
import java.io.*;
import java.util.Date;

public class SessionBindingServlet extends HttpServlet {
public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   // Session besorgen. Falls nicht vorhanden,
   // wird eine neue instantiiert.
   HttpSession session = req.getSession(true);

   // Wie üblich MIME-Typ gleich "text/html" setzen und Writer besorgen.
   res.setContentType("text/html");
   Writer out = res.getWriter();
   // Titel ausgeben.
   out.write("<h2>SessionServlet2</h2>");

   if (session == null) {
      // Melden, falls keine Session erlangt werden konnte.
      out.write("Konnte keine Session erlangen.");
      return;
   }
   // ID besorgen.
   String sessionID = session.getId();

   // Eigenen Namen besorgen.
   String servletName = req.getContextPath() + req.getServletPath();
```

```
   // Abfragen, ob die Session verfallen soll.
   // Der Einfachheit halber wird nur getestet, ob es einen
   // Query-String gibt.
   if (req.getQueryString() != null) {
      session.invalidate();
      out.write("Session " + sessionID + " wurde gelöscht.<p>");
      out.write("<a href='" + servletName
         + "'>Session neu anlegen.</a>");
      return;
   }

   // Testen, ob die Session neu ist.
   if (session.isNew()) {
      out.write("Session " + sessionID
         + " wurde erfolgreich angelegt.<p>");
      out.write("<a href='" + servletName
         + "'>Großes Objekt anlegen.</a>");
      return;
   }

   // Großes Objekt von der Session erfragen.
   GrossesObjekt grossesObjekt =
      (GrossesObjekt)session.getAttribute("grossesObjekt");
   if (grossesObjekt == null) {
      // Da es noch kein GrossesObjekt gibt, eins instantiieren...
      grossesObjekt = new GrossesObjekt(getServletContext());
      // ... und an die Session binden.
      session.setAttribute("grossesObjekt", grossesObjekt);
      out.write("Grosses Objekt wurde angelegt und in Session"
         + sessionID + " gespeichert.<p>");
      out.write("<a href='" + servletName
         + "'>Großes Objekt benutzen.</a>");
      return;
   }
   // GrossesObjekt benutzen.
   grossesObjekt.nutzMich();
   out.write("Grosses Objekt der Session " + sessionID
      + " wurde benutzt.<p>");
   out.write("<a href='" + servletName
      + "'>Großes Objekt erneut benutzen.</a><p>");
   out.write("<a href='" + servletName +
      "?verfallen'>Session verfallen lassen.</a>");
}

} // Ende der Klasse

/** Dummy-Objekt, das HttpSessionBindingListener implementiert. */
class GrossesObjekt implements HttpSessionBindingListener {
/** ServletContext, den wir zum Loggen benötigen. */
protected ServletContext context = null;
```

```
/** Hier wird nur der ServletContext gesetzt. */
public GrossesObjekt(ServletContext context) {
   this.context = context;
}

/** Dummy-Methode */
public void nutzMich() {
   log("Großes Objekt wurde benutzt.");
}

/** Wird beim Binden einer Instanz dieses Objektes
an eine Session aufgerufen. */
public void valueBound(HttpSessionBindingEvent event) {
   String sessionID = event.getSession().getId();
   log("Großes Objekt wurde von der Session " + sessionID
      + " gebunden.");
}

/** Wird beim Entbinden einer Instanz dieses Objektes
von einer Session aufgerufen. */
public void valueUnbound(HttpSessionBindingEvent event) {
   String sessionID = event.getSession().getId();
   log("Großes Objekt wurde von der Session " + sessionID
      + " entbunden.");
}

/** Hilfsmethode zum Loggen */
public void log(String msg) {
   context.log("GrossesObjekt: " + msg);
}
} // Ende der Klasse
```

*Listing 3.12: Die Klasse* `SessionBindingServlet`

Das Beispiel `SessionBindingServlet` (Listing 3.12) demonstriert, wie dies vor sich geht. Beim ersten Aufruf wird zunächst eine Session erzeugt, beim zweiten ein `GrossesObjekt` instantiiert und mittels `setAttribute(String name, Object value)` an die Session gebunden. Beim dritten Aufruf kann der Nutzer das Objekt benutzen, beim vierten erhält er die Möglichkeit, das Objekt erneut zu benutzen oder aber die Session verfallen zu lassen. In der Log-Datei wird dabei festgehalten, wenn das Objekt gebunden, entbunden oder benutzt wurde.

Die Existenz einer Reihenfolge für mögliche Aktionen unterstreicht, daß die Kommunikation zwischen Client und Server nun nicht mehr zustandslos ist. Lediglich, wenn der Client (Browser) keine Cookies annimmt, tritt ein Problem auf. Die Session scheint immer neu zu sein. Aus diesem Grund legen wir das `GrosseObjekt` auch erst an, wenn wir sicher wissen, daß der Client cookiebasierte Sessions unterstützt. Dies ist genau dann der Fall, wenn die Session nicht mehr neu ist. Täten wir dies nicht, würde das

ressourcenhungrige `GrosseObjekt` bei jeder Anforderung eines Clients neu instantiiert – dem Mißbrauch stände Tür und Tor offen.

Natürlich hätten wir auch einfach das URL-Rewriting nutzen können. Da aber das URL-Rewriting insbesondere bei größeren Anwendungen sehr aufwendig ist – schließlich muß jeder einzelne URL umgeschrieben werden –, erscheinen Cookies als die sauberere Lösung, selbst wenn einige Browser Cookies nicht unterstützen oder die Annahme verweigern.

## 3.9 Umleiten mit RequestDispatcher

Die Schnittstelle `javax.servlet.RequestDispatcher` dient dazu, Ressourcen eines Servers in ein Servlet einzubinden, beziehungsweise dazu, eine Anfrage weiterzuleiten. Sie wurde in der Version 2.1 des API eingeführt. Unter Ressourcen kann man dabei alles verstehen, was vom Server auch nach außen hin angeboten wird – also Servlets, Dateien, CGI-Skripten etc. Soll beispielsweise die Datei, die sich hinter dem URI `/index.html` verbirgt, in die Ausgabe eines Servlets eingebunden werden, so muß zunächst ein `RequestDispatcher` für diesen URI von der Servlet-Engine angefordert werden.

```
RequestDispatcher aDispatcher = ServletContext.getRequestDispatcher("/index.html");
```

Wichtig ist dabei, daß der URI relativ zur Basis des `ServletContext` angegeben werden muß, wenn der Dispatcher über einen `ServletContext` angefordert wird. Ist also ein Kontext auf den URI `/meinKontext` abgebildet, so verweist `getRequestDispatcher("/index.html")` letztendlich auf den URI `/meinKontext/index.html`. Ähnlich verhält es sich, wenn Sie den Dispatcher über die gleichlautende Methode des `ServletRequests` anfordern. Der einzige Unterschied ist, daß der URI hier nicht mit einem Schrägstrich beginnen muß, sondern auch relativ zum aktuellen Request interpretiert werden kann:

```
RequestDispatcher aDispatcher =
ServletRequest.getRequestDispatcher("LoginServlet?Name=Schreiber");
```

In beiden Fällen gilt, daß Sie über einen Query-String Argumente übergeben können. Diese werden vom Servlet-Container für die Dauer eines `include()`- oder `forward()`-Aufrufs vor den ursprünglichen Request-Parametern eingeordnet.

Um sich gar nicht erst mit URIs herumärgern zu müssen, existiert seit dem Servlet-API 2.2 zudem die Methode `getNamedDispatcher("einServletName")`. Mit ihr läßt sich ein `RequestDispatcher` für eine benannte Ressource erlangen. Dabei handelt es sich in der Regel um ein Servlet oder eine Java-ServerPage, deren Name der Engine bekannt ist.

Nachdem man den `RequestDispatcher` erhalten hat, kann man dessen Methode `include(ServletRequest, ServletResponse)` aufrufen, um die Ausgabe von `/index.html` in die eigene Antwort einzufügen. Dieses Verfahren bietet sich insbesondere dann an,

wenn eine Seite aus mehreren Elementen besteht. Oft ist es sinnvoll, immer gleiche Seitenbestandteile wie Header und Footer in Extradateien zu schreiben und diese dann von Servlets einfügen zu lassen:

```
public void service(ServletRequest req, ServletResponse res)
      throws IOException, ServletException {
   res.setContentType("text/html");
   getServletContext().getRequestDispatcher("/header.html").include(req, res);
   printBody(...);
   getServletContext().getRequestDispatcher("/footer.html").include(req, res);
}
```

*Listing 3.13: `RequestDispatcher.include()`-Codebeispiel*

Um einen reibungslosen Ablauf zu garantieren, können alle eingefügten Ressourcen auf den `Writer` und den `ServletOutputStream` des `ServletResponse`-Objekts zugreifen. Jegliche gesetzten Kopfdaten werden dabei ignoriert.

Manchmal ist es sinnvoll, von einem eingebundenen Servlet aus auf die Pfade zugreifen zu können, die zum Einbinden führten. Zu diesem Zweck werden die Pfade von der Servlet-Engine als Attribute im Request-Objekt hinterlegt. Die entsprechenden Attributsschlüssel heißen:

```
javax.servlet.include.request_uri
javax.servlet.include.context_path
javax.servlet.include.servlet_path
javax.servlet.include.path_info
javax.servlet.include.query_string
```

Im Vergleich zu `include(ServletRequest, ServletResponse)` hat die Methode `forward(ServletRequest, ServletResponse)` eine etwas andere Bedeutung. Sie kann benutzt werden, um einen Request tatsächlich weiterzuleiten und nicht bloß die Ausgabe einer anderen Ressource einzubinden. Dies ist nützlich, wenn beispielsweise der Logikteil eines Vorgangs von der Anzeige des Ergebnisses getrennt werden soll. Ein Servlet kann auf diese Weise Datenbankoperationen übernehmen, während ein anderes die formatierte Ausgabe von Datensätzen erledigt. Um die Datensätze zu übergeben, sollten sie mit der Methode `ServletRequest.setAttribute(String, Object)` an den Request gehängt werden:

```
public void service(ServletRequest req, ServletResponse res)
      throws IOException, ServletException {
   req.setAttribute("dbresult", getDBResult());
   getServletContext().getRequestDispatcher("/displayer").forward(req, res);
}
```

*Listing 3.14: `RequestDispatcher.forward()`-Codebeispiel*

Im Unterschied zu `include()` ermöglicht `forward()` es der aufgerufenen Ressource, Header-Felder zu setzen. Bedingung ist, daß das aufrufende Servlet zuvor noch keine

Daten endgültig an den Client gesendet hat. `ServletResponse.isCommitted()` muß also falsch sein. Ist das nicht der Fall, wird eine `IllegalStateException` ausgelöst.

## 3.10 Andere Länder, andere Zeichen

Die Internationalisierung von Produkten wird angesichts des globalen Marktes immer wichtiger. Mit der Version 2.2 des Servlet-APIs trägt Sun diesem Umstand verstärkt Rechnung.

Zunächst wollen wir uns dem Zeichensatz-Problem widmen. Nehmen wir einmal an, Sie haben einen hochdotierten Auftrag von einem japanischen Großunternehmen erhalten. Wie, werden Sie sich fragen, kann ich meine Ausgaben in japanischen Schriftzeichen kodieren? Der Zeichensatz wird leider nicht explizit, sondern implizit über die Methode `setContentType(String contenttype)` des Response-Objektes gesetzt. Hierzu wird nach dem MIME-Typ noch ein Zeichensatz angegeben. Wenn Sie also japanisch schreiben wollen, müssen sie folgenden Content-Type setzen:

```
text/html; charset=Shift_JIS
```

Entsprechendes gilt für alle anderen Zeichensätze. Wird kein Zeichensatz angegeben, entspricht dies automatisch ISO-8859-1 (Latin-1) – einem Zeichensatz, der für die meisten westeuropäischen Sprachen geeignet ist, Ihrem japanischen Kunden jedoch wahrscheinlich wenig zusagt. Nachdem Sie den Content-Type gesetzt haben, können Sie über `ServletResponse.getWriter()` einen entsprechenden `PrintWriter` erlangen.

Das Lesen kodierter Daten erfolgt hingegen vollautomatisch. Wenn Sie mit `ServletRequest.getReader()` einen Reader erlangt haben, sollte dieser die Daten korrekt dekodieren können, sofern dies vom Hersteller Ihrer Servlet-Engine richtig implementiert wurde.

Wenn Sie jetzt dem Client noch mitteilen wollen, daß Ihre Ausgabe deutsch oder englisch ist, müssen Sie den entsprechenden Header `Content-Language` setzen:

```
HttpServletResponse.setHeader("Content-Language", "de"); // deutsch
HttpServletResponse.setHeader("Content-Language", "en"); // englisch
```

Sie finden das umständlich? Dann kann Ihnen geholfen werden. Rufen Sie einfach die Methode `ServletRequest.setLocale(Locale loc)` mit dem `Locale` Ihrer Wahl auf, und die Servlet-Engine setzt automatisch den korrekten `Content-Language`-Header und Zeichensatz. Natürlich sollte dies geschehen, bevor Sie über `getWriter()` einen `Writer` anfordern.

## 3.11 RealLife-Servlets

Nachdem wir das Servlet-API nun hinreichend erklärt haben, möchten wir drei ausgesuchte Servlets vorstellen, um den realen Einsatz zu demonstrieren. Den Anfang macht ein Datenbank-Servlet, gefolgt von einem Mail-Servlet und einem Gästebuch.

### 3.11.1 Datenbank-Servlet

Unser Ziel soll es sein, eine sehr einfache Oberfläche für relationale Datenbanken zu entwickeln. Die einzige Anforderung an die Datenbank ist die Existenz eines JDBC-Treibers. Über ein Servlet soll der Nutzer in die Lage versetzt werden, SQL-Kommandos abzusetzen und sich deren Ergebnis anzeigen zu lassen.

Daraus folgt, daß das Servlet beim ersten Aufruf ein Eingabeformular anzeigen muß. Nach dem Abschicken des Formulars sollte das Ergebnis angezeigt werden. Zusätzlich wollen wir das Formular erneut anzeigen lassen, um sofort einen neuen Befehl absetzen zu können.

Soweit die offensichtlichen Anforderungen. Bei ihrer Umsetzung stoßen wir auf weitere. Typischerweise lassen Datenbanken nur eine begrenzte Zahl an Verbindungen zu. Außerdem stellt ein Verbindungsaufbau einen recht beachtlichen Aufwand dar. Daher soll unser Servlet für alle Anfragen nur eine Verbindung nutzen, die zur Initialisierung aufgebaut und beim Zerstören wieder abgebaut wird.

Damit haben wir implizit festgelegt, daß die Verbindungsparameter in den Servlet-Properties festgelegt werden müssen. Daher liest unser `JDBCServlet` (Listing 3.16) beim Start die Parameter `driverClass`, `url`, `user` und `password`. Die entsprechenden Einträge in der `servlet.properties`-Datei sehen so aus:

```
...
# JDBC-Servlet
jdbc.code=de.webapp.Examples.Servlet.JDBCServlet
jdbc.initArgs=user=einNutzer,password=seinPasswort,url=jdbc:odbc:solid,driverClass=su
n.jdbc.odbc.JdbcOdbcDriver
...
```

*Listing 3.15: `servlet.properties`-Ausschnitt für ein `JDBCServlet`, das über die ODBC-Bridge von Sun mit der ODBC-Datenquelle »solid« verbunden ist.*

Mit Hilfe dieser Daten versucht das Servlet, in der `init()`-Methode eine Verbindung aufzubauen. Gelingt dies, wird die Verbindung für den späteren Gebrauch in der Instanzvariablen `myConnection` hinterlegt. Geht etwas schief, wird eine `ServletException` ausgelöst.

Anschließend fragt das Servlet die Verbindung nach aufgetretenen `SQLWarnings`, die gegebenenfalls ins Log geschrieben werden. Ebenso werden die Verbindungs-Metadaten in Erfahrung gebracht und in der Instanzvariablen `myDatabaseMetaData` hinterlegt. Jetzt ist das `JDBCServlet` betriebsbereit.

Bei jedem folgenden Aufruf des Servlets überprüfen wir, ob eine Anfrage mitgesandt wurde. Ist das nicht der Fall, geben wir einfach nur den HTML-Kopf, die Verbindungs-Metadaten (URL und Datenbanktreibername), das Formular sowie einen HTML-Footer aus. Wurde jedoch eine Anfrage mitgeschickt, rufen wir zwischen Metadaten und Formular noch die Methode `executeRequest` auf. Sie gibt zunächst einmal die Anfrage aus. Anschließend wird getestet, um was für eine Art von SQL-Ausdruck es sich handelt. JDBC unterscheidet nämlich prinzipiell zwischen lesenden, schreibenden und sonstigen Zugriffen. Ist geklärt, um welche Art von Befehl es sich handelt, wird er mit der entsprechenden Methode ausgeführt.

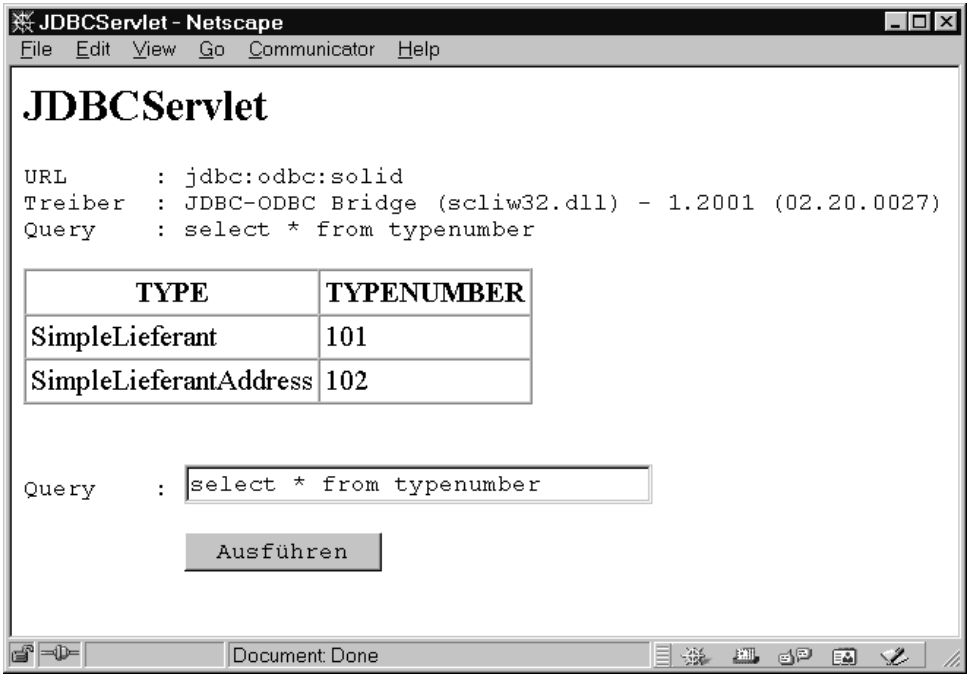


*Abbildung 3.6: Screenshot JDBC-Servlet*

Damit der Nutzer auch informiert ist, wird eine passende Rückmeldung ausgegeben. Im Falle eines `SELECT`-Ausdrucks handelt es sich dabei um eine Tabelle (Abbildung 3.6). Bei schreibenden Zugriffen wird gemeldet, wie viele Datensätze geändert wurden. Das Ausführen anderer Befehle resultiert in einer Erfolgs- beziehungsweise Mißerfolgsmeldung.

Da wir mit der Datenbankverbindung eine wertvolle Ressource nutzen, müssen wir hinter uns aufräumen, wenn das Servlet zerstört wird. Daher schließt `destroy()` die Datenbankverbindung und setzt die beiden Instanzvariablen `myConnection` und `myData-`

`baseMetaData` auf `null`. Um sicherzugehen, daß dies auch geschieht, falls beim Schließen der Verbindung etwas schiefgeht, befinden sich die beiden letzten Zuweisungen in einem `finally`-Block.

```
package de.webapp.Examples.Servlet;

import java.io.*;
import java.util.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.math.BigDecimal;

public class JDBCServlet extends HttpServlet {

/** Hält die Datenbankverbindung */
protected Connection myConnection = null;
/** Datenbank-Metadaten */
protected DatabaseMetaData myDatabaseMetaData = null;
/** Servlet-Info */
public String getServletInfo() {
   return "JDBCServlet";
}

public void init(ServletConfig aConfig)
      throws ServletException {
   super.init(aConfig);
   String url = getInitParameter("url");
   String driverClass = getInitParameter("driverClass");
   String user = getInitParameter("user");
   String password = getInitParameter("password");
   if (url == null
      || driverClass == null
      || user == null
      || password == null)
         throw new ServletException("Missing initparameter\nURL: "
               + url + "\nDriverClass: " + driverClass + "\nUser: "
               + user + "\nPassword: " + password);
   try {
      // Treiber laden.
      Class.forName(driverClass);
      // Verbindung aufbauen.
      myConnection = DriverManager.getConnection(url, user, password);
      // Warnungen ausgeben.
      checkForWarning(myConnection.getWarnings());
      // Metadaten holen.
      myDatabaseMetaData = myConnection.getMetaData();
   }
   catch (Exception e) {
      throw new ServletException(e.toString());
   }
```

```
}

public void doGet(HttpServletRequest req, HttpServletResponse res)
         throws ServletException, IOException {
   res.setContentType("text/html");
   PrintWriter writer = res.getWriter();
   printHTMLHeader(writer);
   printMetaData(writer);
   String query = req.getParameter("query");
   if ( query != null
      && (!query.equals(""))) {
         executeQuery(writer, query);
   }
   printRequestForm(writer, req.getRequestURI());
   printHTMLFooter(writer);
}

protected void printRequestForm(PrintWriter writer, String servletPath)
      throws IOException {
   // Eventuell vorhandenen QueryString entfernen.
   int i = servletPath.indexOf('?');
   if (i != -1) servletPath = servletPath.substring(0, i);
   writer.println("<form method='get' action='" + servletPath + "'>");
   writer.println("Query    : <input type=text size=30 name='query'>\n");
   writer.println("           <input type=submit value='Ausführen'>");
   writer.println("</form>");
}

protected void printMetaData(PrintWriter writer)
      throws IOException {
   try {
      if (myDatabaseMetaData != null) {
         writer.println("URL      : " + myDatabaseMetaData.getURL());
         writer.print("Treiber  : " + myDatabaseMetaData.getDriverName());
         writer.println(" - " + myDatabaseMetaData.getDriverVersion());
      }
   }
   catch (SQLException ex) {
      printException(ex, writer);
   }
}

protected void printHTMLHeader(PrintWriter writer)
      throws IOException {
   writer.println("<html>");
   writer.println("<head><title>JDBCServlet</title></head>");
   writer.println("<body>");
   writer.println("<h2>JDBCServlet</h2>");
   writer.println("<pre>");
}
```

```
protected void printHTMLFooter(PrintWriter writer)
      throws IOException {
   writer.println("</pre>");
   writer.println("</body>");
   writer.println("</html>");
}

/** Führt die Anfrage aus. */
protected void executeQuery(PrintWriter writer, String query)
      throws IOException {
   // Zeige Query.
   writer.println("Query    : " + query);
   try {
      Statement statement = myConnection.createStatement();
      query = query.trim();
      String lowerQuery = query.toLowerCase();
      if (lowerQuery.startsWith("select")) {
         showResultSet(statement.executeQuery(query), writer);
      }
      else if (lowerQuery.startsWith("insert")
         || lowerQuery.startsWith("delete")
         || lowerQuery.startsWith("update")) {
            writer.println(statement.executeUpdate(query)
               + " Datensätze wurden erfolgreich geändert.");
      }
      else {
         writer.println("Ausführung "
            +(statement.execute(query)?"war erfolgreich.":"ist fehlgeschlagen."));
      }
      statement.close();
   }
   catch (Exception ex) {
      printException(ex, writer);
   }
}

private void checkForWarning(SQLWarning warn)
      throws SQLException {
   if (warn != null) {
      log("*** Eine Warnung ist aufgetreten ***");
      while (warn != null) {
         log("SQLState: " + warn.getSQLState()
            + "\nMessage : " + warn.getMessage()
            + "\nVendor  : " + warn.getErrorCode());
         warn = warn.getNextWarning();
      }
   }
   else {
      log("*** Datenbankverbindung aufgebaut ***");
   }
}
```

```
/** Anzeige des Resultats als Tabelle. */
private void showResultSet(ResultSet rs, PrintWriter writer)
      throws SQLException, IOException {
   if (rs == null) return;
   // Aus den Metadaten werden die Beschreibungen der Spalten gewonnen.
   ResultSetMetaData rsmd = rs.getMetaData();
   int numCols = rsmd.getColumnCount();
   writer.println("</pre>");
   writer.println("<table border=1 cellspacing=0 cellpadding=3>");
   writer.println("<tr>");
   // Tabellenüberschriften ausgeben.
   for (int i=1; i<=numCols; i++) {
      writer.println("<th>" + rsmd.getColumnLabel(i) + "</th>");
   }
   writer.println("</tr>");
   // Daten ausgeben...
   while (rs.next()) {
      writer.println("<tr>");
      // ...für jede Resultatzeile:
      for (int i=1; i<=numCols; i++) {
         formatElement(rs, rsmd.getColumnType(i), writer, i);
      }
      writer.println("</tr>");
   }
   writer.println("</table>");
   writer.println("<pre>");
   rs.close();
}

/** Anzeige eines Elements */
private void formatElement(ResultSet rs, int dataType,
      PrintWriter writer, int col)
         throws SQLException, IOException {
   // Nun formatiere die Daten je nach Datentyp.

   switch (dataType) {
      case Types.BINARY:
      case Types.VARBINARY:
      case Types.LONGVARBINARY:
         byte[] binary = rs.getBytes(col);
         writer.println("<td>" + new String(binary, 0) + "</td>");
         break;
      default:
         writer.println("<td>" + rs.getObject(col).toString() + "</td>");
   }
}
/** Schließt die verwandte Datenbankverbindung */
public void destroy() {
   try {
      if (myConnection != null) {
         myConnection.close();
```

```
            log("*** Die Datenbankverbindung '" + myConnection.toString()
               + "' wurde geschlossen ***");
         }
      }
      catch (Exception e) {
         log("*** Fehler beim Schließen der Datenbankverbindung: "
            + e.toString() + " ***");
      }
      finally {
         myConnection = null;
         myDatabaseMetaData = null;
      }
   }

   private void printException(SQLException ex, PrintWriter writer) {
      writer.println("*** SQLException aufgetreten ***<p>");
      while (ex != null) {
         writer.println("SQLState: " + ex.getSQLState() + "<br>");
         writer.println("Message: " + ex.getMessage() + "<br>");
         writer.println("Vendor: " + ex.getErrorCode() + "<br>");
         ex = ex.getNextException();
      }
   }

   private void printException(Exception ex, PrintWriter writer) {
      writer.println("*** Exception aufgetreten ***<p>");
      writer.println(ex.toString());
   }
}
```

*Listing 3.16: Die Klasse `JDBCServlet`*

### 3.11.2 Mail-Servlet

Feedback-Formulare sind schon lange ein Muß auf jeder Website. Oft werkelt im Hintergrund ein kleines Skript, das mit einem Mailserver kommuniziert und die Kommentare per E-Mail an den zuständigen Mitarbeiter weiterleitet. Natürlich läßt sich so etwas auch leicht mit einem Servlet erledigen.

Unser Beispiel-Servlet (Listing 3.17) soll eine etwas allgemeinere Aufgabe erledigen, nämlich über einen beliebigen SMTP-Host (Simple Mail Transfer Protocol) Mails versenden. Daraus ergibt sich die Notwendigkeit, ein paar Daten vom Nutzer in Erfahrung zu bringen: SMTP-Host, Absender, Adresse, Thema und Nachricht. Um die Angelegenheit ein bißchen komfortabler zu gestalten, sollen die fünf Parameter dem Servlet optional als Initialisierungsargumente übergeben werden können.

```
package de.webapp.Examples.Servlet;

import java.io.*;
import java.util.*;
```

```
import javax.servlet.http.*;
import javax.servlet.*;
import javax.mail.*;
import javax.mail.internet.*;

/** Servlet, das über SMTP Mails versenden kann. */
public class SMTPServlet extends HttpServlet {

/** Standard-SMTP-Host */
protected String myHost = null;

/** Standard-to-Adresse */
protected String myTo = null;

/** Standard-from-Adresse */
protected String myFrom = null;

/** Standardthema */
protected String mySubject = null;

/** Standardnachricht*/
protected String myBody = null;

/** Setzt die Standardwerte */
public void init(ServletConfig aConfig)
      throws ServletException {
   super.init(aConfig);
   myHost = getInitParameter("host");
   myTo = getInitParameter("to");
   myFrom = getInitParameter("from");
   mySubject = getInitParameter("subject");
   myBody = getInitParameter("body");
}

/** Gibt das Formular zum Eingeben der Nachricht aus. */
public void doGet(HttpServletRequest req, HttpServletResponse res)
         throws ServletException, IOException {
   res.setContentType("text/html");
   PrintWriter   writer = res.getWriter();
   String actionPath = removeQueryString(req.getRequestURI());
   String host = req.getServerName();
   writer.println("<html>");
   writer.println("<head>\n<title>SMTPServlet</title>\n</head>");
   writer.println("<body>");
   writer.println("<h2>SMTPServlet</h2>");
   writer.println("<pre>");
   writer.println("<form action='" + actionPath + "' method='post'>");
   if (myHost == null)
      writer.println("Host   : <input name='host' type=text maxlength=100 size=50
value='" + host + "'>");
   if (myFrom == null)
      writer.println("From   : <input name='from' type=text maxlength=100 size=50>");
```

```
   if (myTo == null)
      writer.println("To     : <input name='to' type=text maxlength=100 size=50>");
   if (mySubject == null)
      writer.println("Subject: <input name='subject' type=text maxlength=100
size=50>");
   if (myBody == null)
      writer.println("         <textarea name='body' wrap='soft' cols='50'
rows='10'></textarea>\n");
   writer.println("         <input type='submit' value='Senden'>");
   writer.println("</form>");
   writer.println("</pre>");
   writer.println("</body>");
   writer.println("</html>");
}

/** Schickt die Mail */
public void doPost(HttpServletRequest req, HttpServletResponse res)
         throws ServletException, IOException {
   res.setContentType("text/html");

   // Die eigene Adresse
   String actionPath = removeQueryString(req.getRequestURI());
   PrintWriter   writer = res.getWriter();

   // HTML-Header etc. schreiben.
   writer.println("<html>\n<head>\n<title>SMTPServlet</title>\n</head>");
   writer.println("<body>");
   writer.println("<h2>SMTPServlet</h2>");

   // Parameter einlesen.
   String from = myFrom==null?req.getParameter("from"):myFrom;
   String to = myTo==null?req.getParameter("to"):myTo;
   String subject = mySubject==null?req.getParameter("subject"):mySubject;
   String body = myBody==null?req.getParameter("body"):myBody;
   String host = myHost==null?req.getParameter("host"):myHost;
   if (host == null)
      host = req.getServerName();
   if (  from == null
      || to   == null
      || subject == null
      || body == null) {
         writer.println("Mindestens ein benötigter Parameter fehlt.<p>");
   }
   else {
      // Mail-Session besorgen.
      Properties props = new Properties();
      props.put("mail.smtp.host", host);
      Session session = Session.getDefaultInstance(props, null);
      try {
         // Message-Objekt zusammenbasteln.
         StringTokenizer st = new StringTokenizer(to, " ,;");
         int length = st.countTokens();
```

```
            InternetAddress[] toAddresses = new InternetAddress[length];
            for (int i=0; st.hasMoreTokens(); i++) {
               toAddresses[i] = new InternetAddress(st.nextToken());
            }
            Message msg = new MimeMessage(session);
            msg.setFrom(new InternetAddress(from));
            msg.setSubject(subject);
            msg.setRecipients(Message.RecipientType.TO, toAddresses);
            msg.setContent(body, "text/plain");

            // Senden
            Transport.send(msg);
            writer.println("Die Nachricht wurde versandt.<p>");
         }
         catch (MessagingException mex) {
            writer.println("Beim Senden der Nachricht trat ein Fehler
auf:<br><blockquote>"
               + mex.toString() + "</blockquote><p>");
         }
      }
      writer.println("<a href='" + actionPath + "'>Neue Nachricht</a>");
      writer.println("</body>\n</html>");
   }

   protected static String removeQueryString(String aURI) {
      int i = aURI.indexOf('?');
      if (i != -1) aURI = aURI.substring(0, i);
      return aURI;
   }

} // Ende der Klasse
```

*Listing 3.17: Die Klasse* `SMTPServlet`

Die Initialisierungsargumente befinden sich in der `servlet.properties`-Datei (Listing 3.18). Entsprechend liest unser Servlet während der Initialisierung über `init()` die Parameter mit den Namen `host`, `to`, `from`, `subject` und `body` und weist die Werte den entsprechenden Instanzvariablen zu.

```
...
# Mail-Servlet
mail.code=de.webapp.Examples.Servlet.SMTPServlet
mail.initArgs=host=mein.host.de

# Feedback-Servlet
feedback.code=de.webapp.Examples.Servlet.SMTPServlet
feedback.initArgs=host=mein.host.de,from=webapp.nutzer,to=webapp@webapp.de,subject=Kr
itik
...
```

*Listing 3.18: Möglicher* `servlet.properties`*-Ausschnitt für* `SMTPServlet`

Wird nun das Servlet über die `GET`-Methode aufgerufen, erscheint im Browser des Nutzers ein Formular, in das die fehlenden Angaben einzutragen sind (Abbildung 3.7). Ein Klick auf den Sendeschalter schickt das ausgefüllte Formular mittels `POST` an das Servlet zurück. Die Aufgaben im Servlet sind also klar verteilt: `doGet()` dient lediglich zur Anzeige, `doPost()` erledigt die eigentliche Arbeit.

Nachdem die üblichen HTML-Zeilen geschrieben wurden, liest das Servlet die fünf benötigten Parameter ein. Falls kein Standardwert vorgegeben ist, wird über `HttpServletRequest.getParameter()` auf Werte der Anfrage zurückgegriffen. Außerdem wird der Host gleich dem Servernamen gesetzt, falls er weder in der Anforderung noch in den Standardwerten festgelegt wurde. Ist danach immer noch ein Parameter unbelegt, wird eine Fehlermeldung an den Browser zurückgeschickt.

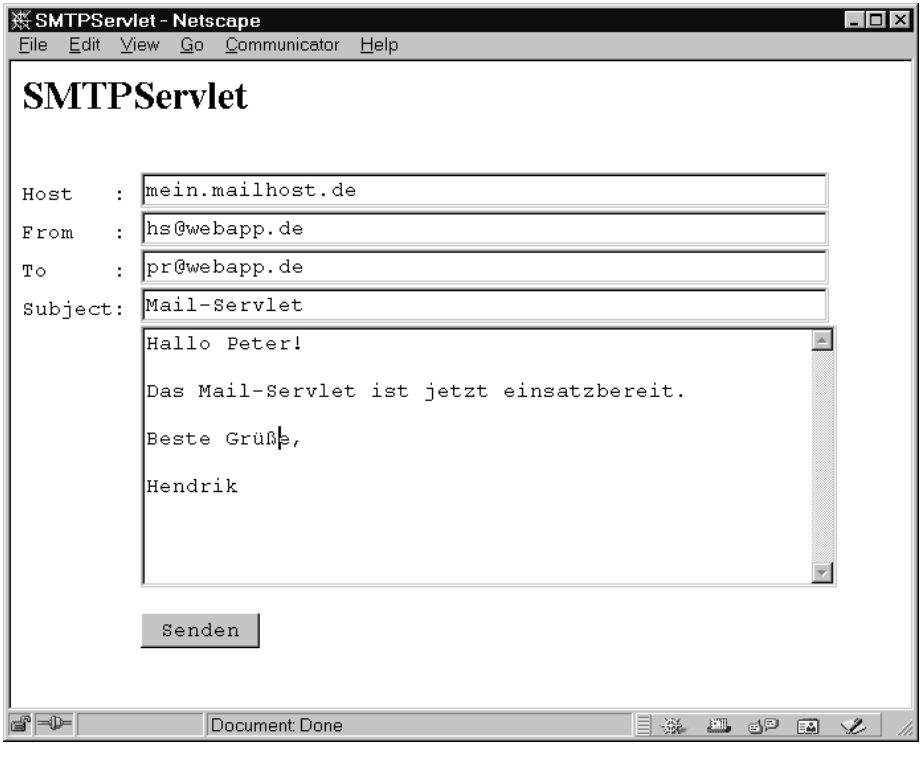


*Abbildung 3.7: Mail-Servlet, das über keinerlei Vorinitialisierung verfügt*

Sofern bis hierhin alles gut verlaufen ist, wird ein Mail-Session-Objekt für den angegebenen Host erzeugt und ein Message-Objekt erzeugt und gesendet. Mit der Unterstützung des JavaMail-API (erhältlich von JavaSoft: `http://www.javasoft.com`) ist das sehr

unkompliziert – es müssen nur die restlichen vier Parameter an das Message-Objekt übergeben werden. Anschließend wird nur noch `Transport.send(Message)` aufgerufen.

Je nachdem wie viele Parameter vorgegeben wurden, läßt sich das Servlet so als Universal-Mailer oder Feedback-Formular einsetzen.

### 3.11.3 Gästebuch

Sie sind die Poesiealben des WWW: Gästebücher – vornehmlich auf privaten Homepages zu finden und meistens mit viel sinnlosem Zeug gefüllt. Trotzdem erfreuen sie sich ungebrochener Beliebtheit und zählen zu den Standardanwendungen im Netz. Natürlich sind sie auch mit Servlets zu verwirklichen.

Unser Gästebuch soll über alle üblichen Fähigkeiten verfügen. Ein Formular lädt zum Eintragen eines Kommentars ein, dabei sind Name und E-Mail-Adresse mit anzugeben (Abbildung 3.8). Auf derselben Seite sollen bereits die 25 aktuellsten Einträge des Gästebuchs angezeigt werden – selbstverständlich in chronologischer Reihenfolge. Ein Eintrag sollte neben dem Namen und der Adresse des Autors einen Zeitstempel und eine Nummer aufweisen.

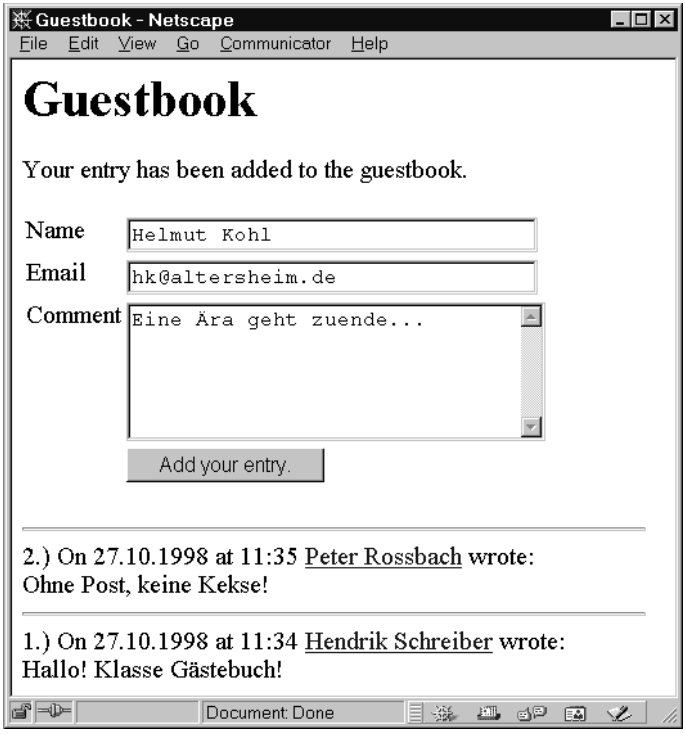


*Abbildung 3.8: Ausgabe des Gästebuch-Servlets*

Damit unser Gästebuch auch nach einem Neustart des Servers noch alle Einträge enthält, müssen wir das Buch speichern. Da es sich um eine überschaubare Datenmenge handelt und Gästebücher normalerweise nicht über eine Suchfunktion verfügen, verzichten wir auf den Einsatz einer Datenbank und speichern die Daten auf der lokalen Festplatte. Am einfachsten geht dies über den Serialisierungsmechanismus von Java.

Beim Start von `Guestbook` (Listing 3.20) muß also als erstes ein Gästebuch geladen werden – vorausgesetzt, es existiert schon eins. Damit verschiedene Instanzen des Servlets verschiedene Bücher anzeigen können, übergeben wir den Dateinamen als Initialisierungsargument `file` (Listing 3.19) und speichern ihn in der Instanzvariablen `myFile`. Nun müssen wir die Datei nur noch lesen. Dazu öffnen wir in der `init()`-Methode einen `ObjectInputStream` und lesen den Vektor `myEntries`. In diesem Vector werden später auch die neuen Kommentare eingetragen.

```
...
# Guestbook-Beispiel
guestbook.code=de.webapp.Examples.Servlet.Guestbook
guestbook.initparams=file=johns-guestbook.serialized
...
```

*Listing 3.19: Möglicher `servlet.properties`-Ausschnitt für `Guestbook`*

Soll das Servlet zerstört werden, müssen wir zuvor das Buch speichern. Zu diesem Zweck öffnen wir in der Methode `destroy()` wieder unsere Gästebuchdatei `myFile` und schreiben den Vektor `myEntries` über einen `ObjectOutputStream` in diese Datei. Damit das funktioniert, müssen die Elemente des Vektors die Schnittstelle `java.io.Serializable` implementieren.

Was sind das aber für Elemente? Es bietet sich an, jeden Eintrag in einem Objekt zu kapseln. Dies führt uns zu der Klasse `GuestbookEntry`. Sie verfügt über die relevanten Attribute Name, E-Mail, Kommentar sowie Erstellungsdatum und ist außerdem in der Lage, diese über `toHTMLString()` in HTML auszugeben. Zudem implementiert die Klasse die Schnittstelle `Serializable`.

Somit haben wir das Laden und Speichern geregelt. Jetzt müssen wir nur noch das Service-Verhalten erklären. Genau wie im `SMTPServlet` lassen wir die `GET`-Methode die simple Anzeige erledigen und die `POST`-Methode die Arbeit übernehmen. Entsprechend werden in `doGet()` nur das Eingabeformular und die Einträge des Gästebuchs angezeigt. Hierzu werden die Methoden `printForm(PrintWriter, String)` und `printEntries(PrintWriter, HttpServletRequest)` aufgerufen.

Im Formular wird als `action`-Attribut der URI des Servlets angegeben und als Methode `POST` festgelegt. Obwohl die Methode `getRequestURI()` laut Spezifikation den URI ohne Querystring zurückgibt, ist diese Methode in einigen Engines fehlerhaft implementiert (zum Beispiel in JRun 2.1). Daher muß das Servlet die Hilfsmethode `removeQueryString(String)` aufrufen, um kompatibel zu sein.

Die Methode `printEntries()` sorgt neben der eigentlichen Anzeige noch für einen Blätter-Mechanismus, falls das Buch zu viele Einträge enthält, um sie auf einer Seite anzuzeigen. Die maximale Anzahl der Einträge pro Seite ist in der Instanzvariablen `entriesPerPage` auf 25 festgelegt. Um den Mechanimus zu realisieren, müssen wir wissen, von welchem Eintrag an die nächsten 25 Einträge angezeigt werden sollen. Dieser `offset` muß also bei jedem Aufruf des Servlets übergeben werden. Wird er nicht übergeben, wird wieder bei null angefangen. Um zu blättern, muß ein Verweis mit einem entsprechend höheren oder niedrigeren `offset` angezeigt werden. Dies geschieht in der Methode `printEntries` direkt vor der eigentlichen Ausgabe der Einträge, die in einer simplen Schleife erfolgt.

Am Code von `printEntry` fällt auf, daß er bezüglich des Vektors `myEntries` synchronisiert ist. Dies geschieht, um zu verhindern, daß der Vektor während der Ausgabe von anderen Threads verändert wird und so den Anzeige-Algorithmus durcheinanderbringt.

Die Methode `doPost()` wird aufgerufen, wenn das per `doGet()` erhaltene Formular vom Client abgeschickt wurde. Sie hat die Aufgabe, dem Gästebuch einen Eintrag hinzuzufügen. Zu diesem Zweck wird die Methode `addEntry(PrintWriter, HttpServletRequest)` aufgerufen. Dort werden die übergebenen Parameter gelesen, und es wird versucht, ein `GuestbookEntry`-Objekt zu instantiieren. Da der Konstruktor von `GuestbookEntry` bei fehlerhaften oder fehlenden Parametern eine `IllegalArgumentException` auslöst, ist es leicht, im Fehlerfall eine entsprechende Meldung auszugeben. Geht alles glatt, fügen wir den Eintrag dem Buch hinzu und zeigen Formular und Einträge an.

```
package de.webapp.Examples.Servlet;

import javax.servlet.http.*;
import javax.servlet.*;

import java.io.*;
import java.util.*;

import java.text.*;

public class Guestbook extends HttpServlet {

/** Gästebucheinträge */
protected Vector myEntries = new Vector();
/** Gästebuchdatei */
protected File myFile;
/** Einträge pro Seite */
protected int entriesPerPage = 25;

public void init(ServletConfig aConfig)
      throws ServletException {
   super.init(aConfig);
```

```
   // Dateinamen lesen.
   String myFilename = getInitParameter("file");
   // Default setzen.
   if (myFilename == null) {
      myFilename = new String("guestbook.serialized");
   }
   myFile = new File(myFilename);

   // Gästebuch lesen, falls die Datei existiert.
   if (myFile.exists()) {
      try {
         FileInputStream fileIn = new FileInputStream(myFile);
         ObjectInputStream objectIn = new ObjectInputStream(fileIn);
         myEntries = (Vector)objectIn.readObject();
         objectIn.close();
      }
      catch (Exception e) {
         log("Failed to read " + myFilename + ". " + e);
      }
   }
}

public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   // MIME-Typ gleich "text/html" setzen.
   res.setContentType("text/html");
   PrintWriter out = res.getWriter();
   printHeader(out);
   printForm(out, req.getRequestURI());
   printEntries(out, req);
   printFooter(out);
}

public void doPost(HttpServletRequest req, HttpServletResponse res)
      throws ServletException, IOException {
   // MIME-Typ gleich "text/html" setzen.
   res.setContentType("text/html");
   PrintWriter out = res.getWriter();
   printHeader(out);
   addEntry(out, req);
   printFooter(out);
}

public void destroy() {
   try {
      FileOutputStream fileOut = new FileOutputStream(myFile);
      ObjectOutputStream objectOut = new ObjectOutputStream(fileOut);
      objectOut.writeObject(myEntries);
      objectOut.flush();
      objectOut.close();
   }
```

```
   catch (Exception e) {
      log("Failed to write guestbook to " + myFile.toString() + ". " + e.toString());
   }
   super.destroy();
}

protected void printHeader(PrintWriter out) throws IOException {
   out.println("<html>");
   out.println("<title>Guestbook</title>");
   out.println("<body>");
   out.println("<h1>Guestbook</h1>");
}

protected void printFooter(PrintWriter out) throws IOException {
   out.println("</body>");
   out.println("</html>");
}

protected void printForm(PrintWriter out, String submituri) throws IOException {
   int offset
   try {
      offset = Integer.parseInt(req.getParameter("offset"));
   }
   catch (NumberFormatException nfe) {
      offset = 0;
   }
   out.println("<form method='post' action='" + removeQueryString(submituri) + "'>");
   out.println("<input type=hidden name='offset' value='" + offset + "'>");
   out.println("<table border=0>");
   out.println("<tr><td valign=top>Name</td>");
   out.println("<td><input type='text' name='name' size=30 maxlength=60></td></tr>");
   out.println("<tr><td valign=top>Email</td>");
   out.println("<td><input type='text' name='email' size=30 maxlength=80></td></
tr>");
   out.println("<tr><td valign=top>Comment</td>");
   out.println("<td><textarea name='comment' rows=5 cols=30 wrap='virtual'></
textarea></td></tr>");
   out.println("<tr><td> </td><td><input type='submit' value='Add your entry.'></
td></tr>");
   out.println("</table>");
   out.println("</form>");
}

protected void printEntries(PrintWriter out, HttpServletRequest req)
      throws IOException {
   synchronized (myEntries) {
      // Paging
      int offset;
      try {
         offset = Integer.parseInt(req.getParameter("offset"));
      }
```

```
      catch (NumberFormatException nfe) {
         offset = 0;
      }
      if (myEntries.size() > entriesPerPage) {
         if (offset > 0) {
            int previous = offset - entriesPerPage;
            out.println("<a href='" + removeQueryString(req.getRequestURI())
            + "?offset=" +  previous + "'>[previous page]</a> ");
         }
         if (offset + entriesPerPage < myEntries.size()) {
            int next = offset + entriesPerPage;
            out.println("<a href='" + removeQueryString(req.getRequestURI())
            + "?offset=" +  next + "'>[next page]</a>");
         }
      }
      // Einträge drucken.
      for (int i=offset; i<myEntries.size() && i < offset+entriesPerPage; i++) {
         GuestbookEntry entry = (GuestbookEntry)myEntries.elementAt(i);
         int number = myEntries.size()-i;
         out.println("<hr>");
         out.println(Integer.toString(number) + ".) " + entry.toHTMLString());
      }
   }
}

protected void addEntry(PrintWriter out, HttpServletRequest req)
      throws IOException {
   String name = req.getParameter("name");
   String email = req.getParameter("email");
   String comment = req.getParameter("comment");
   try {
      GuestbookEntry entry = new GuestbookEntry(name, email, comment);
      myEntries.insertElementAt(entry, 0);
      out.println("Your entry has been added to the guestbook.<p>");
      printForm(out, req.getRequestURI());
      printEntries(out, req);
   }
   catch(IllegalArgumentException iae) {
      out.println("At least one field is missing or incorrect.<br>");
      out.println("Please try again.<p>");
      printForm(out, req.getRequestURI());
   }
}
protected static String removeQueryString(String aURI) {
   int i = aURI.indexOf('?');
   if (i != -1) aURI = aURI.substring(0, i);
   return aURI;
}
} // Ende der Klasse

class GuestbookEntry implements Serializable {
```

```
/** Name des Eintragenden */
protected String myName;
/** E-Mail-Adresse des Eintragenden */
protected String myEmail;
/** Kommentar */
protected String myComment;
/** Erstellungsdatum */
protected Date myCreationDate;

/** No-Arg-Konstruktor */
public GuestbookEntry() {}

public GuestbookEntry(String name, String email, String comment) {
   if (name == null) throw new IllegalArgumentException();
   myName = name;

   if (email == null) throw new IllegalArgumentException();
   if (email.indexOf("@") == -1)  throw new IllegalArgumentException();
   myEmail = email;

   if (comment == null) throw new IllegalArgumentException();
   myComment = comment;

   myCreationDate = new Date();
}

public String toHTMLString() {
   SimpleDateFormat formatter = new SimpleDateFormat ("'On' dd.MM.yyyy 'at' HH:mm");
   return formatter.format(myCreationDate) + " <a href='mailto:"
      + myEmail + "'>" + myName + "</a> wrote:<br>" + myComment;
}

} // Ende der Klasse
```

*Listing 3.20: Guestbook-Servlet*

## 3.12 Sicherheit

Sofern Sie nicht nur simple Servlets, sondern echte Applikationen programmieren wollen, kommen Sie um das Thema Sicherheit nicht herum. Üblicherweise hat dieses Thema vier Aspekte:

- Authentifikation: Ein Nutzer muß sich gegenüber dem Server ausweisen beziehungsweise umgekehrt.
- Zugangskontrolle: Der Zugang zu Ressourcen ist nur einem bestimmten Benutzerkreis erlaubt.

- Integrität: Es wird sichergestellt, daß die Daten während der Übertragung in keiner Weise verändert wurden.
- Vertraulichkeit: Es wird sichergestellt, daß die Daten während der Übertragung nicht von Dritten belauscht werden können.

Alle vier Punkte finden sich in der ein oder anderen Art in Methoden des API oder im Deployment-Deskriptor (siehe Anhang C) wieder. Zunächst wollen wir uns die Möglichkeiten der Authentifizierung ansehen.

### 3.12.1 Authentifikation

Würden Sie jedem Ihre Kreditkartennummer geben? Natürlich nicht. Kreditkartennummern gibt man nur Personen oder Firmen, denen man vertraut oder die einem zumindest vertrauenswürdig erscheinen. Im Netz ist das ähnlich, nur leider kann man sein Gegenüber nicht sehen. Um sicherzugehen, daß man seine Kreditkartennummer an jemanden weitergibt, dem man vertraut, muß dieser sich ausweisen, sich authentifizieren. Umgekehrt ist es bei Programmen mit Login. Damit Sie das Recht haben, diese zu benutzen, müssen Sie dem System mit einem Paßwort beweisen, daß Sie derjenige sind, der Sie vorgeben zu sein. Authentifizierung ist also eine Grundvoraussetzung für Handel und Computersicherheit. Das Servlet-API unterstützt drei Authentifizierungsverfahren:

- HTTP-Basic-Authentifikation
- HTTP-Digest-Authentifikation
- HTTPS-Client-Authentifikation

Die HTTP-Basic- und die HTTP-Digest-Authentifikation sind beide in HTTP/1.1 (RFC 2617) definiert. Während die Basic-Variante Paßwörter quasi im Klartext übermittelt und leicht belausch- und manipulierbar ist, verschlüsselt die Digest-Version das Paßwort und ist schwer zu manipulieren. Belauschbar ist sie trotzdem. Zudem wird die Digest-Authentifikation noch nicht von allen gängigen Browsern unterstützt.

Wenn Sie sich vor Lauschern schützen wollen, sollten Sie zu anderen Verfahren wie beispielsweise Virtual Private Networks (VPN) oder HTTPS (HTTP über SSL) greifen. Die Verschlüsselung ist sicher genug, um sich auf ansonsten unsichere Verfahren wie die HTTP-Basic-Authentifikation verlassen zu können. Ist Ihnen sogar das Eintippen von Login und Paßwort zu unsicher, können Sie im Fall von SSL auch ein X.509-Zertifikat einfordern, das den Client eindeutig ausweist. Dieses Verfahren wird HTTPS-Client-Authentifikation genannt. Im Fall einer SSL-Verbindung unter Java 2 befindet sich dann ein Objekt des Typs `java.security.cert.X509Certificate` hinter dem Request-Attribut `javax.servlet.request.X509Certificate`.

### 3.12.2 Programmgesteuerte Sicherheit

Auch nach geglückter Authentifikation hat sich das Thema Sicherheit in der Regel noch nicht erledigt. Sie wissen jetzt zwar, wer am anderen Ende der Leitung sitzt. Damit ist aber noch nicht geregelt, was diese Person für Rechte hat. Grundsätzlich gibt es zwei Möglichkeiten dies zu tun: Durch Konfiguration oder durch Maßnahmen schon bei der Programmierung. Letztere werden durch Methoden ermöglicht, die `HttpServletRequest` zur Verfügung stellt. Diese Methoden heißen:

```
String getRemoteUser()
boolean isUserInRole(String role)
java.Security.Principal getUserPrincipal()
boolean isSecure()
```

Lassen Sie uns diese Methoden genauer anschauen. Sofern sich der Nutzer in irgendeiner Weise authentifiziert hat, gibt `getRemoteUser()` seinen Login-Namen zurück. Hat er sich nicht authentifiziert, ergibt die Methode `null`. In diesem Fall gibt auch `getUserPrincipal() null` zurück und `isUserInRole() false`. Soviel zu den einfachen Fällen. Um die Methoden im Detail zu erklären, wollen wir zunächst ein paar Begriffe erläutern. Den Anfang macht die Rolle.

Eine Rolle kann man eindeutig identifizierbaren, handelnden Objekten zuweisen. Dies sind zum Beispiel Nutzer, Gruppen und auch Clients, die sich von einem bestimmten Netzwerksegment aus zu Ihrem Server verbinden. Ihnen lassen sich Rollen wie »Mitarbeiter«, »Administrator« oder »Programmierer« zuweisen. Genau wie im wirklichen Leben können Personen in verschiedene Rollen schlüpfen. Ein »Programmierer« kann theoretisch auch »Administrator« sein, was praktisch jedoch zu einem Rollenkonflikt führt – aber das ist wohl eher ein soziologisches Problem. Festzuhalten bleibt, daß Rollen mit unterschiedlichen Privilegien verknüpft sind. Ferner lassen sich Rollen sowohl auf einzelne Nutzer als auch auf Gruppen abbilden.

Das ergibt die schöne Situation, daß man bestehende Nutzer- und Gruppendaten zwar nutzen kann, um die Rechte innerhalb einer Web-Anwendung zu verwalten, sie jedoch nicht eins zu eins nutzen muß. Die Rolle bietet eine weitergehende Abstraktion. Dies ist insbesondere nützlich, wenn man verschiedene Web-Anwendungen installieren möchte, die sich durch unterschiedliche Nutzungsbeschränkungen auszeichnen.

In der Praxis weisen Sie bestehende Nutzer und Gruppen einer Rolle zu und räumen dieser Rolle bestimmte Rechte in Ihrer Anwendung ein. Während die erste Zuordnung herstellerabhängig in Ihrer Servlet-Engine erfolgt, können Sie die zweite Zuordnung im Deployment-Deskriptor vornehmen.

Bleibt der Begriff des `Principal` zu erklären. Man kann einen `Principal` am ehesten mit einer juristischen Person vergleichen. Vor dem Gericht kann eine GmbH oder ein Sportverein genauso wie eine Person behandelt werden, wie eine Person aus Fleisch und Blut. Genauso ist auch ein `Principal` anzusehen.

Hier ein Beispiel:

Hendrik und Peter verfügen beide über ein Login des Computer-Systems ihres Arbeitgebers. Da es sich um ein fortschrittliches, auf Java 2 aufbauendes System handelt, existiert für beide Personen ein Objekt mit der Schnittstelle `java.security.Principal`, das sie in der Anwendung repräsentiert. Nun wohnt Hendrik in Dortmund und Peter in Bochum, und wie der Zufall es will, sind die Privilegien für Personen aus Dortmund andere als die für Personen aus Bochum. Es gibt also eine Gruppe »Dortmunder« und eine Gruppe »Bochumer«. Beide Gruppen werden in der Anwendung wiederum von einem `Principal` repräsentiert. Im Verlauf einer Umstrukturierung beschließt die Firma, Teleworking einzuführen. Der »Administrator«, der anwendungsintern selbstverständlich auch durch ein `Principal`-Objekt repräsentiert wird, hat daher eine Web-Oberfläche auf Basis des Servlet-API 2.2 installiert. Es stellt sich heraus, daß Hendrik zum sinnvollen Arbeiten über das Web auch die Privilegien der Bochumer benötigt. Dies soll jedoch nur für den Web-Zugang gelten. Der »Administrator« weist also der Rolle »BochumerNetzzugang« die Gruppe »Bochumer« und den Nutzer »Hendrik« zu. Der Rolle »DortmunderNetzzugang« weist er nur die Gruppe »Dortmunder« zu.

Wird nun im Programm die Methode `getUserPrincipal()` aufgerufen, so wird das entsprechende `Principal`-Objekt zurückgegeben, sofern der Nutzer sich authentifiziert hat. Dabei wird immer das am genauesten passende `Principal`-Objekt zurückgegeben. Also im Falle von Peter nicht das »Bochumer«-Objekt, sondern das `Principal`-Objekt, das genau zu Peter paßt – das entsprechende Nutzer-Objekt also.

Machmal möchte man im Programm nicht nur prüfen, ob sich jemand rechtmäßig eingeloggt hat, sondern auch, in welcher Eigenschaft er gerade handelt. Dazu dient die Methode `isUserInRole()`. Sie gibt an, ob sich der Nutzer in einer bestimmten Rolle befindet.

Nehmen wir an, unser eben schon erwähnter »Administrator« sei nicht nur Herr über Absturz und Neustart, sondern gleichzeitig noch ein grundsolider »Bochumer«. Nun mögen »Bochumer« ehrliche Leute sein, manche Dinge sollten sie aber lieber dem »Administrator« überlassen, daher sind ihre Privilegien im Vergleich zu denen des »Administrators« eingeschränkt. Wenn sich nun der »Administrator« einloggt, so macht er dies eventuell auf einer Seite, die noch keine Extra-Privilegien erfordert. Wenn er dann aber zu einer Seite wechselt, die nur dem »Administrator« offensteht, muß das Programm prüfen, ob er berechtigt ist, diese zu nutzen. Dies könnte mit einem Aufruf von `HttpServletRequest.isUserInRole("Administrator")` geschehen.

Zu guter Letzt mag es sein, daß bestimmte Operationen des »Administrators« nur über ein Protokoll erlaubt sind, das gegen Lauschangriffe oder Manipulation gesichert ist. Um dies bereits durch die Programmierung abzusichern, können Sie sich der `isSecure()`-Methode des Requests bedienen. Sie gibt an, ob der Request über ein sicheres Protokoll übertragen wurde.

### 3.12.3 Deklarative Sicherheit

Nun ist das ständige programmgesteuerte Absichern eher mühsam. Daher gibt es mit dem Deployment-Deskriptor eine einfache und übersichtliche Möglichkeit, bestimmte Absicherungen einfach zu konfigurieren.

Zu diesem Zweck können Sie eine Sammlung von Ressourcen Ihrer Web-Anwendung angeben, für die bestimmte Sicherheitsbestimmungen gelten. Ein Beispiel dazu finden Sie im nächsten Abschnitt.

## 3.13 Web-Applikationen

Haben Sie schon einmal versucht, eine CGI-basierte Anwendung von einem Betriebssystem mit einem bestimmten Server zu einem anderen Betriebssystem auf einen anderen Server zu portieren? Sofern sich die Servlet-Engine-Hersteller an die Spezifikation halten, ist dies mit Servlets und JavaServer-Pages seit Version 2.2 des Servlet-API einfach. Möglich macht dies das Konzept der Web-Applikation. Es sieht vor, daß Servlets, JavaServer-Pages, Klassen, Dateien und andere Ressourcen in einem Web-Archiv (war) gebündelt und so sehr einfach auf einem 2.2-konformen System installiert werden können – sozusagen Plug&Play.

Die elementare Voraussetzung hierfür ist eine standardisierte Konfigurations-Möglichkeit. Diese ist mit dem Deployment-Deskriptor gegeben, der sich in der Datei `web.xml` im Verzeichnis `/WEB-INF` befindet.

Im übrigen hat ein Web-Archiv folgende Struktur:

```
/index.html
/umsatz.jsp
/feedback.jsp
/images/logo.gif
/images/anfahrtsskizze.gif
/WEB-INF/web.xml
/WEB-INF/lib/logic.jar
/WEB-INF/classes/com/meinefirma/servlets/MeinServlet.class
/WEB-INF/classes/com/meinefirma/util/MeineHelfer.class
```

Das Verzeichnis `/WEB-INF/lib/` enthält von der Anwendung benötigte Java-Archive (jar), und in `/WEB-INF/classes/` befinden sich alle übrigen Klassen oder Ressourcen, die im Klassenpfad der Anwendung sein müssen. Alle anderen Dateien des Web-Archivs sind Ressourcen, die dem Nutzer auf Anfrage hin zur Verfügung gestellt werden.

Im Deployment-Deskriptor sind folgende Informationen kodiert:

- Initialisierungsparameter des `ServletContextes`
- Konfiguration der Session

- Servlet- und Java-ServerPages-Definitionen
- Servlet- und Java-ServerPages-Abbildungen
- MIME-Typ-Abbildungen
- Willkommensdatei-Liste
- Fehler-Seiten
- Sicherheit

Eine genaue Definition des in XML spezifizierten Deskriptors (also die kommentierte DTD, d.h. Document Type Definition) finden Sie in Anhang C. Hier wollen wir beispielhaft nur auf einige wichtige Punkte eingehen. Wenn Sie dieses Buch lesen wird es hoffentlich ohnehin Werkzeuge mit grafischer Benutzeroberfläche geben, die Ihnen die Mühe ersparen, einen Deskriptor von Hand zu kodieren.

### 3.13.1 Minimaler Deployment-Deskriptor

Um wie in Abschnitt 3.3 ein `HelloWorld`-Servlet zu installieren, muß ein minimaler Deployment-Deskriptor folgendermaßen aussehen:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_1_2.dtd">
<web-app>
   <servlet>
      <servlet-name>HelloWorld</servlet-name>
      <servlet-class>de.webapp.Examples.Servlets.HelloWorld</servlet-class>
   </servlet>
   <servlet-mapping>
      <servlet-name>HelloWorld</servlet-name>
      <url-pattern>/HelloWorld</url-pattern>
   </servlet-mapping>
</web-app>
```

*Listing 3.21: Minimaler Deployment-Deskriptor für `HelloWorld`*

Wie in XML-Dokumenten üblich, wird in der ersten Zeile die DTD definiert. Diese Zeile ist also obligatorisch. Danach folgt die ebenfalls obligatorische Basis-Klammer aus dem Start-Tag `<web-app>` und dem End-Tag `</web-app>`. In dieser Klammer befindet sich die Definition Ihrer Web-Applikation. Genau wie in Abschnitt 3.3 definieren wir in `<servlet>...</servlet>` zunächst ein Servlet durch einen Namen und eine Klasse. Anschließend definieren wir eine Zuordnung des benannten Servlets zu einem URL-Muster (`<url-pattern>...</url-pattern>`). Im Gegensatz zu den vielfältigen Zuordnungen, die hoffentlich nicht mehr in Gebrauch sind, wenn Sie diese Zeilen lesen, ist diese Abbildung klar definiert.

- Ein URL-Muster, das mit einem Schrägstrich (»/«) beginnt und mit einem Schrägstrich, gefolgt von einem Stern (»/*«), endet, bezeichnet eine Pfad-Abbildung. Das bedeutet, daß auch URLs, die länger als das Muster sind, jedoch bis auf den Stern gleich beginnen, auf dieses Muster passen. Der Stern fungiert somit als Joker.
- Ein URL-Muster, das mit einem Stern, gefolgt von einem Punkt (»*.«), beginnt, stellt eine Erweiterungsabbildung dar. URLs, die abgesehen vom Stern genauso enden, passen auf dieses Muster.
- URL-Muster, die nur aus einem Schrägstrich (»/«) bestehen, passen, wenn keine andere Möglichkeit paßt (Default-Abbildung).
- Alle anderen Muster müssen exakt mit dem verlangten URL übereinstimmen, um zu passen.

Nun kann ein URL natürlich mehreren dieser Muster genügen. Daher gibt es eine definierte Reihenfolge, in der Servlet-Engines die Muster anwenden müssen. Zunächst wird nach einer exakten Übereinstimmung gesucht. Gibt es kein solches Muster, wird die längste gültige Pfadabbildung gesucht. Existiert auch eine solche nicht, wird nach einer gültigen Erweiterungsabbildung gesucht. Dazu wird allerdings nur der letzte Punkt innerhalb eines URL herangezogen. Andere Punkte innerhalb des URL werden ignoriert. Falls auch hier kein Treffer erzielt wird, kommt die Default-Abbildung zum Zuge. Diese ist zumeist durch das File-Servlet der Engine implizit vorgegeben.

### 3.13.2 Deployment-Deskriptor für Fortgeschrittene

Dieses Beispiel ist schon ein wenig komplexer.

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_1_2.dtd">
<web-app>
   <display-name>Eine einfache Applikation</display-name>
   <icon>
      <small-icon>/images/kleinesApplikationsSymbol.gif</smallicon>
      <large-icon>/images/grossesApplikationsSymbol.gif</large-icon>
   </icon>
   <description>Beschreibung einer Beispiel-Applikation, die eigentlich gar nichts
macht.</description>
   <context-param>
      <param-name>webmaster</param-name>
      <param-value>webmaster@webapp.de</param-value>
   </context-param>
   <servlet>
      <servlet-name>Beispiel</servlet-name>
      <servlet-class>de.webapp.Examples.Servlets.Beispiel</servlet-class>
      <init-param>
         <param-name>tollesBeispiel?</param-name>
         <param-value>jo!</param-value>
      </init-param>
```

```
    </servlet>
    <servlet-mapping>
        <servlet-name>Beispiel</servlet-name>
        <url-pattern>/Beispiel</url-pattern>
    </servlet-mapping>
    <mime-mapping>
        <extension>pdf</extension>
        <mime-type>application/pdf</mime-type>
    </mime-mapping>
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>
    <error-page>
        <error-code>404</error-code>
        <location>/errors/fileNotFound.html</location>
    </error-page>
    <error-page>
        <exception-type>java.io.FileNotFoundException</exception-type>
        <location>/errors/fileNotFound.html</location>
    </error-page>
</web-app>
```

*Listing 3.22: Beispielhafter Deployment-Deskriptor*

Beim Design des Deployment-Deskriptors ist insbesondere darauf Rücksicht genommen worden, daß er von Werkzeugen mit grafischer Benutzeroberfläche gut zu verarbeiten ist. Daher können Sie dem Deployment-Deskriptor Ihrer Web-Applikation leicht Metainformationen wie einen Namen (`<display-name>...</display-name>`) und eine Beschreibung (`<description>...</description>`) hinzufügen. Ebenso leicht ist es, der abstrakten Anwendung über Icons ein Gesicht zu geben (`<icon>...</icon>`). Neben diesen weichen, für das Funktionieren der Anwendung nicht notwendigen Parametern gibt es natürlich auch noch harte.

Um dem `ServletContext` Initialisierungsparameter zu übergeben, müssen Sie diese mit dem Tag `<context-param>...</context-param>` angeben. Ähnlich funktioniert dies auch mit Initialisierungsargumenten für Servlets.

Unser Beispiel beinhaltet Dateien im pfd-Format. Da wir nicht sicher sind, ob der Server den MIME-Typ für pdf kennt, müssen wir ihn registrieren. Dies geschieht mittels der Auszeichnung `<mime-mapping>...</mime-mapping>`.

Mit Hilfe von `<welcome-file-list>...</welcome-file-list>` können wir angeben, welche Dateien ausgeliefert werden sollen, wenn der Client nicht ausdrücklich eine anfordert. Die Reihenfolge der hier angegebenen Dateien entspricht dabei der Suchreihenfolge.

Zu guter Letzt kommen wir zu `<error-page>...</error-page>`. Sie bietet die Möglichkeit, sehr elegant anwendungsspezifisch mit Fehlern umzugehen. Alternativ können Sie entweder einen Ausnahmetyp (`<exception-type>...</exception-type>`) oder einen HTTP-Feh-

lercode (<error-code>...</error-code>) angeben. Tritt einer der beiden Fälle ein, wird anstelle der normalen Fehlermeldung die in <location>...</location> angegebene Ressource angezeigt. Falls es sich hierbei um eine aktive Ressource wie eine Java-ServerPage handelt, kann diese, soweit es sinnvoll ist, auf folgende Request-Attribute zugreifen:

```
javax.servlet.error.status_code
javax.servlet.error.exception_type
javax.servlet.error.message
```

### 3.13.3 Deployment-Deskriptor für sichere Anwendungen

In unserem nächsten Beispiel beschreiben wir einen Deployment-Deskriptor für eine »sichere« Anwendung:

```
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 1.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_1_2.dtd">
<web-app>
   <display-name>Eine sichere Applikation</display-name>
   <security-role>
      <role-name>Administrator</role-name>
   </security-role>
   <servlet>
      <servlet-name>SicheresServlet</servlet-name>
      <servlet-class>de.webapp.Examples.Servlets.SicheresServlet</servlet-class>
      <security-role-ref>
         <role-name>admin</role-name> <!-- der Rollenname, der im Code benutzt wird -
->
         <role-link>Administrator</role-link> <!-der Konfigurationsname einer
Sicherheitsrolle -->
      </security-role-ref>
   </servlet>
   <servlet-mapping>
      <servlet-name>SicheresServlet</servlet-name>
      <url-pattern>/admin</url-pattern>
   </servlet-mapping>
   <web-resource-collection>
      <web-resource-name>RestrictedArea</web-resource-name>
      <url-pattern>/admin</url-pattern>
      <url-pattern>/internal/*</url-pattern>
      <http-method>GET</http-method>
      <http-method>POST</http-method>
      <user-data-constraint>
         <transport-guarantee>CONFIDENTIAL</transport-guarantee>
      </user-data-constraint>
      <auth-constraint>
         <role-name>Administrator</role-name>
      </auth-constraint>
   </web-resource-collection>
</web-app>
```

*Listing 3.23: Deployment-Deskriptor einer sicheren Applikation*

Im Unterschied zu den bereits vorgestellten Deployment-Deskriptoren wird hier in `<security-role>...</security-role>` eine Sicherheitsrolle definiert. Auf diese wird von zwei Stellen aus verwiesen. Der erste Verweis befindet sich in der Auszeichnung `<security-role-ref>...</security-role-ref>` des Servlets. In ihr wird der Sicherheitsrolle ein Alias zugeordnet, das im Code benutzt wird. In unserem Fall heißt der Verweis auf die Rolle `<role-link>Administrator</role-link>` und der Alias `<role-name>admin</role-name>`.

Der zweite Verweis auf die Sicherheitsrolle befindet sich in `<web-resource-collection>...<auth-constraint><role-name>Administrator</role-name></auth-constraint></web-resource-collection>`. Er besagt, daß die durch die `<web-resource-collection>...</web-resource-collection>` bezeichneten Ressourcen dem Authentifizierungsschema der Rolle `Administrator` unterliegen. Um welche Ressourcen es sich dabei handelt, wird mit Hilfe von `<url-pattern>...</url-pattern>` und `<http-method>...</http-method>` angegeben. Würde hier keine Methode angegeben, gälten die Einschränkungen für alle Methoden. Weiterhin wird unter dem Punkt `<user-data-constraints>...</user-data-constraints>` mit `<transport-guarantee>CONFIDENTIAL</transport-guarantee>` eine gewisse Qualität der Kommunikation eingefordert. Außer `CONFIDENTIAL` (vertraulich) sind `NONE` und `INTEGRAL` (unverfälschte Übertragung) mögliche Werte.

Obwohl sie schon recht umfangreich sind, haben unsere drei Beispiele nicht alle Möglichkeiten des Deployment-Deskriptors ausgeschöpft. Über das Beschriebene hinaus gibt es noch einige Dinge festzulegen – unter anderem auch den Zugriff auf Ressourcen der Java 2 Enterprise Edition (J2EE). Dies geht jedoch über den Fokus dieses Buches hinaus – obgleich wir das J2EE-Konzept für gelungen und sehr zukunftsweisend halten.

# Teil II
# WebApp-Framework

Zur schnellen und sicheren Erstellung webbasierter Anwendungen ist die Nutzung eines darauf abgestimmten Frameworks unabdingbar. In diesem Teil des Buches stellen wir die nötigen Bestandteile eines Frameworks für einen webbasierten Applikationsserver vor und erläutern ihren Aufbau. Das resultierende WebApp-Framework ist als Paket auf der Website `http://www.webapp.de` zum Buch verfügbar.

Für die Entwicklung webbasierter Anwendungen sind drei Themen wichtig:

1. die Programmierschnittstelle des Webservers
2. die Anwendungsschicht
3. die Generierung von HTML

Als Schnittstelle zum Webserver benutzen wir das in Kapitel 3 ausführlich vorgestellte Servlet-API. Als Anwendungsschicht bezeichnen wir den Teil einer Web-Anwendung, der die Anwendungslogik enthält. Dazu zählen wir ausdrücklich nicht die Benutzeroberfläche, da diese zwar zur Ablaufsteuerung notwendig ist, jedoch die Anwendung nicht trägt und austauschbar sein sollte. Daher ist das Generieren von HTML ein eigener, dritter Bereich.

Alle drei Bereiche werden in diesem Teil des Buches abgedeckt. Als Einstieg beginnen wir mit den notwendigen Applikations-Basisdiensten zur Konfiguration und zum Protokollieren. Sie werden in den Kapiteln 4 und 5 beschrieben. Weiter geht es mit den Grundlagen zum Bau von leistungsfähigen Servern, die wir in Kapitel 6 erarbeiten. Dazu wird ein wiederverwendbares Fundament zum Thread-Management entwickelt. In Kapitel 7 benutzen wir das Server-Paket als Basis für das Design der Servlet-Engine jo!. Dabei erhalten Sie einen tiefen Einblick in jo!s Architektur und können Ihre Kenntnisse über die Servlet-Technologie vertiefen. Das trifft auch auf Kapitel 8 zu. Hier entwickeln wir SMI (Servlet Method Invocation) – eine Erweiterung des Servlet-API. SMI bietet die Möglichkeit, quasi hinter einem Servlet eine hochkonfigurierbare, modularisierte Anwendungsarchitektur aufzubauen. Für die Nutzung von Unternehmensdaten, widmen wir uns in Kapitel 9 ausgiebig der objektorientierten Kapselung relatio-

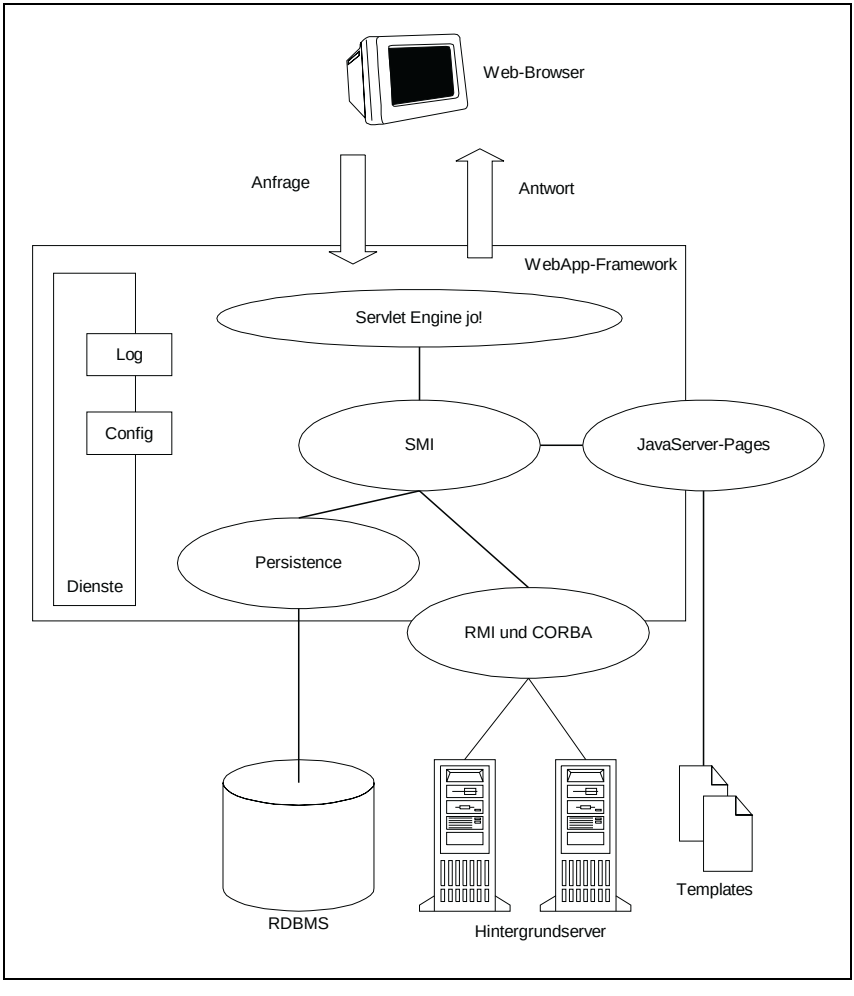


*Abbildung II.1: Architektur des WebApp-Frameworks*

naler Datenbanken. Der beschriebene Persistenz-Dienst ist eine der Säulen des Frameworks, da er einen sehr leicht zu programmierenden Datenbankzugang bietet. Im letzten Kapitel dieses Buchteils beschreiben wir schließlich Möglichkeiten zur dynamischen HTML-Generierung. Vorgestellt werden die sich schnell wandelnden verfügbare Techniken, insbesondere die JavaServer-Pages.

Nachdem Sie nun einen Überblick über den Inhalt dieses zweiten Buchteils erhalten haben, möchten wir Ihnen noch ein paar Sätze zum WebApp-Framework auf den Weg geben.

Was erwartet man von einem Framework zum Bau von webbasierten Applikationen? Natürlich soll es sehr flexibel sein, an verschiedene Anwendungssituationen anzupas-

sen sein und trotzdem von vornherein möglichst viele Einsatzfelder abdecken. Mit wenigen Schritten soll durch das Framework eine verläßliche, erweiterungsfähige Anwendung oder zumindest ein Teil davon erstellt werden können. Um dies gewährleisten zu können, muß das Framework anpassungsfähig sein. Eine wichtige Eigenschaft ist also seine Konfigurierbarkeit. Um diese sicherzustellen, enthält das WebApp-Framework den Konfigurationsmanager `ConfigManager`. Mit ihm können Konfigurationsdaten gelesen werden, die das Verhalten der Anwendung entscheidend prägen.

Trotzdem sind natürlich Sinn und Zweck einzelner Framework-Teile vorgegeben. Man kann über Konfigurationen aus einem Datenbank-Layer kein Malprogramm machen. Falls doch, ist etwas Entscheidendes beim Design schiefgelaufen. Frameworks oder Teile eines Frameworks sollen immer einem bestimmten Zweck dienen. Zumeist geschieht dies noch auf eine bestimmte Art und Weise. Teile des Frameworks benötigen andere Teile. Genau wie das Framework selbst über eine definierte Schnittstelle verfügen muß, sollten alle Teile des Frameworks ebenfalls über definierte Schnittstellen kommunizieren. In Java ist dies sehr elegant durch den intensiven Einsatz von Schnittstellen (Interfaces) möglich. Sie definieren, was ein Objekt können muß – wie es das macht, bleibt weiterhin sein Geheimnis. Schnittstellen ermöglichen es, Teile des Frameworks ohne Umstände auszutauschen, zu verbessern und zu erweitern.

Außer von Schnittstellen wird im WebApp-Framework noch von einer anderen Eigenschaft Javas intensiv Gebrauch gemacht: Das generische Einbinden erlaubt es, Anwendungsteile erst zur Laufzeit in das Programm zu integrieren.

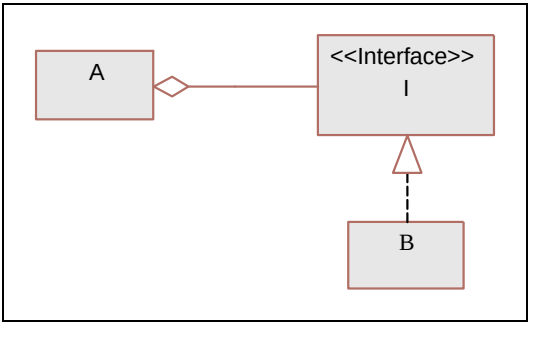


*Abbildung II.2: Entkoppelung des Designs durch Schnittstellen*

Ein Beispiel:

> Objekt `A` benötigt die Fähigkeiten eines anderen Objekts, um eine Aufgabe zu erfüllen. `A` weiß, daß ein Objekt, welches die Schnittstelle `I` implementiert hat, die anstehende Aufgabe erfüllen kann. Alles was es braucht, ist also eine Instanz einer Klasse, welche die Schnittstelle `I` implementiert. Ist ein entsprechender Klassenname bekannt, kann `A` über die Methode `java.lang. Class.forName(String)` ein Klas-

sen-Objekt erlangen. Das `Class`-Objekt wiederum verfügt über die Methode `newInstance()`, mit der sich zur Laufzeit Instanzen einer Klasse `B` anlegen lassen.

Der Trick besteht darin, in Frameworks möglichst mit Objekten vom Typ einer Schnittstelle zu arbeiten, anstatt explizit den Typ der Klasse zu verwenden, welche die Schnittstelle implementiert. Auf diese Weise kann man nachträglich über Konfigurationen festlegen, welche Klasse tatsächlich benutzt werden soll. Um nicht ganz auf formale Kontrollen zu verzichten, findet eine Typprüfung mit Hilfe von Typumwandlungen (Castings) statt – jede erzeugte Klasse muß letztendlich in einen Schnittstellen-Typ wandelbar sein.

Große Teile des WebApp-Frameworks realisieren dieses Design-Prinzip. In Schnittstellen wird das Verhalten des Frameworks festgelegt, die eigentliche Realisierung in Klassen ist lediglich exemplarisch und könnte auch anders erfolgen. Nur so wird größtmögliche Flexibilität garantiert. Natürlich muß dem gegenüber auch immer ein intelligentes Standardverhalten stehen. Es ist schön, wenn man alles ändern kann, noch schöner ist es aber, wenn schon alles so ist, wie man es haben will. Daher verfügen die meisten Komponenten über Standardeinstellungen, die das Konfigurieren stark erleichtern.

# 4 Konfigurationsmanagement

Im WebApp-Framework lassen sich die meisten Komponenten mit Hilfe des ConfigManagers konfigurieren. Dabei handelt es sich um unterschiedliche Konfigurationen: Nicht nur einfache Applikationsdaten, wie zum Beispiel die IP-Adresse eines Servers oder Einstellungen eines Nutzers sollen durch den Konfigurationsmanager verwaltet werden, sondern auch Konfigurationsdaten, die zum Aufbau einer Applikation benötigt werden. Dies ist beispielsweise der Fall, wenn zu einer Schnittstelle eine passende Klasse benötigt wird. Die Konfigurierbarkeit ist für das WebApp-Framework ein zentraler Aspekt.

Der Konfigurationsmanager soll den Komponenten des Frameworks und Applikationen jederzeit alle wichtigen Konfigurationsfragen beantworten können. Daraus folgt, daß es nur genau einen Konfigurationsmanager pro System geben darf, der von allen Punkten des Frameworks aus zugänglich und verfügbar sein muß. Es bedeutet gleichzeitig, daß durch den Konfigurationsmanager keine sensiblen Informationen verwaltet werden sollten.

Die Konfigurationsdaten sollen natürlich nicht nur gelesen, sondern auch geschrieben werden können. Dabei sollen Format und Speicherort prinzipiell frei wählbar sein. Es sollte jedoch ein Standardformat existieren, in dem Informationen hierarchisch und modular gespeichert werden können. Es soll möglich sein, Beobachter zu informieren, sobald sich Konfigurationsdaten ändern. Darüber hinaus sollte der `ConfigManager` ein komfortables Zurückgreifen auf Standardwerte (Defaulting) unterstützen.

Zusammengefaßt muß der `ConfigManager` also folgenden Anforderungen genügen:

- einfache Zugänglichkeit
- keine Festlegung auf bestimmte Speicherorte oder -medien, sondern Zugriff per URL
- Unterstützung hierarchischer Datenstrukturen
- Formatunabhängigkeit
- Rückgriff auf Standardwerte (Defaults)

## 4.1 Einfache Zugänglichkeit

Um die Zugänglichkeit und Einzigartigkeit des `ConfigManagers` sicherzustellen, ist er als Singleton [Gamma et al. 96] implementiert. Das Singleton-Muster stellt sicher, daß es von einer Klasse nur genau eine Instanz gibt. Zudem wird ein gut zu erreichender Zugriffspunkt auf diese Instanz zur Verfügung gestellt. Erreicht wird dies, indem der Konstruktor der Klasse `private` deklariert wird. Dadurch kann nur die Klasse selbst den Konstruktor aufrufen. Dieser Aufruf muß über eine als `static` deklarierte Methode erfolgen. Hinterlegt man nun beim ersten Aufruf der statischen Methode die erzeugte Instanz in einer Klassenvariablen, läßt sich beim nächsten Aufruf der Methode feststellen, ob sie schon einmal aufgerufen wurde. Ist dies der Fall, wird die von der Klassenvariablen referenzierte Instanz zurückgegeben, ansonsten wird eine neue Instanz erzeugt und in der Klassenvariablen hinterlegt (Listing 4.1). Um sicher zu gehen, daß sich zwei nahezu gleichzeitige Aufrufe der Methode nicht behindern, ist es wichtig, daß sich die Überprüfung der Klassenvariable und gegebenenfalls die Instantiierung in einem synchronisierten Block befinden.

```
public class Singleton {
   private static Singleton mySingleton = null;
   private Singleton () {}
   public static Singleton getInstance() {
         if (mySingleton == null) {
            synchronized (Singleton.class) {
              if (mySingleton == null) {
                 mySingleton = new Singleton ();
              }
            }
      return mySingleton;
   }
}
```

*Listing 4.1: Threadsicherer Singleton*

Der `ConfigManager` realisiert genau dieses Prinzip – mit dem kleinen Unterschied, daß Konstruktor wie Klassenvariable nicht `private` sondern `protected` sind, um eine spätere Erweiterung über Vererbung zu erleichtern.

Eine Instanz des Managers kann also von jedem Punkt eines Programms über die Klassenmethode `ConfigManager.getConfigManager()` erlangt werden.

## 4.2 Kein Festlegen auf einen Speicherort oder ein Speichermedium

Beim ersten Aufruf von `getConfigManager()` wird die Konfigurationsdatei des Konfigurationsmanagers gelesen. Diese Datei heißt per Konvention `Registry.cfg`. In ihr stehen die Basisdaten, die benötigt werden, um komfortabel auf andere Konfigurationsdaten

zugreifen zu können. Am wichtigsten ist dabei natürlich der Ort der Konfigurationsdateien. Nur Dateien, deren Ort bekannt ist, können schließlich gelesen werden. Das gilt nicht zuletzt für die Konfigurationsdatei des `ConfigManagers`.

Wie viele andere Programme läßt sich auch die Java-VM mit Zusatzparametern starten. Die Syntax lautet `java -Dkey=value ... myClass`. Über das `-D`-Attribut lassen sich Systemeigenschaften setzen, die durch die Methode `System.getProperty(String key)` zugänglich sind. Während seiner Instantiierung liest der `ConfigManager` die Eigenschaft mit dem Schlüsselnamen `CFGROOT` und erwartet dort den Namen des Verzeichnisses, in dem sich die Datei `Registry.cfg` befindet. Um zentrale Administration zu unterstützen, darf es sich dabei um einen URL (allerdings ohne Dateinamen) handeln. `-DCFGROOT=http://mein.ConfigHost.de/` ist also ebenso eine gültige Angabe wie `-DCFGROOT=/usr/local/meineKonfigurationen/` oder `-DCFGROOT=c:\config\`. Wird kein Protokoll angegeben, nimmt der `ConfigManager` an, daß es sich um eine lokale Datei handelt und stellt dem Dateinamen automatisch den Protokoll-Bezeichner `file://` voran.

Ist beim Start kein entsprechendes System-Attribut übergeben worden, wird über die Methode `ClassLoader.getSystemResource("Registry.cfg")` der Klassenpfad nach der gesuchten Datei durchforstet. Schlägt diese Suche fehl, wird im aktuellen Arbeitsverzeichnis nachgeschaut. Letztendlich wird eine `ConfigDataSourceException` ausgelöst, falls die Konfigurationsdatei nicht gefunden werden konnte.

## 4.3 Hierarchische Datenstrukturen

Nachdem geklärt ist, wie der `ConfigManager` zu seiner Konfiguration kommt, wollen wir uns den Inhalt dieser Konfiguration näher anschauen. Dazu führen wir uns noch einmal vor Augen, was der `ConfigManager` leisten soll.

Eine Applikation – wir nennen sie hier beispielhaft »OnlineShop« – soll möglichst einfach ihre Konfiguration in Erfahrung bringen können. Der `ConfigManager` muß also über eine Methode `getConfiguration("OnlineShop")` verfügen. Beim Aufruf dieser Methode sollte er in seiner Konfigurationsdatei überprüfen, ob ein Eintrag für »OnlineShop« vorliegt und, falls dies der Fall ist, die entsprechende Konfiguration an die Applikation zurückgeben. Wie könnte so ein Eintrag aussehen?

Natürlich muß er unter einem Namen abgelegt werden, in unserem Fall ist das »OnlineShop«. Der Eintrag selbst muß in erster Linie Angaben darüber enthalten, wo die gesuchte Konfiguration zu finden ist. Hilfreich wären also sowohl ein Pfad als auch ein Dateiname. Des weiteren kann es sinnvoll sein, den Pfad als absolut, relativ zum Arbeitsverzeichnis oder relativ zur Datei `Registry.cfg` zu interpretieren. Auch dafür wäre ein Attribut wünschenswert. Der typische Aufbau einer `Registry.cfg`-Datei sieht folgendermaßen aus:

```
{
   OnlineShop = {
      PATH = "OnlineShopPfad"
      FILE = "OnlineShop.cfg"
      RELATIVE2CFGROOT = true
   }
}
```

*Listing 4.2: Einfache* `Registry.cfg`

Der Wert von `PATH` gibt dabei den Basispfad aller Dateien an, die zu dieser Konfiguration gehören. Voreingestellt ist das aktuelle Arbeitsverzeichnis. `FILE` steht für eine Standarddatei, die gelesen wird, wenn nicht explizit eine andere Datei verlangt wird. Und `RELATIVE2CFGROOT` gibt an, ob der Pfad relativ zu `Registry.cfg` oder relativ zum aktuellen Arbeitsverzeichnis interpretiert werden soll. Ist `PATH` absolut, macht das Attribut `RELATIVE2CFGROOT` keinen Sinn. Sein Standardwert ist daher `false`.

Das Format von `Registry.cfg` kann von der Klasse `ConfigFileReader` des `ConfigManager`-Pakets gelesen werden. Wie das geschieht, erklären wir im nächsten Abschnitt.

Wichtig ist festzuhalten, daß allein schon die Konfiguration des `ConfigManagers` Hierarchien beinhaltet. Ein Master-Attribut (`OnlineShop`) kann viele Details (`PATH`, `FILE`, etc.) besitzen, die theoretisch wiederum über weitere Details verfügen können. Konfigurationsdaten müssen sich beliebig schachteln lassen, damit ein gezielter Zugriff auf bestimmte Werte über einen Navigationspfad möglich ist. Dies ermöglicht es zudem, eine Aufzählung aller Attribute von »OnlineShop« zu erlangen. Ohne eine explizite Hierarchie wäre das komplizierter.

Um auf eine bestimmte Konfiguration zugreifen zu können, verfügt der `ConfigManager` über die Methode `getConfiguration()`. Dementsprechend gibt `getConfiguration("OnlineShop")` dem Aufrufer die Konfiguration »OnlineShop« zurück: und zwar die der Standardkonfigurationsdatei, die in `FILE` spezifiziert wurde. Wird eine andere Datei verlangt, so muß deren Name als zweites Argument übergeben werden. Ein Aufruf lautet dann `getConfiguration("OnlineShop", "andereKonfiguration.cfg")`.

Beide Methoden geben je eine Instanz der Klasse `Configuration` zurück (Abbildung 4.1). `Configuration` hat einige Ähnlichkeiten mit der Klasse `java.util.Properties`. So besitzt auch die Klasse `Configuration` Methoden zum Lesen und Schreiben persistenter Daten. Sie verfügt über Methoden, um einzelne Werte zu setzen und zu lesen. Darüber hinaus läßt sich jeder Instanz von `Configuration` eine zweite Instanz zuordnen, die Defaultwerte enthält.

Soweit die Gemeinsamkeiten. Die wesentlichen Unterschiede liegen in der Art des Zugriffs auf einzelne Elemente und des Speicherns der gesamten Daten. Im Gegensatz zu `Properties` unterstützt `Configuration` explizit hierarchische Strukturen. Durch die Methode `getElement(String aPfad)` läßt sich über einen Pfad auf ein bestimmtes Ele-

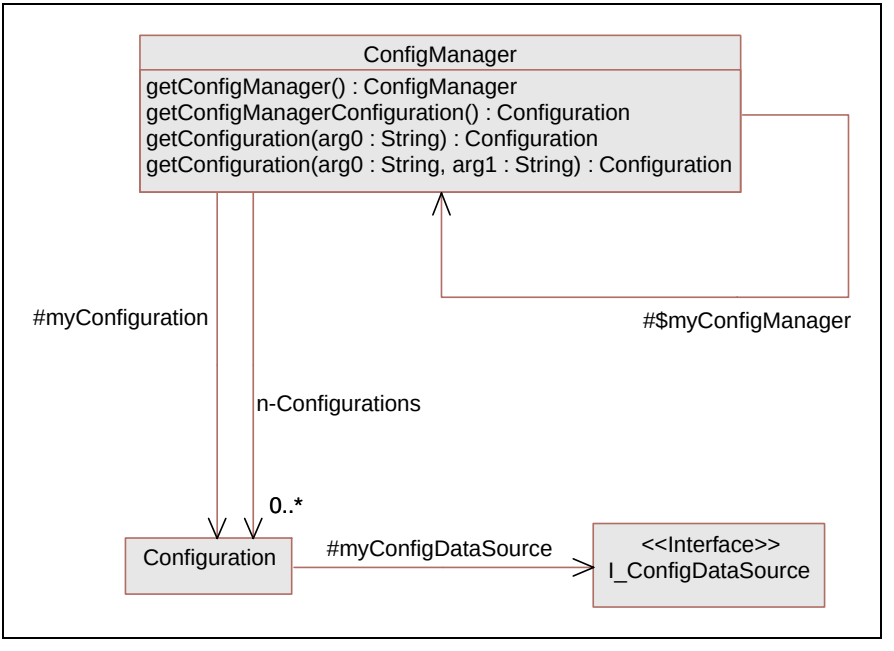


*Abbildung 4.1: Beziehungen zwischen `ConfigManager`, `Configuration` und `I_ConfigDataSource`*

ment zugreifen. Dabei kann es sich bei diesem Element um einen `String`, `Boolean`, `Integer`, `Double`, `Vector` oder um eine `Hashtable` handeln. Der Zugriffspfad wird aus den Schlüsseln der geschachtelten Hashtabellen und den Indizes von enthaltenen Vektoren gebildet. Die Pfadteile müssen durch Kommata oder Semikolons getrennt werden. Auf das Attribut `FILE` von `OnlineShop` läßt sich also folgendermaßen zugreifen: `getElement("OnlineShop;FILE")`.

## 4.4 ConfigFileReader-Dateiformat

Das `ConfigFileReader`-Dateiformat stellt eine exemplarische Realisierung der oben genannten Anforderungen dar. Es ermöglicht Datenstrukturen wie `Hashtable` und `Vector` abzubilden und zu schachteln.

Dabei müssen folgende Konventionen beachtet werden:

- Jede Konfigurationsdatei hat als äußerste Struktur entweder eine Hashtabelle oder einen Vektor.
- Hashtabellen beginnen und enden mit einer geschweiften Klammer.
- Hashtabellen-Schlüssel müssen den Konventionen für Java-Variablennamen folgen.

- Hashtabellen-Werte sind beliebige, durch Anführungszeichen begrenzte Zeichenketten, ganze oder Fließkommazahlen, die beiden Wahrheitswerte `true` und `false` sowie `Hashtables` oder `Vectoren`.
- Schlüssel und Werte der Tabellen sind durch ein Gleichheitszeichen voneinander getrennt.
- Zwischen Werten und Schlüsseln darf ein Trennzeichen (Komma, Semikolon) eingefügt werden.
- Vektoren beginnen und enden mit einer runden Klammer und enthalten keine oder beliebig viele Vektorelemente.
- Vektorelemente müssen durch ein Leerzeichen, Tabstopp, Zeilenvorschubzeichen, Wagenrücklaufzeichen, Komma oder Semikolon voneinander getrennt werden.
- Vektorelemente sind beliebige, durch Anführungszeichen begrenzte Zeichenketten, ganze oder Fließkommazahlen, die beiden Wahrheitswerte `true` und `false` sowie Hashtabellen oder Vektoren und entsprechen somit Hashtabellen-Werten.

Die formale Definition des Dateiformats können Sie in Anhang A nachschlagen. Listing 4.3 zeigt eine Beispieldatei im `ConfigFileReader`-Format.

```
{
   schluessel1 = "eineZeichenkette" // Kommentar
   /*
    * mehrzeiliger
    * Kommentar
    */
   schluessel2 =
         {
            zweiteEbeneSchluessel1 = "eineZeichenkette"
            zweiteEbeneSchluessel2 = 12.5   // Dezimalbruch
            zweiteEbeneSchluessel3 = 42     // Integer
            zweiteEbeneSchluessel4 = true   // Boolean
            zweiteEbeneSchluessel5 = false  // Boolean
            zweiteEbeneSchluessel6 =
               (
                  "Element1", "Element2", 5
                  // das dritte Element ist ein Integer,
                  // die Kommata sind optional
               )
      }
   schluessel3 = #include("eineAndereDatei.cfg")
}
```

*Listing 4.3: Beispielkonfigurationsdatei für den `ConfigFileReader`*

Um auf den Wert `Element1` des Vektors über ein `Configuration`-Objekt zuzugreifen, wäre folgender Aufruf nötig:

```
aConfiguration.getElement("schluessel2;zweiteEbeneSchluessel6;1");
```

Folgenden Aufruf würde `Element2` zurückliefern:

```
aConfiguration.getElement("schluessel2;zweiteEbeneSchluessel6;2");
```

Zudem unterstützt `ConfigFileReader` den modularen Aufbau von Konfigurationen durch den `#include`-Befehl. Er kann anstelle eines Vektorelementes oder Hashtabellen-Wertes in eine Datei eingefügt werden. Hinter dem Befehl muß in Klammern und Anführungszeichen ein URL stehen. Relative URLs werden immer relativ zu der Datei interpretiert, in der sie sich befinden.

Da `Configuration` nicht explizit modulare Dateistrukturen unterstützt, gehen diese beim Speichern verloren. Aus vielen Dateien kann so unter Umständen eine einzige werden.

## 4.5 Formatunabhängigkeit

Mit dem `ConfigFileReader`-Format steht uns ein sehr flexibles Dateiformat zur Verfügung. Nun ist aber nicht anzunehmen, daß immer alle Konfigurationsdaten in genau diesem Format gespeichert werden sollen. Daher ist es sinnvoll, vom Dateiformat zu abstrahieren. Aus diesem Grund ist die Klasse `ConfigFileReader` nicht fest mit `Configuration` verbunden, sondern durch die Schnittstelle `I_ConfigDataSource` entkoppelt:

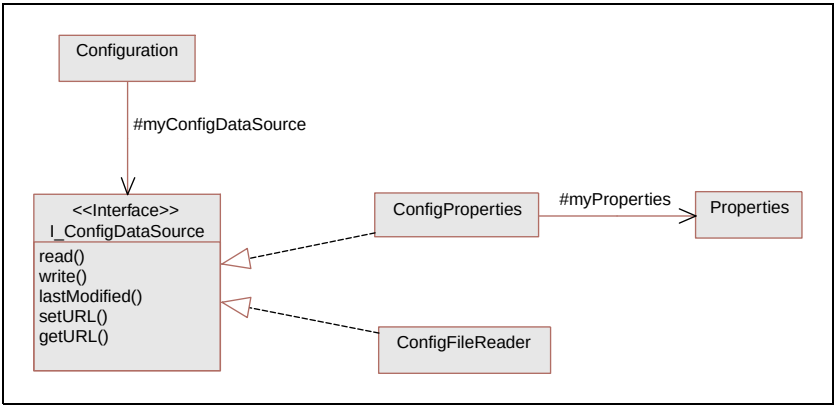


*Abbildung 4.2: `I_ConfigDataSource` und seine Implementierungen `ConfigFileReader` und `ConfigProperties`*

Die Schnittstelle (Listing 4.4) definiert, daß eine Klasse, die `I_ConfigDataSource` implementiert, eine `Configuration` lesen und schreiben können muß. Der Speicherort wird durch einen URL definiert. Außerdem muß das Datum der letzten Änderung abfragbar sein. Wie eine Klasse diese Forderungen erfüllt, bleibt ihr überlassen. Neben dem

Speichern in einer Datei wäre zum Beispiel auch die Anbindung an eine Datenbank denkbar.

```
package de.webapp.Framework.ConfigManager;

import java.net.*;

public interface I_ConfigDataSource {
/** Liest die Datenquelle und gibt das Ergebnis als Object zurück */
public Object read() throws ConfigDataSourceException;
/** Schreibt ein Configuration-Objekt */
public void write(Configuration configuration) throws ConfigDataSourceException;
/** Gibt an, wann die Datenquelle das letzte Mal geändert wurde */
public long lastModified();
/** Setzt den URL der Datenquelle */
public void setURL(String aURL) throws MalformedURLException;
/** Gibt den URL dieser Datenquelle zurück */
public URL getURL();
}
```

*Listing 4.4: Die Schnittstelle `I_ConfigDataSource`.*

Im `ConfigManager`-Paket befinden sich zwei Klassen, die die Schnittstelle implementieren (Abbildung 4.2). Die eine, `ConfigProperties`, ist lediglich ein dünner Mantel um die `java.util.Properties`-Klasse. Entsprechend unterstützt sie nur Zeichenketten und keine echten hierarchischen Strukturen. Die andere, `ConfigFileReader`, erfüllt alle oben genannten Anforderungen. Sie ist darüber hinaus durch den `#include`-Direktive in der Lage, Konfigurationen zu lesen, die aus mehreren Dateien bestehen.

Um eine bestimmte Realisierung von `I_ConfigDataSource` zu benutzen, gibt es mehrere Möglichkeiten. Zum einen läßt sich der Klassenname explizit als Attribut eines `Registry.cfg`-Eintrages festlegen. Dies geschieht über das Attribut `DATASOURCECLASSNAME`. Zum anderen kann die Klasse `Configuration` anhand der Dateiendung erkennen, welche `I_ConfigDataSource` benutzt werden sollte. Voreingestellt ist hier die Klasse `ConfigProperties` für Dateien mit der Endung `properties` und `ConfigFileReader` für alle anderen. Weitere Abbildungen können über entsprechende Methoden der Klasse `Configuration` verwaltet werden.

So registriert `addConfigDataSourceClass(String, Class)` für eine Dateiendung eine `I_ConfigDataSource`-Klasse. Mit `removeConfigDataSourceClass(String)` kann eine solche Abbildung wieder entfernt werden. Da beide Methoden `static` sind, gelten diese Einstellungen für alle Instanzen von `Configuration`.

## 4.6 Rückgriff auf Standardwerte

Oft unterscheiden sich verschiedene Konfigurationen für die gleiche Anwendung nur marginal. Daher ist es sinnvoll, auf eine Standardkonfiguration zurückgreifen zu können. Der Konfigurationsmanager bietet dafür einen einfachen Mechanismus. Zusätzlich zu den Attributen `FILE`, `PATH` etc. kann man mit `DEFAULTS` auf eine andere Konfiguration verweisen (Listing 4.5). Das zurückgegebene `Configuration`-Objekt enthält dann als Standardkonfiguration ein `Configuration`-Objekt einer anderen Konfiguration. Ist ein Element nicht im eigentlichen Konfigurationsobjekt enthalten, wird automatisch auf das Standardkonfigurationsobjekt zurückgegriffen. Dieses kann wiederum über ein Standardobjekt verfügen und so fort. Dabei müssen Ringschlüsse vermieden werden.

Eine entsprechende `Registry.cfg`-Datei könnte also so aussehen:

```
{
   StandardOnlineShop = {
      PATH = "StandardOnlineShopPfad"
      FILE = "StandardOnlineShop.cfg"
      RELATIVE2CFGROOT = true
   }
   CDShop = {
      PATH = "CDShopPfad"
      FILE = "CDShop.cfg"
      RELATIVE2CFGROOT = true
      DEFAULTS= "StandardOnlineShop"
   }
}
```

*Listing 4.5: CD-Shop-Konfiguration mit Rückgriff auf eine Standardkonfiguration*

Zusätzlich läßt sich, genau wie in der `Properties`-Klasse, bei jedem Aufruf von `getElement()` ein Standardwert angeben. Dieser Wert wird zurückgegeben, falls dem Schlüssel weder in der Konfiguration selbst, noch in einer eventuell vorhandenen Standardkonfiguration ein Wert zugeordnet ist. Die vollständige Signatur heißt `Object getElement(String pfad, Object default)`.

## 4.7 Konfigurations-Objekt

Neben dem normalen Zugriff auf Konfigurationsdaten bietet die Klasse `Configuration` noch andere Eigenschaften und Methoden, die das Arbeiten stark erleichtern. Wichtigste Methode ist wohl `setElement(String, Object)`, die es analog zu `getElement(String)` erlaubt, einen Wert der Konfiguration zu verändern. Eine veränderte Konfiguration läßt sich über die `write()`-Methode speichern.

Über die Methode `setAutoReload(boolean)` läßt sich eine Konfiguration in einen Modus schalten, der vor jedem Zugriff überprüft, ob sich die Konfiguration geändert hat. Ist dies der Fall, wird sie automatisch neu geladen. Außerdem ist es möglich, Konfigurations-Beobachter bei der Konfiguration zu registrieren. Dies geschieht über die Methode `addObserver(Observer)`, die `Configuration` von der Klasse `java.util.Observable` erbt. Bei jeder Änderung der Konfiguration werden alle registrierten `java.util.Observer` benachrichtigt.

Diese Aktualitäts-Überprüfung kann auch automatisch periodisch vorgenommen werden. Dazu muß lediglich die Methode `startPeriodicObservation()` aufgerufen werden. `setCheckInterval(long)` setzt die Zeit in Millisekunden, die zwischen jeder Überprüfung vergehen soll; `getCheckInterval()` gibt die gesetzte Zeit zurück; `stopPeriodicObservation()` stoppt die automatische Beobachtung.

# 5 Protokoll-Dienst

Neben einer zentralen Administration durch einen Konfigurations-Dienst ist es in einem Framework unerläßlich, auch Programm-Meldungen zurückverfolgen zu können. Daher benötigt man einen Protokoll-Dienst. Genau wie der ConfigManager muß dieser Dienst problemlos zugänglich sein und eine einfache Schnittstelle besitzen. Schließlich möchte man sich beim Programmieren nicht mit aufwendigen Befehlen herumschlagen, nur um eine Zeile in eine Datei zu schreiben.

Das WebApp-Framework verfügt über einen einfachen Protokoll-Dienst, der sich im Paket `de.webapp.Framework.Log` befindet. Im Gegensatz zum `ConfigManager` ist die Klasse `Log` nicht als Singleton realisiert, da es sehr unwahrscheinlich ist, daß jemals von `Log` geerbt werden soll. Um gut zugänglich zu sein, verfügt `Log` ausschließlich über statische Methoden, die sich in drei Kategorien einteilen lassen.

1. Methoden zum Protokollieren: `Log.log(...)`
2. Methoden, um festzustellen, ob protokolliert werden soll: `Log.isLog(...)`
3. Methoden zur Manipulation der Genauigkeit des Protokolls (Loglevel): `setLogLevel(...)` und `setLogLevelDefault(...)` beziehungsweise `getLogLevel(...)`, `getLogLevelDefault(...)`

## 5.1 Grundlagen

Oft soll nicht jede Ausgabe protokolliert werden – bei einer Fehlersuche können es manchmal jedoch nicht genug Ausgaben sein. Daher ist es notwendig, über einen Mechanismus zu verfügen, mit dem sich die Genauigkeit der Protokoll-Meldungen steuern läßt. Dieser wird in `Log` über das Loglevel gesteuert, das sich beliebig verändern läßt. In der Schnittstelle `C_Log` sind für die verschiedenen Loglevel Integer-Konstanten vordefiniert (Tabelle 5.1). Dabei handelt es sich jedoch nur um Vorschläge. Grundsätzlich sind alle Levels zwischen `0` und der systemabhängigen Konstante `Integer.MAX_VALUE` erlaubt.

| Konstante | Wert | Bedeutung |
|---|---|---|
| NOLOG | 0 | nichts protokollieren |
| ERROR | 1 | Fehler protokollieren |
| MODULE | 2 | Modulmeldungen protokollieren |
| METHOD | 3 | Methodenmeldungen protokollieren |
| FORTYTWO | 42 | So ziemlich alles protokollieren |

*Tabelle 5.1: Loglevel-Konstanten aus `C_Log`*

Mit der Methode `Log.log(Object, int)` wird eine Meldung eines spezifischen Loglevels protokolliert. Der erste Parameter gibt dabei die Meldung, der zweite das Loglevel an. Wird nur eine Meldung angegeben und auf das explizite Setzen des Logevels verzichtet, wird die Meldung mit dem Loglevel protokolliert, das durch die Methode `setLogLevelDefault(int)` vorgegeben wurde. Der entsprechende Methodenaufruf lautet `Log.log(Object)`.

Um zu kontrollieren, ab welchem Loglevel eine Meldung protokolliert wird, muß die Methode `setLogLevel(int)` verwendet werden.

Beispiel:

```
setLogLevelDefault(C_Log.MODULE);
setLogLevel(C_Log.METHOD);
Log.log("Meldung ohne LogLevel");
Log.log("Meldung mit LogLevel", C_Log.FORTYTWO);
```

Das Standard-Loglevel (`LogLevelDefault`) für Meldungen sei `C_Log.MODULE`, das aktuelle Loglevel `C_Log.METHOD`. Versucht nun ein Programm eine Protokoll-Meldung ohne Loglevel via `Log.log("Meldung ohne Loglevel")` abzusetzen, wird diese protokolliert, da das aktuelle Loglevel höher ist als das Standard-Loglevel.

Wird jedoch eine Meldung per `Log.log("Meldung mit Loglevel", C_Log.FORTYTWO)` abgesetzt, wird diese nicht protokolliert, da `C_Log.FORTYTWO` ein niedrigeres Loglevel anzeigt als das aktuelle Loglevel `C_Log.METHOD`. Die Meldung wird also unterdrückt.

## 5.2 Loggen oder nicht loggen?

Leider sind in Java Zeichenketten-Operationen mit der `String`-Klasse sehr teuer. Setzt man verschiedene Zeichenketten per Konkatenations-Operator (`"String1" + "String2"`) zusammen, werden die zwei Objekte der Speicherbereinigung übergeben und ein drittes erzeugt. Kurzfristig ist dies aufwendig, langfristig führt es zu viel Arbeit für die automatische Speicherbereinigung (Garbage Collection). Solche Operationen sollten also möglichst vermieden werden. Aus diesem Grund verfügt `Log` über die Methode

`Log.isLog()`. Sie erlaubt es, vor der Konstruktion einer Meldung festzustellen, ob überhaupt protokolliert werden soll.

Während `Log.isLog()` lediglich anzeigt, ob das aktuelle Loglevel größer als `C_Log.NOLOG` ist, gibt `Log.isLog(int)` an, ob die Meldung eines bestimmten Levels protokolliert würde. Für die Praxis bedeutet dies, daß `Log`-Aufrufe folgendermaßen kodiert werden sollten:

```
if (Log.isLog(C_Log.METHOD)) {
   String meldung = "Irgendwas mit Namen " + name + " ist passiert. " +
   grund + " könnte vielleicht der Grund gewesen sein. Vielleicht aber
   auch folgendes: " + andererGrund + ", " + nochEinGrund;
   Log.log(meldung, C_Log.METHOD);
}
```

`name`, `grund`, `andererGrund` und `nochEinGrund` seien Zeichenketten, welche die Konstruktion von `meldung` sehr aufwendig gestalten. Durch das Absichern des Code-Blocks mittels `isLog()` wird diese speicher- und rechenintensive Konstruktion nur durchgeführt, wenn sie auch tatsächlich benötigt wird. Weiter verbessern läßt sich das Laufzeitverhalten durch den Einsatz von `StringBuffern` anstelle von `Strings`.

## 5.3 Konfiguration des Protokoll-Dienstes

Würden alle Protokoll-Meldungen in der Standardausgabe landen, wäre das ein ziemliches Durcheinander. Daher kann man verschiedene Protokolldateien einrichten.

Während der Initialisierung der Klasse `Log` wird zu diesem Zweck eine Konfigurationsdatei ausgewertet. Diese wird vom `ConfigManager` bezogen. Um dies zu ermöglichen, muß sich in der Datei `Registry.cfg` ein Eintrag mit Namen `Log` befinden (Listing 5.1). Befindet sich dort kein entsprechender Eintrag, erfolgen alle Ausgaben in den Standardfehlerstrom.

```
{
   Log = {
      FILE = "log.cfg";
      RELATIVE2CFGROOT = true ;
   };
   ...
}
```

*Listing 5.1: Beispielhafter Ausschnitt aus `Registry.cfg`*

In der Datei `log.cfg` werden die verschiedenen Protokolle definiert. Um möglichst wenig Parameter angeben zu müssen, ist die Datei in zwei Teile geteilt. Im ersten werden Grund- und Standardwerte gesetzt, im zweiten können sie für die verschiedenen Protokolle überschrieben werden.

```
{
   // TEIL I
   LOGROOT = "../logs"
   RELATIVE2CFGROOT = true
   LEVEL = 4
   DEFAULTLEVEL = 1
   FILE = "default.log"

   // TEIL II
   LOGS = {
      jo = {
         FILE = "jo_event.log"
      }
      jo_access = {
         FILE = "jo_access.log"
         RELATIVE2LOGROOT = true
      }
      AdminService = {
         LEVEL = 1
         DEFAULTLEVEL = 1
      }
   }
}
```

*Listing 5.2: Beispielhafte Konfigurationsdatei für den Protokoll-Dienst*

Im ersten Teil geben die Parameter das Loglevel und den Ort der Ausgabe an. Wird keine Datei angegeben, erfolgt die Ausgabe automatisch in den Standardfehlerstrom. Wird eine Datei über das `FILE`-Attribut spezifiziert, so kann man mit `LOGROOT` einen Pfad angeben, der für alle Protokoll-Dateien als Basisverzeichnis benutzt wird. Den Ort von `LOGROOT` wiederum kann man über `RELATIVE2CFGROOT` relativ zur Datei `Registry.cfg` legen.

Die Einträge im zweiten Teil sind sehr ähnlich. Grundsätzlich werden alle Einstellungen aus dem ersten Teil als Standardwerte benutzt. Daher müssen nicht für jedes Protokoll alle Parameter angegeben werden. Mögliche Parameter sind `RELATIVE2LOGROOT`, `FILE`, `LEVEL` und `DEFAULTLEVEL`. Wird `RELATIVE2LOGROOT` nicht angegeben und dennoch eine Datei spezifiziert, wird diese standardmäßig relativ zum Log-Basisverzeichnis interpretiert.

Um eine Meldung in ein bestimmtes Protokoll zu schreiben, muß dieses angegeben werden. Andernfalls wird immer das Standardprotokoll benutzt.

Beispiel:

```
if (Log.isLog(C_ERROR, "jo")) Log.log("Meldung", C_Log.ERROR, "jo");
```

Diese Programmzeile schreibt eine Meldung in das Protokoll `jo`.

Entsprechend gibt es von allen Methoden Versionen, die als letztes Argument den Namen eines Protokolls annehmen.

In der Realisierung des Protokoll-Dienstes wird diese Funktionalität durch Delegieren an einen `LogHandler` erreicht (Abbildung 5.1). Entsprechend steht für jedes Protokoll ein `LogHandler` zur Verfügung. Um Fehler der `Log`-Klasse dokumentieren zu können, verfügt diese zudem über einen Standard-`LogHandler`.

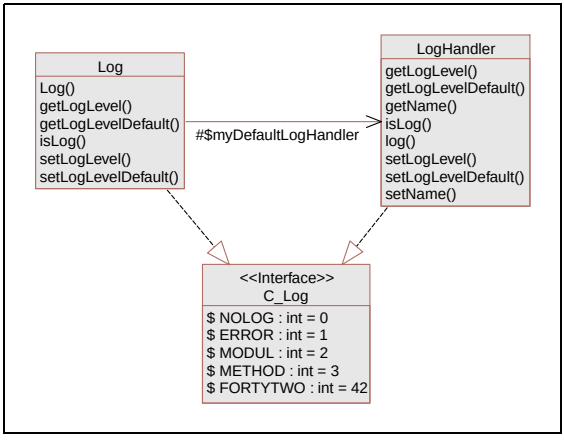


*Abbildung 5.1: Delegation an `LogHandler`*

## 5.4 Protokollieren mit Format

Um nicht nur Zeichenketten, sondern eine Ausnahme oder irgendein anderes Objekt zu protokollieren, akzeptieren alle Logmethoden `Object` als Parameter. Eine Sonderbehandlung widerfährt `Throwable`-Objekten: Sie werden automatisch mit ihrem `StackTrace` protokolliert. Darüber hinaus wird bei `ServletExceptions`, `SQLExceptions`, `InvocationTargetExceptions` und `I_WebAppExceptions` mit möglichen geschachtelten `Throwables` ebenso verfahren. Möchte man tatsächlich nur die Meldung des `Throwables` protokollieren, sollte man daher die `toString()`-Methode aufrufen:

```
Throwable myThrowable = new Throwable("Fehler");
// mit StackTrace
Log.log(myThrowable);
// oder ohne
Log.log(myThrowable.toString());
```

Standardmäßig wird allen Protokollausgaben Datum, Uhrzeit und Protokollname vorangestellt. Erst danach wird die tatsächliche Meldung geschrieben:

In manchen Fällen ist dieses Format jedoch nicht erwünscht – beispielsweise bei Logdateien eines Webservers (Abbildung 5.3). Gewöhnlich wird hier das Common Log Format (CLF, etwa: allgemeines Protokollformat) verwendet, welches genau spezifiziert, was in welcher Reihenfolge in der Protokoll-Datei zu stehen hat.

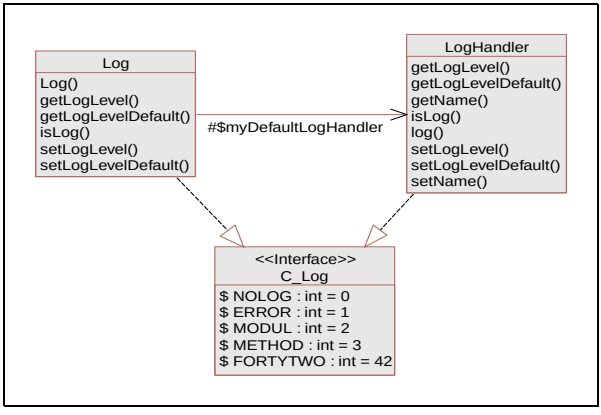


*Abbildung 5.2: Protokollausgabe beim Start des AdminServices*

```
Eingabeaufforderung - jostart
D:\repository\webapp\bin>jostart
ConfigManager: Reading '../etc/ConfigManager.cfg' ...
auster.ping.de - - [21/Nov/1998:17:26:24 +0100] "GET /servlet/HelloWorld HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:26 +0100] "GET /servlet/HelloWorld HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:29 +0100] "GET /servlet/request HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:33 +0100] "GET /servlet/erbsenzaehler HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:34 +0100] "GET /servlet/erbsenzaehler HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:37 +0100] "GET /servlet/session HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:41 +0100] "GET /servlet/session?JSID%3D91166559767870 HTTP/1.
" 200 -
auster.ping.de - - [21/Nov/1998:17:26:46 +0100] "GET /servlet/session HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:47 +0100] "GET /servlet/session HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:52 +0100] "GET /servlet/session2 HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:54 +0100] "GET /servlet/session2 HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:55 +0100] "GET /servlet/session2 HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:56 +0100] "GET /servlet/session2?verfallen HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:26:58 +0100] "GET /servlet/session2 HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:27:03 +0100] "GET /servlet/guestbook HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:27:05 +0100] "POST /servlet/guestbook HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:27:07 +0100] "POST /servlet/guestbook HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:27:12 +0100] "GET /servlet/smtp HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:27:16 +0100] "POST /servlet/smtp HTTP/1.0" 200 -
auster.ping.de - - [21/Nov/1998:17:27:23 +0100] "GET /servlet/smtp HTTP/1.0" 200 -
```

*Abbildung 5.3: Ausgabe im Common Log Format*

Um dies zu erreichen, gibt es die Möglichkeit, der `log()`-Methode beim Aufruf eine Muster-Zeichenkette zu übergeben. Die entsprechenden Signaturen sehen folgendermaßen aus:

```
void log(Object[] args, String aPattern, int aLevel, String aLog)
void log(Object[] args, String aPattern, String aLog)
void log(Object[] args, String aPattern, int aLevel)
```

Das verwendete Muster muß den Konventionen der Klasse `java.text.MessageFormat` (siehe JDK-API-Dokumentation) entsprechen. Das Standardmuster lautet `"[{0, date} {0,time}] - {1}: {2}"`.

# 6 Server-Toolkit

Bereits in Kapitel 2 haben wir uns mit der Entwicklung eines simplen HTTP-Servers beschäftigt. Während es dort eher um eine grundsätzliche Einführung anhand des Beispiels HTTP ging, wollen wir uns nun etwas stärker mit dem zugrundeliegenden Konzept beschäftigen.

Gewöhnlich teilt man eine Server-Architektur in zwei Teile: den Service und einen oder mehrere Handler (Abbildung 6.1). Der Service nimmt in der Regel eine Verbindung oder ein Datenpaket an, um diese(s) an einen Handler weiterzuleiten. Dieser führt den angeforderten Dienst aus. Umgebungsvariablen und sonstige allgemeine Attribute oder Dienste werden dem Handler normalerweise vom Service zur Verfügung gestellt. Das Paket `de.webapp.Framework.Server` enthält mehrere Schnittstellen und Klassen, die dieses Verhalten definieren beziehungsweise realisieren.

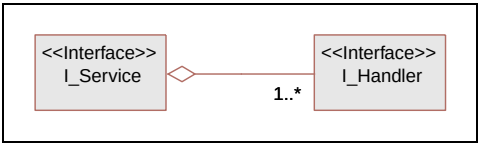


*Abbildung 6.1: Service-Handler-Beziehung*

Das Paket stellt somit eine Plattform zur Verfügung, mit der man innerhalb kürzester Zeit einen Server programmieren kann. Dies wird dadurch erreicht, daß die Klassen aus dem Paket nur noch durch die anwendungsprotokollspezifischen Teile ergänzt werden müssen.

## 6.1 Grundlegende Schnittstellen

Zunächst einmal ist da die Schnittstelle `I_Service`. Sie definiert alle wichtigen Eigenschaften eines Services: angefangen bei Name, Version, Port und IP-Adresse bis hin zu möglichen Attributen sowie dem Klassennamen seiner Handler. Zudem enthält sie Management-Methoden, um den Service zu starten, anzuhalten und einen Neustart auszuführen. Weiterhin wird das Handler-Management definiert.

Die zu `I_Service` korrespondierende Schnittstelle `I_Handler` enthält Methoden zum Zugriff auf den zugehörigen Service und seine Attribute. Zudem verfügt sie über eine `destroy()`-Methode, die es dem Service ermöglicht, den Handler und eventuell von ihm gebundene Ressourcen aktiv freizugeben.

### 6.1.1 Serviceleistungen

Die Methoden des Services (Tabelle 6.1) lassen sich verschiedenen Kategorien zuordnen. Zunächst einmal sind da simple Attribut-Methoden, die lediglich feste Eigenschaften des Services bekanntgeben. Hierzu zählen `getName()`, `getMajorVersion()`, `getMinorVersion()`, `getBindAddress()` und `getPort()`. Eine Sonderrolle nehmen die Methoden `getAttribute()`, `setAttribute()` und `getAttributeNames()` ein, da sie Zugriff auf eine Datenstruktur gewähren, die sich unter Umständen zur Laufzeit stark ändern kann.

Das Laufzeitverhalten wird durch eine andere Art von Methoden definiert. Dazu zählen `start()`, `stop()`, `restart()` und `isAlive()`.

Als letztes bleiben die Methoden für das Handler-Management. Unter Handler-Management verstehen wir den Mechanismus, der dafür sorgt, daß für eine Verbindung (im Fall von TCP als Netzprotokoll) oder ein Datagramm (falls UDP benutzt wird) ein Handler zur Verfügung steht. Natürlich könnte man einfach für jedes Datagramm und jede Verbindung einen neuen Handler instantiieren, wie wir es auch in Kapitel 2 getan haben. Um die Leistungsfähigkeit unseres Servers zu erhöhen, wollen wir dies jedoch nicht tun.

In Java ist der Start eines Threads nämlich recht aufwendig. Zwischen dem Aufruf der `start()`- und der `run()`-Methode eines Threads muß eine Menge Verwaltungsarbeit erledigt werden. Dieser Initialisierungsaufwand läßt sich leider nicht reduzieren. Jedoch können wir einen einmal initialisierten Thread wiederverwenden. Wie das genau funktioniert, werden wir in Abschnitt 6.4.2 erklären. Da Handler in der Regel als Threads realisiert sind, bedeutet dies zunächst einmal, daß der Service neben einer Methode `getHandler()` über eine Methode `recycleHandler()` verfügen sollte.

Um die Leistung unseres Services zu steigern, wollen wir also einen Handler-Pool unterhalten. Ein Pool sei hier durch die Eigenschaft gekennzeichnet, daß man ihm gleichartige Elemente hinzufügen kann und diese auch ohne Angabe eines Schlüssels wieder entnehmen kann. Außerdem besitzt jeder Pool eine begrenzte Kapazität (siehe `de.webapp.Framework.Utilities.I_Pool`). Um ein wenig mehr Kontrolle über diesen Pool zu erhalten, verfügt der Service über die Methoden `setMaxHandlerThreads()` und `setMinHandlerThreads()` sowie die korrespondierenden `get`-Methoden.

Nachdem nun geklärt ist, wie die Handler verwaltet werden, wollen wir ihre Instantiierung näher betrachten. Alle Handler, welche die im Abschnitt 6.1.2 beschriebene Schnittstelle `I_Handler` implementieren, sollen von unserem Service eingesetzt werden können. Das bedeutet, daß wir ihren Klassennamen setzen können müssen; daher ver-

fügt `I_Service` über die Methoden `setHandlerClassname()` und `getHandlerClassname()`. Um eine korrekte Instantiierung gewährleisten zu können, muß die angegebene Handlerklasse über einen argumentlosen Konstruktor verfügen.

| Methode | Bedeutung |
|---|---|
| `Object getAttribute(String name)` | Gibt ein Attribut des Services zurück. |
| `Enumeration getAttributeNames()` | Gibt eine Aufzählung der Attributnamen zurück. |
| `InetAddress getBindAddress()` | Liefert die Internetadresse, an die dieser Service gebunden wurde. |
| `I_Handler getHandler() throws HandlerException` | Stellt einen Handler zur Verfügung. Falls dabei etwas schiefgeht, wird eine `HandlerException` ausgelöst. |
| `String getHandlerClassname()` | Gibt den Klassennamen der Handler an. |
| `int getMajorVersion()` | Gibt die Hauptversionsnummer des Services zurück. Ein Service der Version 2.1 muß hier eine 2 zurückgeben. |
| `int getMaxHandlerThreads()` | Liefert die maximal mögliche Anzahl aktiver Handler-Threads. |
| `int getMinHandlerThreads()` | Liefert die minimale Anzahl im Pool bereitstehender Threads. |
| `int getMinorVersion()` | Gibt die Unterversionsnummer des Services zurück. Ein Service der Version 2.1 muß hier eine 1 zurückgeben |
| `String getName()` | Gibt den Namen dieses Services zurück. |
| `int getPort()` | Gibt die Portnummer dieses Services zurück. |
| `int getSoTimeout()` | Maximale Zeit in Millisekunden, die der verwendete Socket bei Ein-/Ausgabe-Operationen blockiert. |
| `boolean isAlive()` | Gibt an, ob der Service läuft. |
| `void recycleHandler(I_Handler handler)` | Nimmt einen Handler- Thread wieder im Pool auf. |
| `void restart() throws ServerException` | Startet den Service neu. Üblicherweise wird einfach `stop()` und `start()` aufgerufen. |
| `void setAttribute(String name, Object o)` | Setzt ein Attribut. |
| `void setBindAddress(InetAddress aBindAddress)` | Setzt die IP-Adresse, an die dieser Service gebunden werden soll. Das Setzen der Addresse kann nur erfolgen, wenn der Service noch nicht läuft. |

*Tabelle 6.1: Die Schnittstelle `de.webapp.Framework.Server.I_Service`*

| Methode | Bedeutung |
|---|---|
| `void setHandlerClassname(String klassenname)` | Setzt den Klassennamen der zu instantiierenden Handler. |
| `void setMaxhandlerThreads(int anzahl)` | Setzt die maximale Anzahl aktiver Handler-Threads. |
| `void setMinHandlerThreads(int anzahl)` | Setzt die minimale Anzahl im Pool bereitstehender Threads. |
| `void setName(String name)` | Setzt den Namen des Services. |
| `void setPort(int aPort)` | Setzt den Port, an dem der Service horchen soll. Kann nur gesetzt werden, wenn der Service nicht läuft. |
| `void setSoTimeout(int zeit)` | Setzt die maximale Zeit in Millisekunden, die der verwendete Socket bei Ein-/Ausgabe-Operationen blockiert. |
| `void start() throws ServerException` | Startet den Service. |
| `void stop() throws ServerException` | Stoppt den Service. |

*Tabelle 6.1: Die Schnittstelle `de.webapp.Framework.Server.I_Service` (Fortsetzung)*

### 6.1.2 Definition der Handler-Schnittstelle

Im Gegensatz zur Service-Schnittstelle, fällt die Handler-Schnittstelle eher bescheiden aus. Letztlich wird lediglich der Zugriff auf den Service, die Service-Attribute sowie die Lebenszyklusmethoden `init()` und `destroy()` ermöglicht.

`init()` muß aufgerufen werden, bevor der Handler in irgendeiner Weise benutzt wird. Die Methode `destroy()` sollte aufgerufen werden, wenn der Handler nicht mehr benutzt wird. Sie muß dafür sorgen, daß sämtliche Ressourcen, die durch den Handler gebunden wurden, wieder freigegeben werden.

| Methode | Bedeutung |
|---|---|
| `void destroy()` | Gibt eventuell belegte Ressourcen dieses Handlers explizit frei. |
| `I_Service getService()` | Gibt den Service dieses Handlers zurück. Dieser muß zuvor über `init()` gesetzt worden sein. |
| `Object getServiceAttribute(String name)` | Proxymethode, die auf die Attribute des Services zugreift |
| `void init(I_Service)` | Initialisiert diesen Handler. |
| `void setServiceAttribute(String name, Object o)` | Proxymethode, die es erlaubt, ein Attribut des Services zu setzen. |

*Tabelle 6.2: Die Schnittstelle `de.webapp.Framework.Server.I_Handler`*

## 6.2 UDP- und TCP-Services

Was beide Schnittstellen nicht definieren, ist die Art und Weise ihrer Kommunikation. Das liegt größtenteils daran, daß im Paket `java.net` keine abstrakte Oberklasse beziehungsweise keine Schnittstelle für TCP-Sockets und UDP-Sockets realisiert ist. Um die übertragenen Daten vom Service an seinen Handler zu übergeben, müssen wir einen `I_TCPService` und einen `I_TCPHandler` sowie einen `I_UDPService` und einen `I_UDPHandler` definieren. In beiden Fällen werden jeweils die protokollspezifischen Methoden hinzugefügt.

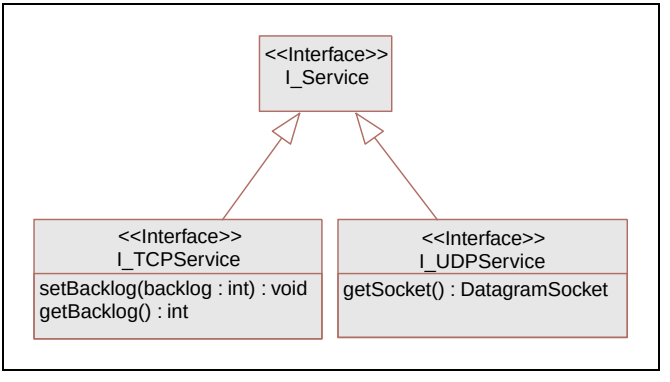


*Abbildung 6.2: Beziehung zwischen `I_TCPService`, `I_UDPService` und `I_Service`*

`I_TCPService` erweitert `I_Service` lediglich um Backlog-Methoden. Als Backlog wird die Schlange (Queue) ankommender Verbindungen bezeichnet. `I_TCPService` verfügt über Methoden, die maximale Länge der Schlange zu setzen beziehungsweise zu erfragen. Ist die Schlange voll und ein Client versucht einen Verbindungsaufbau, wird die Verbindung abgelehnt.

Da UDP ein verbindungsloses Protokoll ist, kann es keinen Backlog für Verbindungen geben. Statt dessen könnte man Methoden zur Manipulation der Sende- und Empfangspufferspeicher in die Schnittstelle mit aufnehmen. Diese Methoden sind jedoch auch über den verwendeten `DatagramSocket` zugänglich, zu dem die Schnittstelle `I_UDPService` über die Methode `getSocket()` Zugang gewährt. Daher wollen wir hier darauf verzichten.

Die Schnittstellen `I_UDPHandler` und `I_TCPHandler` enthalten jeweils nur eine Methode. Im Fall von `I_UDPHandler` heißt diese `handlePacket(DatagramPacket)`, bei `I_TCPHandler` ist es `handleConnection(Socket)`.

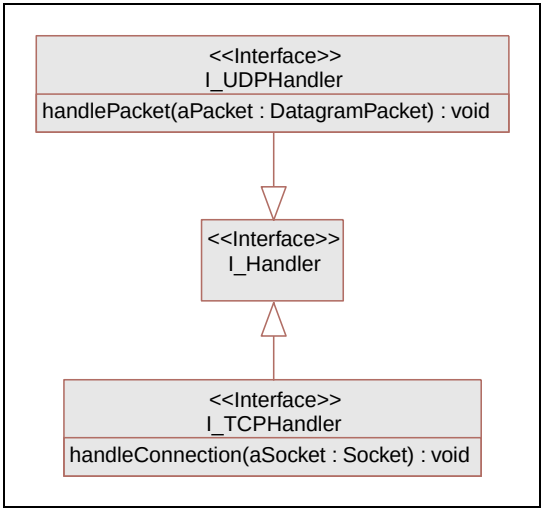


*Abbildung 6.3: Beziehung zwischen `I_TCPHandler`, `I_UDPHandler` und `I_Handler`*

## 6.3 Interaktion zwischen Service und Handler

Auch wenn sich die Schnittstellen unterscheiden, so ist der Kommunikationsalgorithmus (Abbildung 6.4) doch weitgehend der gleiche:

1. Eine Verbindung mit einem Client wird aufgebaut beziehungsweise ein Datagramm wird angenommen.
2. Über `getHandler()` wird ein Handler für diese Verbindung besorgt. Dieser Handler muß bereits initialisiert sein.
3. Die Methode `handleConnection(Socket)` beziehungsweise `handlePacket(DatagramPacket)` wird aufgerufen.
4. `handleConnection(Socket)` beziehungsweise `handlePacket(DatagramPacket)` antworten dem Client.
5. Die Methode `recycleHandler(I_Handler)` des Services wird vom aufgerufenen Handler aufgerufen. Danach darf der Handler nichts mehr tun.

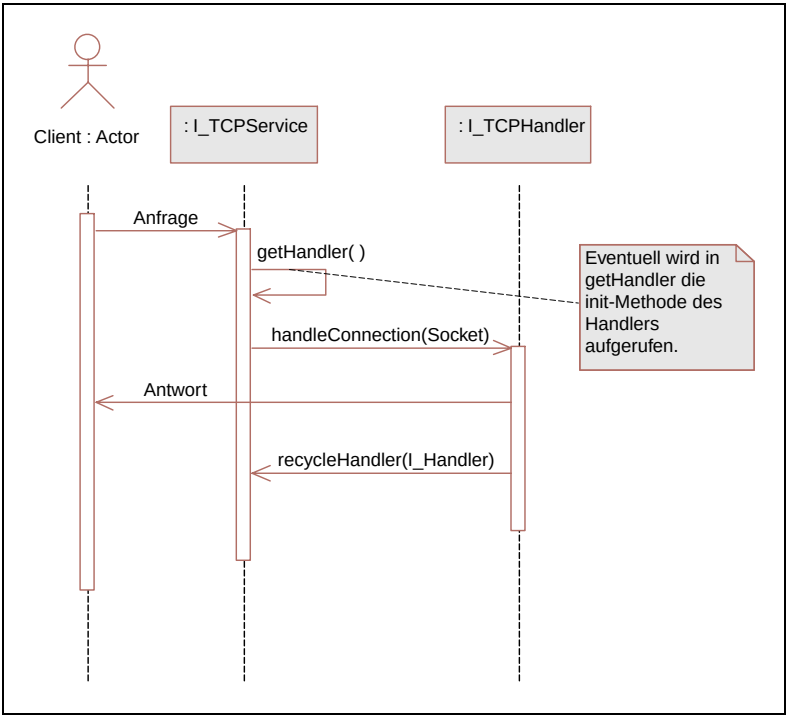


*Abbildung 6.4: Sequenzdiagramm eines TCP-Services*

## 6.4 Handler-Recycling

Der interessanteste Teil des Server-Pakets ist sicherlich das Handler-Management. Wie in Abschnitt 6.1.1 schon erwähnt, lohnt es sich, einmal initialisierte Handler wiederzuverwenden. Daher wollen wir uns die Realisierung in den Klassen `Service`, `Handler`, `TCPService` und `TCPHandler` genauer anschauen.

### 6.4.1 Service-Sicht

Trifft eine Anfrage beim Service ein, so fordert er mittels `getHandler()` einen initialisierten Handler an. Da dieser Vorgang für TCP und UDP gleich ist, befindet sich die Realisierung in der Klasse `Service`. Ebenso befindet sich die Methode `recycleHandler(I_Handler)` in dieser Klasse (Listing 6.1).

Um zu verstehen, was `getHandler()` leistet, müssen wir zunächst die Klasse `HandlerPool` erklären. Der Pool übernimmt einen großen Teil der Verwaltung der Handler (Abbildung 6.5). Über die Methoden `get()` und `add(Object)` kann man einen Handler aus dem Pool erlangen beziehungsweise an den Pool zurückgeben. Zudem lassen sich beim Pool Handler registrieren sowie die Anzahl aller registrierten Handler über die

Methode `countRegisteredHandlers()` abfragen. Auf diese Weise haben wir jederzeit einen Überblick über die Anzahl aller Handler. Die Methode `clear()` entfernt alle Handler aus dem Pool und ruft ihre `destroy()`-Methoden auf.

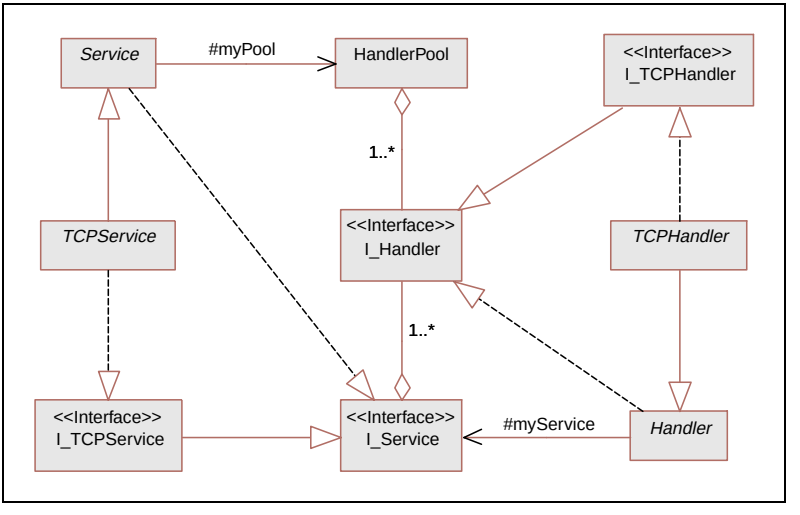


*Abbildung 6.5: Zusammenhänge zwischen Schnittstellen und Klassen im Server-Paket*

Nachdem dies geklärt ist, zurück zur Methode `getHandler()`. Zunächst wird mittels `myPool.get()` versucht, aus dem Pool einen bereits initialisierten Handler zu erlangen. Ist dieser Versuch erfolgreich, können wir den erlangten Handler zurückgeben. Haben wir nicht so viel Glück, müssen wir zunächst prüfen, ob die Anzahl der registrierten Handler noch kleiner als die maximal erlaubte Anzahl an Handler-Threads ist. Ist dies der Fall, können wir einen neuen Handler instantiieren, über `init()` initialisieren und per `registerHandler(I_Handler)` beim Pool registrieren.

Da `registerHandler(I_Handler)` das Ergebnis von `countRegisteredHandlers()` beeinflußt, muß sich dieser Teil in einem bezüglich des Pools synchronisierten Abschnitt befinden.

Ist bereits die maximale Anzahl an Handlern instantiiert und registriert worden, müssen wir mittels `myPool.wait()` warten, bis ein Handler sich wieder zum Pool hinzufügt. Das geschieht über den Aufruf der Methode `recycleHandler(I_Handler)`. Damit wir benachrichtigt werden, wenn wieder ein Handler verfügbar ist, wird nach dem Hinzufügen über `myPool.add(I_Handler)` noch `myPool.notify()` aufgerufen.

```
/** enthält die Handler */
protected HandlerPool myPool = new HandlerPool();

public I_Handler getHandler() throws HandlerException {
   I_Handler aHandler = null;
   aHandler = (I_Handler)myPool.get();
   if (aHandler != null) return aHandler;
   synchronized (myPool) {
      if (myPool.countRegisteredHandlers() <= getMaxHandlerThreads()) {
         try {
            aHandler = (I_Handler)Class.forName(getHandlerClassname()).newInstance();
            aHandler.init(this);
            myPool.registerHandler(aHandler);
         }
         catch (Exception e) {
            throw new HandlerException(e);
         }
         return aHandler;
      }
      while ((aHandler = (I_Handler)myPool.get()) == null) {
         try {
            myPool.wait();
         }
         catch (InterruptedException ie) {}
      }
   }
   return aHandler;
}

public void recycleHandler(I_Handler aHandler) {
   synchronized (myPool) {
      myPool.add(aHandler);
      myPool.notify();
   }
}
```

*Listing 6.1: Die Methoden `getHandler()` und `recycleHandler(I_Handler)` aus `de.webapp.Framework.Server.Service`*

Wurde über `getHandler()` ein Handler erlangt, ist dieser zwar initialisiert, jedoch noch nicht aktiviert. In der Klasse `TCPService` wird der `I_Handler` zunächst in einen `I_TCPHandler` umgeformt (Typecast), um anschließend dessen `handleConnection(Socket)` aufzurufen. Ähnliches passiert in `UDPService` – hier wird `handlePacket(DatagramPacket)` eines `I_UDPHandlers` aufgerufen.

In Listing 6.2 ist der gesamte Vorgang von der Verbindungsannahme bis zum Aufruf eines Handlers abgedruckt. Der Ausschnitt stammt aus der Klasse `TCPService`, die genau wie `Service` und `UDPService` die Schnittstelle `Runnable` implementiert. Daher enthält der Ausschnitt eine `run()`-Methode. Zudem enthält er eine `init()`-Methode. In der

(nicht abgedruckten) `start()`-Methode von `Service` werden die beiden Methoden nacheinander aufgerufen.

#### *Initialisierung*

Zunächst wird in der `init()`-Methode ein `ServerSocket` instantiiert. Ist keine Adresse definiert, an die der Service gebunden werden soll, wird er an alle verfügbaren Adressen gebunden. In der Log-Ausgabe wird dies durch die Adresse 0.0.0.0 angezeigt. Anschließend wird der `SO_TIMEOUT` gesetzt. Dabei handelt es sich um die maximale Dauer in Millisekunden, die der Socket bei Lese- und Schreiboperationen blockiert. Nach Verstreichen dieser Zeit wird ein `java.io.InterruptedIOException` ausgelöst.

#### *Service-Phase*

Nachdem der Socket initialisiert ist, wird die `run()`-Methode aufgerufen. Ihr zentraler Bestandteil ist eine `while`-Schleife, in der ankommende Verbindungen angenommen und an Handler weitergegeben werden.

Dreierlei Ausnahmen müssen in dieser Schleife behandelt werden: Eine `HandlerException`, da diese von `getHandler()` ausgelöst wird, falls kein Handler erlangt werden konnte. In diesem Fall sollte der Socket geschlossen werden, da er nicht weiter bearbeitet werden kann. Die zweite mögliche Ausnahme ist die bereits erwähnte `InterruptedIOException`. Wird sie ausgelöst, rufen wir die Methode `handleSoTimeout()` der Oberklasse `Service` auf. Sie sorgt dafür, daß der Pool auf die minimal benötigte Anzahl an Handlern reduziert wird. Dies ist sinnvoll, wenn der Service längere Zeit keine Anforderungen angenommen hat.

Bleibt noch die `SocketException`: Sie wird unter anderem ausgelöst, wenn die `close()`-Methode des `ServerSockets` aufgerufen wird, während dieser über `accept()` oder `receive()` auf Daten wartet. Ein solcher Aufruf kann absichtlich geschehen, wenn der Service gestoppt werden soll. Daher loggen wir nur dann einen Fehler, wenn das Flag `stopped` gleich `false` ist.

#### *Stoppen des Services*

Insgesamt gibt es drei Möglichkeiten, die Schleife zu unterbrechen und somit den Service zu stoppen:

1. Eine Ausnahme innerhalb der Schleife wird nicht behandelt.
2. Der Socket wurde geschlossen und `stopped` ist `true`.
3. Der `SO_TIMEOUT` ist verstrichen und `stopped` ist `true`.

In jedem Fall gibt es noch einige Aufräumarbeiten zu erledigen. Das Flag `stopped` muß auf `true` gesetzt, der Pool geleert und der Socket geschlossen werden. Dies geschieht im `finally`-Block.

```
public synchronized void init() throws IOException, ServerException
   {
      super.init();
      if(Log.isLog(getName()))
      {
         if(getBindAddress() != null)
            Log.log("Trying to bind service to " + getBindAddress() + ":" +
                    getPort(),getName());
         else
            Log.log("Trying to bind service to port " + getPort() + ".",getName());
      }
      myServerSocket = new ServerSocket(getPort(),getBacklog(),getBindAddress());
      // Set Blocking-Timeout
      myServerSocket.setSoTimeout(getSoTimeout());
      // Set addresse, if not known before.
      if(getBindAddress() == null)
         myBindAddress = myServerSocket.getInetAddress();
      // Set port, if not known before.
      if(getPort() == 0)
         myPort = myServerSocket.getLocalPort();
      if(Log.isLog(getName()))
         Log.log("Bound service to " + getBindAddress() + ":" + getPort(),getName());
      if(Log.isLog(getName()))
         Log.log("Listening...",getName());
   }

   public void run()
   {
      try
      {
         I_TCPHandler aHandler;
         Socket aSocket = null;
         while(!stopped)
         {
            try
            {
               aSocket = null;
               aSocket = myServerSocket.accept();
               aHandler = (I_TCPHandler)getHandler();
               aHandler.handleConnection(aSocket);
            }
            catch(HandlerException he)
            {
               if(aSocket != null)
                  aSocket.close();
               if(Log.isLog(getName()))
                  Log.log(he.toString(),getName());
            }
            catch(InterruptedIOException ie)
            {
               handleSoTimeout();
```

```
            }
            catch(SocketException se)
            {
               //  is initiated, if socket is closed during stopping.
               if(!stopped && Log.isLog(getName()))
                  Log.log(se.getMessage(),getName());
            }
         }
      }
      catch(IOException ioe)
      {
         if(Log.isLog(C_Log.ERROR,getName()))
            Log.log(ioe,C_Log.ERROR,getName());
         throw new ServerRuntimeException(ioe);
      }
      finally
      {
         stopped = true;
         myPool.clear();
         try
         {
            close();
         }
         catch(IOException ioe)
         {
            if(Log.isLog(C_Log.ERROR,getName()))
               Log.log(ioe,C_Log.ERROR,getName());
            throw new ServerRuntimeException("Failed to close socket.",ioe);
         }
      }
   }
```

*Listing 6.2: Methoden `init()` und `run()` aus `de.webapp.Framework.Server.TCPService`*

### 6.4.2 Lebenszyklus des Handlers

Um den Anforderungen des Services zu genügen, muß der Handler eine zentrale Eigenschaft besitzen: Er muß wiederbenutzbar sein. Um eine asynchrone Ausführung zu ermöglichen, sind Handler oft als Thread realisiert. Threads sind jedoch nicht ohne weiteres wiederbenutzbar. Wie in Kapitel 1 (Abbildung 1.1) deutlich wird, gibt es keinen Zustandsübergang von »tot« nach »neu«. Ein einmal abgelaufener Thread ist nicht wiederzubeleben. Will man einen Thread mehrfach benutzen, müssen *innerhalb* der `run()`-Methode nacheinander *mehrere* Anfragen abgearbeitet werden. Zwischen den Anfragen sollte der Thread in einer Ruheposition verharren.

Listing 6.3 zeigt die entsprechenden Codeausschnitte aus den Klassen `TCPHandler` und `Handler`. Der besseren Übersicht wegen, sind sie hier zusammen abgedruckt.

### Initialisierung

Zu Beginn des Handler-Lebenszykluses wird die `init()`-Methode aufgerufen. Hier wird in der Instanzvariable `myService` der Service hinterlegt und die Methode `start()` aufgerufen. Da `Handler` von `java.lang.Thread` erbt, startet das den Thread. Somit wird die `run()`-Methode aufgerufen. Sie prüft als erstes, ob der Handler bereits gestoppt worden ist. Ist dies der Fall, kehrt die `run()`-Methode zurück, ohne etwas getan zu haben – der Thread ist tot. Ist jedoch der Thread noch nicht gestoppt, wird überprüft, ob eine Verbindung vorliegt. Wenn dies so ist, wird die `service()`-Methode aufgerufen. Diese ist in der Klasse `Handler` nur abstrakt definiert und sollte zum Realisieren eines Services überschrieben werden. Nachdem `service()` seine Pflicht getan hat, wird die Verbindung geschlossen und `mySocket` gleich `null` gesetzt. Der Handler ist jetzt wieder in einem benutzbaren Zustand, daher dürfen wir die `recycleHandler(I_Handler)`-Methode des Services aufrufen. Der anschließende Aufruf von `wait()` versetzt den Handler in Warteposition. Wichtig ist zu beachten, daß `wait()` auch aufgerufen wird, wenn keine Verbindung vorlag. Dies ist insbesondere direkt nach der Initialisierung der Fall.

### Service-Phase

Wie aber kommt der Handler zu seiner Verbindung? Dazu muß ein `TCPService` die `handleConnection(Socket)`-Methode eines Handlers aufrufen. Dort wird die Verbindung in der Instanzvariable `mySocket` hinterlegt und eventuell wartende Threads per `notify()` benachrichtigt. Befindet sich ein Handler in Warteposition, wacht er auf und überprüft, ob eine Verbindung vorliegt. Da dies jetzt der Fall ist, wird die Anfrage von `service()` abgearbeitet.

### Stoppen des Handler-Threads

Um einen Handler-Thread endgültig zu stoppen, muß die `destroy()`-Methode aufgerufen werden. Sie setzt das Flag `stopped` auf `true` und ruft `notify()` auf. Befindet sich der Handler gerade in Warteposition, wird er benachrichtigt und beendet die `run()`-Methode. Führt der Handler gerade einen Request aus, beendet er diesen und anschließend die `run()`-Methode. Der Handler ist nun »tot«, kann also nicht wiederverwendet werden.

```
// from TCPHandler
protected Socket mySocket = null;

// from Handler
protected I_Service myService = null;
protected boolean stopped = false;
protected boolean started = false;

public void init(I_Service aService) throws HandlerException
{
   if(aService == null)
```

```
         throw new IllegalArgumentException("Service can't be null.");
      myService = aService;
      start();
      synchronized (this) {
       try {
            while (!started) wait(1);
         } catch (InterruptedException ie) {}
      }
   }

   // from TCPHandler
   public synchronized void handleConnection(Socket aSocket)
   {
      if (Log.isLog(Log.METHOD, myService.getName())) {
    Log.log("TCPHandler#handleConnection(): " + getName(), Log.METHOD,
myService.getName());
      }
      mySocket = aSocket;
      notify();
   }

   public void run()
   {
      try
      {
         synchronized (this) {
            started=true;
            wait();
         }
      }
      catch(InterruptedException ie)
      {
      }
      while(!stopped)
      {
         if(mySocket != null)
         {
            try
            {
               service();
            }
            catch(Throwable t)
            {
               if(Log.isLog(C_Log.ERROR,getService().getName()))
                  Log.log(t.toString(),C_Log.ERROR,getService().getName());
            }
            try
            {
               mySocket.close();
            }
            catch(Exception e)
```

```
            {
            }
            mySocket = null;
            if (!stopped) {
      try {
synchronized (this) {
                    myService.recycleHandler(this);
   wait();
                }
              }
              catch(InterruptedException ie) {}
            }
        } else {
            try {
       if(!stopped) {
                  synchronized (this) {
                     wait();
  }
              }
    } catch(InterruptedException ie) {}
       }
     }
  }

  // from Handler
  public abstract void service() ;

  public synchronized void destroy()
  {
     stopped = true;
     notify();
  }
```

*Listing 6.3: Codeausschnitte aus `TCPHandler` und `Handler`*

## 6.5 Echo-Service

Als Beispielanwendung für die Server-Klassen programmieren wir einen Echo-Service. Er soll sich dadurch auszeichnen, daß er über eine TCP-Verbindung eine Zeile liest und diese zurückschreibt. Das Schlüsselwort »bye« soll die Verbindung beenden.

Als erstes müssen wir die Klasse `TCPService` instantiieren und die nötigen Parameter für diesen Service setzen. Glücklicherweise verfügt `TCPService` über einige Methoden, die dies erleichtern. So instantiiert die `main(String[])`-Methode eine `TCPService`-Klasse und setzt den Namen gleich dem ersten Kommandozeilenparameter. Anschließend wird die `start()`-Methode der Instanz aufgerufen. Innerhalb von `start()` erfolgt ein Aufruf der `init()`-Methode, die wiederum `loadConfiguration()` ausführt. `loadConfiguration()` schließlich fragt den Konfigurationsmanager (Kapitel 4) nach einer passenden

Konfiguration für den Service. Als Schlüssel benutzt er dazu den Servicenamen. Die Konfiguration (Listing 6.4) wird gelesen und gesetzt – dann wird der Service gestartet.

```
{
   // Portnummer
   PORT = 8090
   // maximale Anzahl gepufferter Verbindungen
   BACKLOG = 50
   // maximale Anzahl an HandlerThreads
   MAXHANDLERTHREADS = 50
   // minimale Anzahl an HandlerThreads
   MINHANDLERTHREADS = 5
   // Klassenname der Handler
   HANDLERCLASSNAME = "de.webapp.Examples.Server.EchoHandler"
   // Zeit in ms , die ohne Verbindung verstreichen muß,
   // bis der HandlerPool auf seine Minimalgröße gestutzt wird.
   SO_TIMEOUT = 30000
   // Hauptversionsnummer
   MAJORVERSION = 0
   // Unterversionsnummer
   MINORVERSION = 1
}
```

*Listing 6.4: Konfigurationsdatei des EchoServices, `webapp/projects/Examples/Server/EchoService.cfg`*

Letztlich müssen wir also keine neue Service-Klasse schreiben, sondern nur Konfigurationsdateien anlegen. Beim Handler dagegen müssen wir das Service-Verhalten definieren. Dazu muß die Methode `service()` überschrieben werden. Listing 6.5 zeigt die sehr einfache Realisierung des `EchoHandlers`:

```
package de.webapp.Examples.Server;

import de.webapp.Framework.Server.*;
import java.net.Socket;
import java.io.*;

public class EchoHandler extends TCPHandler {

public void service() {
   try {
      BufferedReader in = new BufferedReader(
         new InputStreamReader(getSocket().getInputStream())
      );
      BufferedWriter out = new BufferedWriter(
         new OutputStreamWriter(getSocket().getOutputStream())
      );
      String line;
      while ((line = in.readLine()) != null
            && !line.toLowerCase().startsWith("bye")) {
         out.write(line);
         out.newLine();
```

```
            out.flush();
         }
      }
      catch (IOException ioe) {
         ioe.printStackTrace();
      }
   }

} // Ende der Klasse
```

*Listing 6.5: Die Klasse `de.webapp.Examples.Server.EchoHandler`*

Zum Start des Services müssen wir der Virtuellen Maschine den Ort der Datei `Registry.cfg` übergeben, sofern diese nicht im selben Verzeichnis liegt. In `Registry.cfg` sollte zudem unter dem Namen »EchoService« ein Eintrag auf die Konfigurationsdatei des Services verweisen (Listing 6.6). Alle Einträge sind in der Software zum Buch bereits vorgenommen. In Verzeichnis `/webapp/bin` befindet sich ein Skript `EchoService`, das Ihnen den Start erleichtern soll.

Um den gestarteten Echo-Service zu testen, richten Sie über Telnet eine Verbindung mit der Adresse des Rechners ein, auf dem der Echo-Service läuft. Als Port sollten Sie 8090 angeben.

```
{
   ...
   EchoService = {
      PATH = "../projects/Examples/Server";
      FILE = "EchoService.cfg";
      RELATIVE2CFGROOT = true ;
   };
   ...
}
```

*Listing 6.6: Möglicher Eintrag in `Registry.cfg`*

## 6.6 Metadienst

Vielleicht ist Ihnen aufgefallen, daß der `EchoService` sich zwar wunderbar über die `main()`-Methode starten läßt, jedoch kein Weg vorgesehen ist, ihn auch ordnungsgemäß wieder zu stoppen. Da Java die Verarbeitung von Betriebssystem-Signalen nicht unterstützt, müssen wir einen anderen Weg finden, einem Service ein Stop-Signal zu senden. Mit dem uns zur Verfügung stehenden Server-Paket liegt nichts näher, als einen Meta-Service zu schreiben, über den wir andere Dienste starten und stoppen können. Das wollen wir im folgenden tun.

### 6.6.1 Service

Um einen Dienst starten zu können, müssen wir den Namen seines Eintrages in der Datei `Registry.cfg` sowie seinen Klassennamen kennen. Beides sollte in einer Konfigurationsdatei gespeichert sein, die unser `AdminService` bei seiner Initialisierung liest. Sinnvoll erscheint es uns außerdem, Diensten ein Attribut zuweisen zu können, das den `AdminService` dazu veranlaßt, den jeweiligen Dienst direkt beim Start mitzustarten. In Listing 6.7 sehen Sie eine entsprechende Konfigurationsdatei.

```
{
   EchoService = {
      Class = "de.webapp.Examples.Server.EchoService";
      AutoStart = true;
   }
   EinAndererService = {
      Class = "EinAndererService";
      AutoStart = false;
   }
}
```

*Listing 6.7: Die Konfigurationsdatei `services.cfg` des `AdminServices`*

Natürlich sollte das Starten und Stoppen von Diensten nur Nutzungsberechtigten gestattet sein. Um unser Beispiel einfach zu gestalten, müssen wir auf aufwendige Zugangsberechtigungsprüfungen verzichten. Um jedoch nicht jedem uneingeschränkt Zugang zum `AdminService` zu ermöglichen, überprüfen wir, ob die IP-Adresse des Clients in einer Liste von Adressen enthalten ist, denen wir pauschal den Zugang zum Service erlauben. Der Einfachheit halber schreiben wir diese Liste in die Server-Konfiguration, so daß die Datei `server.cfg` für unseren `AdminService` wie in Listing 6.8 aussieht:

```
{
   PORT = 9090
   BACKLOG = 50
   MAXHANDLERTHREADS = 50
   MINHANDLERTHREADS = 1
   HANDLERCLASSNAME = "de.webapp.Framework.Server.AdminHandler"
   VALID_ADDRESSES = ("localhost","127.0.0.1")
   MAJORVERSION = 0
   MINORVERSION = 1
}
```

*Listing 6.8: Datei `server.cfg` des `AdminServices`*

Listing 6.9 zeigt die Realisierung des `AdminServices`. Beachtenswert ist, daß die Klasse aus nur neun Methoden besteht.

Um die Anforderungen erfüllen zu können, benötigen wir eine Liste erlaubter Adressen, eine Hashtabelle mit den aktiven Diensten sowie die Konfiguration der zu startenden Dienste. Zum Start des Services muß also die Konfiguration gelesen werden.

Da die Methode `init()` aus `TCPService` beziehungsweise `Service` ohnehin `loadConfiguration()` aufruft, macht es Sinn, `loadConfiguration()` zu ergänzen. Der zusätzliche Code überprüft die Liste der gültigen Adressen und legt diese anschließend im Vektor `myValidAddresses` ab. Zudem wird noch die Konfigurationsdatei `services.cfg` gelesen und in `myServiceConfiguration` geladen. Um eine Änderung der Datei mitzubekommen, setzen wir das Attribut `AutoReload` des `Configuration`-Objektes auf `true`. Das stellt sicher, daß bei jedem Versuch, einen Eintrag der Konfiguration zu lesen, überprüft wird, ob diese sich geändert hat. Ist dies der Fall, wird sie automatisch neu geladen.

Nachdem über `super.init()` die Konfiguration geladen und der `AdminService` gestartet wurde, registriert der `AdminService` sich selbst in der Hashtabelle der Services und startet die Dienste mit der Eigenschaft `AutoStart`. Dazu wird die Methode `autoStart()` aufgerufen, die für die zu startenden Dienste `startService(String)` aufruft. Der Methode `startService(String)` wird als Parameter der Name eines Dienstes übergeben, der zu starten ist. Falls der benannte Dienst nicht bereits gestartet ist, wird er instantiiert, gestartet und der Hashtabelle der Services hinzugefügt. Analog funktioniert `stopService(String)`. Hier wird der benannte Service der Hashtabelle der Services entnommen und seine `stop()`-Methode aufgerufen, sofern er überhaupt läuft.

`stopAllServices()` schließlich stoppt alle Services außer dem `AdminService`. Der kann nur durch den Aufruf seiner eigenen `stop()`-Methode gestoppt werden, die jedoch wiederum `stopAllServices()` aufruft.

Darüber hinaus enthält der Service nur noch die Methode `isValidAddress(InetAddress)`, die überprüft, ob eine Internet-Adresse in der Liste privilegierter Adressen enthalten ist.

```
package de.webapp.Framework.Server;

import java.net.*;
import java.util.*;
import java.io.*;

import de.webapp.Framework.Log.*;
import de.webapp.Framework.ConfigManager.*;

/** MetaService */
public class AdminService extends TCPService {
/** Enthält die gültigen Einlog-Adressen */
protected Vector myValidAddresses = new Vector();
/** Enthält die verwalteten Services */
protected Hashtable myServices = new Hashtable();
/** Konfiguration dieses Services */
```

```
protected Configuration myServiceConfiguration = null;
/** Gibt an, ob sich ein Nutzer von einer Adresse aus einloggen darf*/
public boolean isValidAddress(InetAddress aAddress) {
   return myValidAddresses.contains(aAddress);
}
/** Initialisiert diesen Service */
public synchronized void init() throws ServerException, IOException{
   super.init();
   myServices.put(getName(), this);
   autoStart();
}
/** Stoppt zunächst alle verwalteten Services und dann sich selbst */
public synchronized void stop() throws ServerException {
   stopAllServices();
   super.stop();
}
/** Startet alle Services, die automatische gestartet werden sollen */
protected void autoStart() throws ServerException {
   Hashtable config = (Hashtable)myServiceConfiguration.getConfigData();
   Enumeration e = config.keys();
   while (e.hasMoreElements()) {
      String name = (String)e.nextElement();
      Hashtable entry = (Hashtable)config.get(name);
      if (Boolean.TRUE.equals(entry.get("AutoStart"))) {
         startService(name);
      }
   }
}
/** Startet einen Service anhand seines Namens */
public synchronized void startService(String name) throws ServerException {
   I_Service aService = (I_Service)myServices.get(name);
   if (aService == null) {
      try {
         Class serviceClass = Class.forName(
            (String)myServiceConfiguration.getElement(name + ";Class"));
         aService = (I_Service)serviceClass.newInstance();
         myServices.put(name, aService);
      }
      catch (Throwable t) {
         throw new ServerException("Instantiierungsfehler von " + name, t);
      }
   }
   if (!aService.isAlive()) {
      aService.setName(name);
      aService.start();
   }
   else {
      throw new ServerException("Service " + name + " läuft bereits.");
   }
}
/** Stoppt einen Service anhand seines Namens */
```

```
public synchronized void stopService(String name) throws ServerException {
   I_Service aService = (I_Service)myServices.get(name);
   if (aService != null) {
      if (!aService.isAlive()) throw new ServerException("Service " + name
         + " läuft nicht.");
      aService.stop();
      myServices.remove(name);
   }
   else {
      throw new ServerException("Service " + name + " existiert nicht.");
   }
}
/** Stoppt alle Services außer dem AdminService */
protected void stopAllServices() throws ServerException {
   Enumeration e = myServices.elements();
   while (e.hasMoreElements()) {
      I_Service aService = (I_Service)e.nextElement();
      if (aService != this && aService.isAlive()) aService.stop();
   }
   myServices.clear();
}
/** Lädt die Konfiguration */
public void loadConfiguration() throws ServerException {
   super.loadConfiguration();
   Configuration conf = getConfiguration();
   Vector v = (Vector)conf.getElement("VALID_ADDRESSES");
   myValidAddresses.removeAllElements();
   Enumeration e = v.elements();
   while (e.hasMoreElements()) {
      String aAddress = (String)e.nextElement();
      try {
         myValidAddresses.addElement(InetAddress.getByName(aAddress));
      }
      catch (UnknownHostException uhe) {
         if (Log.isLog(getName())) Log.log("Unbekannter Host: "
            + aAddress, Log.ERROR, getName());
      }
   }
   ConfigManager cm = ConfigManager.getConfigManager();
   myServiceConfiguration = cm.getConfiguration(getName(), "services.cfg");
   myServiceConfiguration.setAutoReload(true);
}
/** Startet den Service */
public static void main(String[] args) throws ServerException {
   AdminService admin = new AdminService();
   if (args.length == 1) {
      admin.setName(args[0]);
   }
   else {
      admin.setName("AdminService");
   }
```

```
   admin.start();
}
} // Ende der Klasse
```

*Listing 6.9: Die Klasse* `AdminService`

### 6.6.2 Handler

Nachdem wir mit der Klasse `AdminService` die allgemeinen Funktionen des Services realisiert haben, fehlt uns noch ein Protokoll, um diese ansprechen zu können. Dieses soll durch den Handler realisiert werden. Dazu müssen wir es zunächst einmal definieren.

Ein Client soll in der Lage sein, Dienste zu starten und zu stoppen. Zudem erscheint es uns sinnvoll, mehr als ein Kommando pro Verbindung absetzen zu können; etwa um einen Dienst zu stoppen und anschließend wieder zu starten. Daher benötigen wir einen Befehl, der die Verbindung schließt. Prinzipiell reichen für unseren einfachen Dienst also drei Befehle: `start`, `stop` und `exit`.

Während `exit` offensichtlich parameterlos ist, muß bei `start` und `stop` jeweils noch der Name des Dienstes angegeben werden. Jeder Befehl sollte zudem durch ein Zeilenvorschubzeichen (Newline) beendet werden.

Zur Implementierung des Protokolls überschreiben wir die `service()`-Methode. In ihr prüfen wir zunächst, ob die angenommene Verbindung von einer der erlaubten Adressen aufgebaut worden ist. Dazu benutzen wir die Methode `isValidAddress(InetAddress)` des `AdminSevices`. Ist die Adresse nicht gültig, geben wir mit einem `PrintWriter` eine Fehlermeldung aus und schließen die Verbindung.

Geht die Verbindung von einer privilegierten Adresse aus, heißen wir den Nutzer willkommen und geben eine kurze Beschreibung der Nutzung aus, indem wir die Methode `usage(PrintWriter)` aufrufen.

Nachdem die Formalitäten erledigt sind, soll der Nutzer nun Befehle eingeben können. Dazu wird ein `BufferedReader` geöffnet, aus dem über eine `while`-Schleife jeweils eine Zeile gelesen wird.

Mittels eines `java.util.StringTokenizers` wird nun die gelesene Zeile in Tokens geteilt. Ist die Anzahl der Tokens Null, wird gleich die nächste Zeile gelesen. Ist sie eins und das Token gleich `exit`, wird die `service()`-Methode über `return` verlassen. Im Fall von zwei Tokens gilt es, zwischen den beiden Befehlen `start` und `stop` zu unterscheiden. Handelt es sich um `start`, wird die `startService(String)`-Methode des Services aufgerufen. Dabei wird als Argument das zweite Token übergeben. Analog wird der `stop`-Befehl ausgeführt – mit dem kleinen Unterschied, daß hier die `stopService(String)`-Methode des Services ausgeführt wird. In beiden Fällen geben wir eventuell ausgelöste `ServerExceptions` einfach aus.

Enthält die gelesene Zeile mehr als zwei Tokens und entspricht nicht den bis hier geschilderten Fällen, wird die Methode `usage()` aufgerufen und die nächste Zeile gelesen.

```
package de.webapp.Framework.Server;

import java.net.*;
import java.util.*;
import java.io.*;

import de.webapp.Framework.Log.*;

public class AdminHandler extends TCPHandler {

public void service() {
   AdminService service = (AdminService)getService();
   try {
      Socket aSocket = getSocket();
      PrintWriter out = new PrintWriter(new
            OutputStreamWriter(aSocket.getOutputStream()), true);
      if (!service.isValidAddress(aSocket.getInetAddress())) {
         out.println("You are not allowed to log in.");
      }
      else {
         out.println("Welcome to " + service.getName() + ".");
         usage(out);
         String line;
         BufferedReader in = new BufferedReader(
            new InputStreamReader(aSocket.getInputStream()));
         while ((line = in.readLine()) != null) {
            if (Log.isLog(service.getName())) Log.log(line,
               Log.MODUL, service.getName());
            Log.log(line, Log.ERROR, service.getName());
            StringTokenizer st = new StringTokenizer(line, " ");
            int count = st.countTokens();
            if (count == 0) continue;
            if (count == 1 && line.toLowerCase().equals("exit")) {
               out.println("Auf Wiedersehen!");
               return;
            }
            else if (count == 2) {
               String command = st.nextToken().toLowerCase();
               String serviceName = st.nextToken();
               try {
                  if (command.equals("start")) {
                     service.startService(serviceName);
                     out.println("Service " + serviceName + " wurde gestartet.");
                  }
                  else if (command.equals("stop")) {
                     service.stopService(serviceName);
                     out.println("Service " + serviceName + " wurde gestoppt.");
```

```
                    if (serviceName.equals(service.getName())) {
                       out.println("Auf Wiedersehen!");
                       return;
                    }
                 }
              }
              catch (ServerException se) {
                 out.println(se.toString());
              }
           }
           else {
              usage(out);
           }
        }
        in.close();
     }
     out.close();
  }
  catch (IOException ioe) {
     if (Log.isLog(service.getName()))   Log.log(ioe.toString(),
        Log.ERROR, service.getName());
  }
}

public static void usage(PrintWriter out) {
  out.println("Nutzung: exit | (start|stop) <servicename>");
}
} // Ende der Klasse
```

*Listing 6.10: Die Klasse* `AdminHandler`

### 6.6.3 Ausführung

Da der `AdminService` über eine `main()`-Methode verfügt, kann man ihn genau wie den `EchoService` von der Kommandozeile starten. Ebenfalls existiert ein Skript im Verzeichnis `/webapp/bin/` mit dem Namen `AdminService`.

Um sich mit dem `AdminService` zu verbinden, leistet das Programm Telnet hervorragende Dienste. Der `AdminService` horcht normalerweise auf Port 9090. Falls Sie nicht über ein leistungsfähiges Telnet[1] verfügen, können sie statt dessen den `AdminClient` nutzen, der sich ebenfalls über ein Skript im gleichen Verzeichnis starten läßt.

Da der `AdminService` weitgehend wie jeder andere Service behandelt wird, kann man ihn über den Befehl `stop AdminService` stoppen. In diesem Fall werden alle anderen Dienste auch gestoppt.

1. Darunter verstehen wir ein Telnet, das Verbindungen zu beliebigen Ports erlaubt. Leider gibt es immer noch mindestens einen Betriebssystem-Hersteller, der seine Betriebssysteme nicht mit einem solchen Programm ausliefert.

## 6.7 Serverbauen stark erleichtert

An den Beispielen ist deutlich geworden, wie einfach das Programmieren von Servern sein kann, wenn eine solide Grundlage gegeben ist. Das ServerKit sorgt für diese Grundlage:

- ServerKit realisiert das Service/Handler-Muster.
- Durch definierte Schnittstellen ist ein einfache Administration über einen Metaservice möglich.
- Es realisiert eine leistungsfähige, konigurierbare Thread-Verwaltung.
- Der Entwickler muß nur noch protokollspezifische Funktionalitäten realisieren.

# 7 Entwicklung der Servlet-Engine jo!

Nachdem wir mit dem Server-Paket nun über eine Grundlage zur einfachen Entwicklung von Servern verfügen, wollen wir uns die Ausführungsumgebung von Servlets einmal genauer anschauen. Dazu werden wir die wesentlichen Design-Grundlagen der Servlet-Engine jo! anhand ihrer Schnittstellen und einiger Klassen nachvollziehen. jo! ist ein Webserver, der komplett in Java geschrieben wurde. Seine Servlet-Engine (siehe Kapitel 3) realisiert die Spezifikation des Servlet-API 2.1. Darüber hinaus ist jo! in der Lage, virtuelle Hosts zu verwalten (Abbildung 7.1).

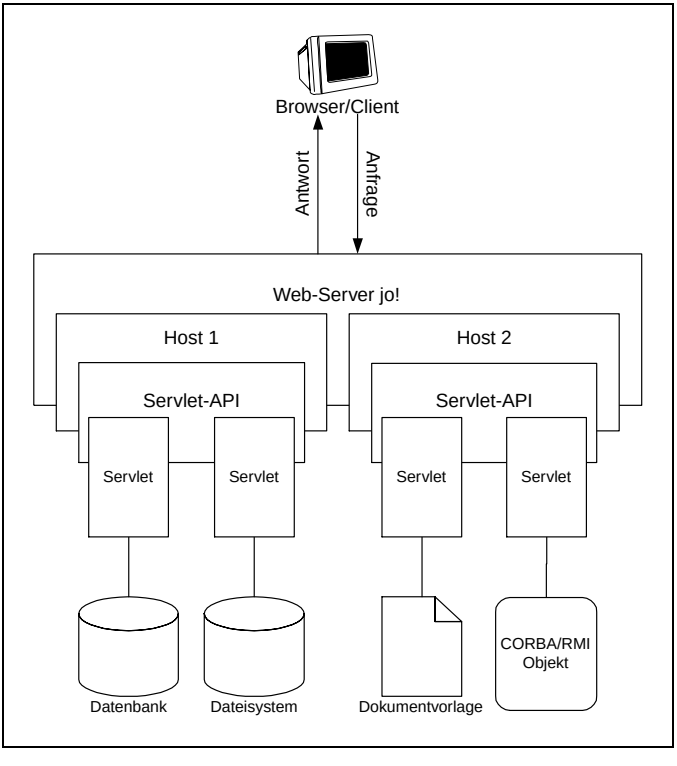


*Abbildung 7.1: Architektur von jo!*

Darunter versteht man die Fähigkeit, Anfragen an eine IP-Adresse in Anfragen an unterschiedliche Hosts umzuwandeln. Um dies zu ermöglichen, wird in einer Anfrage immer der Hostname als Kopffeld mitgeschickt.

## 7.1 Grunddesign

Genau wie im Server-Paket wollen wir unser Design von einem klassischen Service-Handler-Modell ableiten. Das bedeutet, daß ein Service-Objekt Verbindungen annimmt. Jede angenommene Verbindung wird in Form eines `Socket`-Objekts an einen Handler weitergegeben. Dieser ist für die Erfüllung des verlangten Dienstes zuständig – in unserem Fall ist dies der Aufruf eines Servlets.

Da unser Server virtuelle Hosts unterstützen soll, liegt es nahe, diese durch Objekte abzubilden. Das bedeutet, daß unser Service-Objekt eine Reihe von Host-Objekten kennen sollte. Laut Servlet-Spezifikation muß jeder virtuelle Host mindestens einen Servletkontext besitzen. Jeder Servletkontext wiederum besitzt theoretisch eine beliebige Anzahl an Servlets. Eine Servlet-Instanz kann dabei nicht mehreren Kontexten zugeteilt sein. Genauso ist eine Kontext-Instanz höchstens einem Host zugeordnet. Daher existiert immer ein eindeutiger Navigationspfad von einem Host zu einem Servlet. Es ergibt sich zwangsläufig ein Grunddesign wie in Abbildung 7.2.

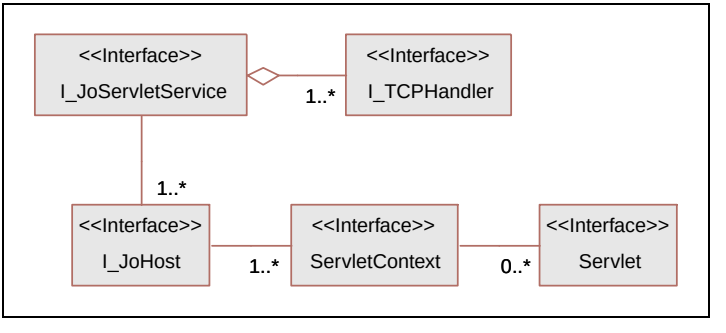


*Abbildung 7.2: Grunddesign von jo!*

Eine gegebene Anfrage wird immer nach dem selben Schema bearbeitet: Zunächst muß der Handler den richtigen Host finden, dann den richtigen Servletkontext und schließlich das gefragte Servlet. Läßt sich der Anfrage ein Servlet zuordnen, wird dieses ausgeführt und eine Antwort zurück an den Client geschickt.

## 7.2 Verfeinerung des Designs

Um der Servlet-Spezifikation gerecht zu werden, müssen wir jedoch einige Verfeinerungen vornehmen. Dies betrifft insbesondere die Handhabung von Servlets und Servletkontexten.

### 7.2.1 Servletmodel

Servlets verfügen über einen Kontext, eine Konfiguration und einen Lebenszyklus. All dies muß verwaltet werden. Hierzu werfen wir einen Blick in die Spezifikation.

Aus der Signatur der `init()`-Methode folgt, daß jedes Servlet über ein `ServletConfig`-Objekt verfügt. Jedes `ServletConfig`-Objekt wiederum besitzt ein `ServletContext`-Objekt. Über diese expliziten Eigenschaften hinaus hat jedes Servlet einen Ausführungszustand. Es kann von vielen Threads gleichzeitig ausgeführt werden. Dies ist wichtig, da laut Spezifikation ein Servlet nur freigegeben werden darf, wenn es von keinem Thread ausgeführt wird. Zu dieser Regel gibt es nur eine Ausnahme: Nach einer durch Konfiguration definierten Wartezeit darf die Servlet-Engine ein Servlet zerstören, obwohl es noch ausgeführt wird.

Daher erscheint es uns sinnvoll, jedes Servlet in einem Objekt zu kapseln, das den Lebenszyklus des Servlets verwaltet. Dieses Objekt soll in der Lage sein, ein Servlet zu instantiieren, zu initialisieren, auszuführen und korrekt zu zerstören. In der WebApp-Servlet-Engine jo! ist dieses Objekt durch die Schnittstelle `I_JoServletModel` (Abbildung 7.3) definiert.

### 7.2.2 Servletkontext-Peer

Ein weiteres Problem ergibt sich im Zusammenhang mit dem Servletkontext. Zwar sind Servlets jeweils einem Kontext zugeordnet, laut Spezifikation soll es jedoch keine Möglichkeit geben, von einem Kontext ein Servlet zu erhalten. Die mittlerweile veraltete (deprecated) Methode `getServlet(String)` soll seit API Version 2.1 immer `null` zurückgeben. Explizit steht in der Spezifikation, daß Servlets keinen Zugang zu anderen Servlets haben sollen, da es keine Möglichkeit gibt, nachzuprüfen, in welchem Zustand sich das Servlet befindet. Diese Vorschrift widerspricht unserem Grunddesign, in dem eindeutig Servlets vom Kontext erlangt werden sollen.

Einerseits benötigen wir also Zugriff auf die Servlets, um sie auszuführen, andererseits müssen wir sie vor anderen Servlets verstecken. Zur Lösung dieses Dilemmas führen wir eine neue Klasse ein, den Servletkontext-Peer. Seine Aufgabe besteht darin, die Servletmodels zu verwalten und den Servletkontext zur Verfügung zu stellen. So erhalten wir also zwei Sichten auf den Kontext. Eine für die Engine, nämlich über den Peer, und eine für Servlets über das `ServletConfig`-Objekt.

Da man über den Peer zwar den Kontext erlangen, jedoch über den Kontext nicht zum Peer vordringen kann, ist sichergestellt, daß kein Servlet Zugang zu einer anderen Servlet-Instanz hat (Abbildung 7.3).

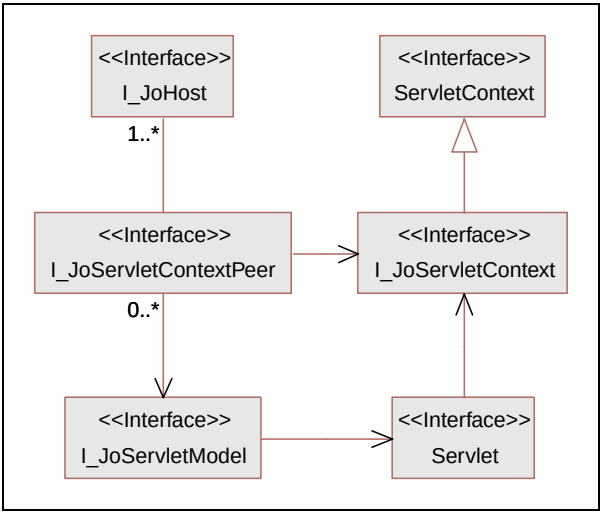


*Abbildung 7.3: Beziehungen zwischen `I_JoServletContextPeer` und `I_JoServletContext` sowie `I_JoServletModel` und `Servlet`*

### 7.2.3 Zusammenspiel

Aus unseren Überlegungen heraus ergibt sich ein präziseres Bild der Abläufe innerhalb der Engine, als wir dies vorher hatten. Wie bereits beschrieben, wird eine Anfrage vom Service angenommen und an einen Handler weitergeleitet. Der Handler fragt den Service nach einem passenden Host. Der Host wiederum wird nach einem Servletkontext-Peer gefragt. Vom Peer wird ein Servletmodel erlangt und dessen gekapseltes Servlet ausgeführt. Der Ablauf ist in Abbildung 7.4 dargestellt.

Im folgenden werden wir die Aufgaben der Objekte genauer betrachten.

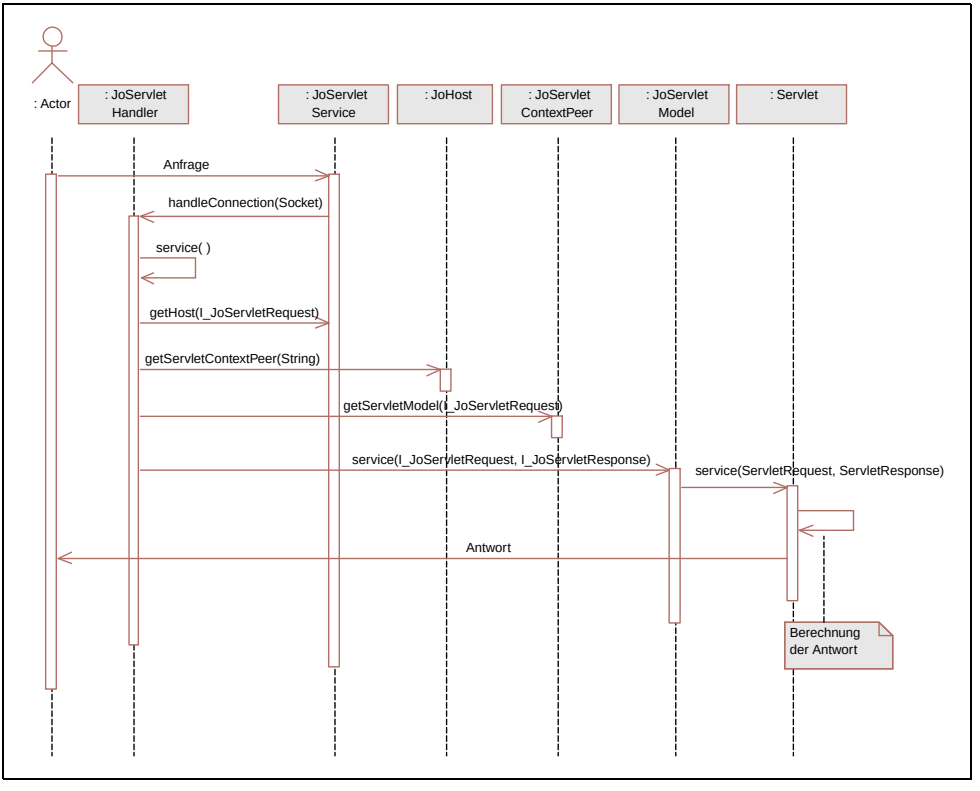


*Abbildung 7.4: Zusammenspiel der Objekte beim Beantworten einer Anfrage*

## 7.3 Service

Über seine bereits durch das Server-Paket definierte Rolle hinaus dient das Service-Objekt zum Bereitstellen serverweiter, anfrageunabhängiger Dienste sowie der Vermittlung von Hosts. Unsere Vorüberlegungen und eine Analyse der Schnittstellen des Servlet-API führen uns zu folgender Liste von zu verwaltenden Objekten:

- Server-Socket
- MIME-Typen
- Hosts

Der Server-Socket dient der Annahme einer Verbindung und wird bereits durch den `I_TCPService`, von dem unser Service `I_JoServletService` erbt, definiert. Diesbezügliches Verhalten muß also nicht mehr realisiert werden.

Mit sehr wenig Aufwand können die MIME-Typen verwaltet werden. Der Service muß lediglich in der Lage sein, eine Dateiendung einem MIME-Typ zuzuordnen. Zwar

nicht ganz so simpel, aber durchaus noch zu bewältigen, ist das Abbilden einer Anfrage auf einen Host. Hierfür muß eine Methode `getHost(ServletRequest)` zur Verfügung gestellt werden, die als Argument eine Anfrage entgegennimmt, analysiert, für welchen Host diese bestimmt ist und das richtige Host-Objekt zurückgibt. Zudem muß der Service über Methoden verfügen, die es ermöglichen, einen Host hinzuzufügen sowie ihn wieder zu entfernen.

Aus den gesammelten Anforderungen ergibt sich die recht übersichtliche Schnittstelle `I_JoServletService` (Listing 7.1), eine Erweiterung von `I_TCPService`:

```
package de.webapp.Framework.ServletEngine;

import javax.servlet.ServletException;
import java.util.Hashtable;
import de.webapp.Framework.Server.I_TCPService;

public interface I_JoServletService extends I_TCPService {
/** Gibt einen Host für eine Anfrage zurück */
public I_JoHost getHost(I_JoServletRequest aRequest);
/** Fügt dem Service einen Host hinzu */
public void addHost(I_JoHost aHost);
/** Entfernt einen Host aus diesem Service */
public void removeHost(I_JoHost aHost);
/** Fügt einen MIME-Typ hinzu */
public void addMimeType(String aSuffix, String aMimeType);
/** Entfernt einen MIME-Typ */
public void removeMimeType(String aSuffix);
/** Gibt Hashtabelle aller registrierten MIME-Typen zurück */
public Hashtable getMimeTypes();
/** Gibt den MIME-Typ eines Dateinamens zurück */
public String getMimeType(String aFile);
} // Ende der Schnittstelle
```

*Listing 7.1: Die Schnittstelle `I_JoServletService`*

## 7.4 Host

Zu den Hauptaufgaben des Hosts zählt insbesondere das Verwalten und Vermitteln von Servletkontext-Peers für eine gegebene Anfrage. Der Host soll außerdem das Dokumentenbasisverzeichnis (Document Root) sowie Informationen über den Server zur Verfügung stellen. Dazu gehört zum Beispiel der Servername. Weiterhin soll er einen Bezeichner (Name), einen Hostnamen sowie Synonyme für seinen Hostnamen (Aliases) kennen. Methoden für all diese Funktionen sind in der Schnittstelle `I_JoHost` (Listing 7.2) definiert.

Darüber hinaus vermittelt der Host noch eine weitere Ressource, die wir bisher noch nicht erwähnt haben: den Sessionkontext.

Der Sessionkontext ist für die Verwaltung sämtlicher Sessions eines Hosts zuständig. Bis zur Version 2.1 des Servlet-APIs war eine entsprechende Schnittstelle auch im API vorgesehen, wurde dann aber mißbilligt, da sie Servlets den Zugriff auf fremde `HttpSession`-Objekte gestattete. Dies ist aus Sicherheitsgründen abzulehnen. Dennoch müssen die Session-Objekte verwaltet werden. Dies geschieht in jo! durch den Sessionkontext eines Hosts. Beide Objekte sind in jo! vom Servlet aus nicht zugänglich, daher besteht auch kein Sicherheitsproblem.

Davon abgesehen bietet der Sessionkontext ein großes Erweiterungspotential. So gibt es beispielsweise Servlet-Engines (JRun, Java Webserver), die längere Zeit nicht benutzte Sessions serialisieren und auf die Festplatte auslagern, um Arbeitsspeicherplatz zu sparen. Für einen solchen Mechanismus, wäre eine intelligentere Sessionverwaltung nötig, als jo! sie realisiert. Wegen der Abstraktion durch Schnittstellen, ist eine solche Erweiterung jedoch innerhalb kürzester Zeit installierbar.

Genau wie die meisten anderen Schnittstellen des Pakets `de.webapp.Framework.ServletEngine` verfügt auch `I_JoHost` über die beiden Lebenszyklus-Methoden `init()` und `destroy()`. Da sich in Schnittstellen keine Konstruktoren definieren lassen, benutzen wir statt dessen die `init()`-Methode, um die Initialisierung des Objekts zu definieren. Hier wird zumeist auch eine Referenz auf das Service-Objekt übergeben. `destroy()` sollte aufgerufen werden, um aktiv Ressourcen freizugeben. Dies sollte geschehen, bevor ein Objekt der automatischen Speicherbereinigung überlassen wird.

```
package de.webapp.Framework.ServletEngine;

import de.webapp.Framework.ConfigManager.Configuration;
import de.webapp.Framework.Server.ServerException;
import java.util.*;

public interface I_JoHost {
/** Initialisiert diesen Host */
public void init(String aName, String aHostname, Vector aAliases, String aDocRoot,
I_JoServletService aService) throws ServerException;
/** Gibt den Namen dieses Hosts zurück */
public String getName();
/** Gibt den Hostnamen dieses Hosts zurück */
public String getHostname();
/** Gibt eine Aufzählung der Aliase für den Hostnamen zurück */
public Enumeration getAliases();
/** Gibt Name und Version des Servers an */
public String getServerInfo();
/** Setzt das Dokumentenbasisverzeichnis */
public void setDocumentRoot(String aDocRoot);
/** Gibt das Dokumentenbasisverzeichnis zurück */
public String getDocumentRoot();
/** Gibt den passenden ContextPeer für einen URI-Pfad zurück */
public I_JoServletContextPeer getServletContextPeer(String aURI);
```

```
/** Fügt dem Host einen ContextPeer hinzu */
public void addServletContextPeer(I_JoServletContextPeer aPeer);
/** Entfernt einen ContextPeer aus dem Host */
public void removeServletContextPeer(String aName);
/** Gibt den SessionContext zurück */
public I_JoSessionContext getSessionContext();
/** Gibt alle Ressourcen frei */
public void destroy();
} // Ende der Schnittstelle
```

*Listing 7.2: Das Schnittstelle `I_JoHost`*

## 7.5 Servletkontext-Peer

Der Servletkontext-Peer verwaltet die Servletmodels. Daher muß es Methoden geben, um dem Peer Models hinzuzufügen, sie zu entfernen und zu erlangen. Zudem muß man einen `ServletContext` über den Peer durch eine Methode `getServletContext()` erlangen können. Eine weitere Eigenschaft des Kontextes ist seine Zuordnung zu einem URI-Pfad. Das ist nötig, um Anfragen einem Kontext zuordnen zu können. Dementsprechend muß der Servletkontext-Peer über eine Methode `getURIPath()` verfügen. Die komplette Schnittstelle `I_JoServletContextPeer` ist in Listing 7.3 definiert.

Der Kontext-Peer verfügt über Lebenszyklus-Methoden mit gleicher Semantik wie der Host. Der einzige Unterschied ist, daß die `init()`-Methode eine `FactoryException` auslösen kann. Dies passiert dann, wenn es nicht gelingt, ein `ServletContext`-Objekt über die `JoFactory` zu instantiieren (mehr dazu in Abschnitt 7.8).

```
package de.webapp.Framework.ServletEngine;

import de.webapp.Framework.Utilities.FactoryException;

public interface I_JoServletContextPeer {
/** Initialisiert diesen ContextPeer */
public void init(String aName, String aURIPath, String aDocRoot, I_JoHost aHost,
I_JoServletService aService) throws FactoryException;
/** Fügt dem Peer ein ServletModel hinzu */
public void addServletModel(I_JoServletModel aModel);
/** Entfernt ein ServletModel aus dem Peer */
public void removeServletModel(String aName);
/** Gibt das passende ServletModel für eine Anfrage zurück */
public I_JoServletModel getServletModel(I_JoServletRequest aRequest);
/** Gibt den Namen dieses ContextPeers zurück */
public String getName();
/** Gibt den URIPfad dieses Kontextes zurück */
public String getURIPath();
/** Gibt den zugehörigen ServletContext zurück */
public I_JoServletContext getServletContext();
```

```
/** Gibt sämtliche Ressourcen frei */
public void destroy();
} // Ende der Schnittstelle
```

*Listing 7.3: Die Schnittstelle `I_JoServletContextPeer`*

## 7.6 Servletmodel

Wie bereits beschrieben, soll ein Servletmodel je ein Servlet kapseln. Daher muß es über alle relevanten, das Servlet betreffende Daten verfügen. Mit diesen Daten sollte es dem Servletmodel möglich sein, das Servlet zu instantiieren, zu initialisieren, auszuführen und freizugeben. Tabelle 7.1 zeigt, welche Parameter ein Servletmodel zur korrekten Verwaltung benötigt.

| Kategorie | Benötigte Parameter |
|---|---|
| Initialisierungsparameter | Name, Aliases, Klassenname, Initialisierungsparameter für ein `ServletConfig`-Objekt, `ServletContext`-Objekt, Freigabezeitschranke (Timeout), maximale Anzahl der Instanzen eines `SingleThreadModel`-Servlets |
| Ausführungsparameter | `ServletRequest`- und `ServletResponse`-Objekt |
| Interne Parameter | Zustand |

*Tabelle 7.1: Vom Servletmodel benötigte Parameter*

Die Initialisierungsparameter können bereits zur Erzeugungszeit des Servletmodels gesetzt werden, die Ausführungsparameter sind Parameter eines `service()`-Aufrufes und interne Parameter werden – wie die Bezeichnung schon sagt – intern verwaltet.

Unsere Überlegungen haben wir in der Schnittstelle `I_JoServletModel` zusammengefaßt (Listing 7.4):

```
package de.webapp.Framework.ServletEngine;

import javax.servlet.*;

import java.util.*;
import java.io.*;
import de.webapp.Framework.Utilities.FactoryException;

public interface I_JoServletModel {
/** Initialisiert dieses Model */
public void init(String aClassname,
      Hashtable aInitParameters,
      int aDestroyTimeout,
      int aMaxActiveInstances,
      String aName,
      Vector aliases,
```

```
      I_JoServletContext aContext,
      I_JoServletService aService)
         throws   FactoryException;
/** Gibt den Namen des Models zurück */
public String getName();
/** Gibt eine Aufzählung der Aliase dieses Models zurück */
public Enumeration getAliases();
/** Setzt die maximale Anzahl aktiver Instanzen */
public void setMaxActiveInstances(int max);
/** Gibt die maximale Anzahl aktiver Instanzen zurück */
public int getMaxActiveInstances();
/** Setzt die Zeit, nach der das Servlet zwangsweise freigegeben werden darf */
public void setDestroyTimeout(int aTimeout);
/** Gibt die Zeit zurück, nach der das Servlet zwangsweise freigegeben werden darf */
public int getDestroyTimeout();
/** Instantiiert das Servlet, falls dies noch nicht geschehen ist */
public void preload() throws FactoryException, ServletException;
/** Führt eine Anfrage aus */
public void service(I_JoServletRequest aRequest, I_JoServletResponse aResponse)
      throws IOException, ServletException;
/** Gibt das Servlet frei */
public void unload();
/** Zeigt an, ob das Servlet einsatzbereit ist */
public boolean isLoaded();
/** Gibt den Klassennamen des Servlets zurück */
public String getClassname();
} // Ende der Schnittstelle
```

*Listing 7.4: Die Schnittstelle `I_JoServletModel`*

Genau wie in der Schnittstelle `I_JoServletContextPeer` kann auch hier die `init()`-Methode eine `FactoryException` auslösen. Dies passiert, wenn es nicht gelingt, ein `ServletConfig`-Objekt über die `JoFactory` zu instantiieren (mehr dazu in Abschnitt 7.8). Das `ServletConfig`-Objekt wird genau wie das `ServletContext`-Objekt bereits bei der Initialisierung des Models gesetzt. Dabei muß das `ServletContext`-Objekt nicht selbst instantiiert werden, sondern wird als Parameter übergeben, da es sich um eine Ressource handelt, die sich mehrere Servletmodels teilen.

Neben der `init()`-Methode ist die `service()`-Methode elementar. Sie soll so realisiert sein, daß das gekapselte Servlet instantiiert und initialisiert wird, sofern das noch nicht geschehen ist. Verfügt das Model über eine einsatzbereite Servlet-Instanz, so führt es deren `service()`-Methode aus.

Wenn der Lebenszyklus des gekapselten Servlets beendet werden soll, kann die Methode `unload()` aufgerufen werden. Sie soll die `destroy()`-Methode aller von diesem Model referenzierten Servlet-Instanzen aufrufen. Dabei sollte vor der Freigabe eines gerade ausgeführten Servlets, die mit der Freigabezeitschranke (Timeout) angegebene Zeit abgewartet werden. Die Freigabezeitschranke läßt sich über die entsprechenden Methoden setzen und lesen.

Um etwas mehr Kontrolle über den Zustand des Models zu erhalten, verfügt es außerdem über die Methoden `preload()` und `isLoaded()`. Die Methode `preload()` instantiiert und initialisiert ein Servlet, ohne es auszuführen; `isLoaded()` gibt an, ob das Model über mindestens eine einsatzbereite Instanz des Servlets verfügt.

Die Methode `getClassname()` gibt den Klassenamen des gekapselten Servlets zurück, `getAliases()` eine Aufzählung der Regeln, nach denen das Servlet auf einen URI abgebildet wird.

Bleiben noch die Methoden `setMaxActiveInstances(int)` und `getMaxActiveInstances()` zu erklären. Für den Sonderfall, daß ein Servlet die Schnittstelle `javax.servlet.SingleThreadModel` (Abschnitt 3.8.2) implementiert, muß das Servletmodel in der Lage sein, einen Pool von Servlet-Instanzen zu verwalten. Die Größe dieses Pools sollte begrenzt sein, um den Speicherverbrauch limitieren zu können. Die beiden Methoden dienen also dazu, das Pool-Verhalten zu steuern. Handelt es sich nicht um ein `SingleThreadModel`-Servlet, sind beide Methoden bedeutungslos.

## 7.7 Handler

Der Handler kommt zum Zuge, wenn vom Service bereits eine Verbindung aufgebaut worden ist und eine Anfrage abgearbeitet werden soll. In unserem Fall bezieht sich die Anfrage immer auf ein Servlet. Es gilt also, die Anfrage zu lesen und das entsprechende Servlet zu finden und auszuführen.

Bisher haben wir die Anfrage eher abstrakt betrachtet. Innerhalb des Servlet-APIs handelt es sich aber um ein konkretes Objekt, das durch die Schnittstellen `ServletRequest` und dessen Erweiterung `HttpServletRequest` definiert ist. Ebenso verhält es sich mit der Antwort des Servers. Sie wird durch die Objekte `ServletResponse` beziehungsweise `HttpServletResponse` abgebildet.

Die Aufgabe des Handlers besteht also darin, ein `ServletRequest`- und ein `ServletResponse`-Objekt zu erzeugen, sich mit Hilfe des Anfrage-Objekts ein Servletmodel beim Service zu besorgen und dessen `service()`-Methode mit `ServletRequest` und `ServletResponse` als Parameter aufzurufen (Abbildung 7.4 und Abbildung 7.5). Da dies ein recht einfacher Vorgang ist, der leicht in der `service()`-Methode eines `TCPHandlers` Platz findet, ist es nicht nötig, eine eigene Schnittstelle für unseren Handler zu definieren.

Interessanter sind die Anfrage- und Antwort-Objekte: Beide benötigen als Initialisierungsparameter den Socket der Verbindung. Dies ist nötig, um den Eingabe- beziehungsweise Ausgabestrom zu erlangen und andere Daten über die Verbindung wie Servername und Adresse sowie die Portnummer erfahren zu können. Zudem benötigen beide eine Referenz auf den Service. Dies ist insbesondere nötig, um das Session-Management korrekt abwickeln zu können, da dies vom Host verwaltet wird, der über

den Service zugänglich ist. Ebenso können über Service und Host Logdienste und ähnliches vermittelt werden. Dazu gehört auch der Servername, der im Kopf jeder Antwort gesetzt sein sollte.

Während auf den ersten Blick die logische Trennung von `ServletRequest` und `ServletResponse` sinnvoll erscheint, macht sie beim Session-Management Probleme. Tatsächlich ist von Sun zunächst das Anfrage/Antwort-Modell spezifiziert worden, dem erst später das Session-Management hinzugefügt wurde. Um dies zu verwirklichen, ist es nämlich unerläßlich, daß ein `HttpServletRequest` sein `HttpServletResponse` kennt und umgekehrt (Abbildung 7.5). Anders wäre es nicht möglich, über das Anfrage-Objekt eine Session zu erzeugen, da dazu in der Antwort ein Cookie gesetzt werden muß. Ebenso könnte vom Antwort-Objekt kein URL umgeschrieben (URL-Rewriting) werden, ohne dazu Session-Daten zu benutzen, die nur vom `HttpServletRequest` zur Verfügung gestellt werden können. Zur Realisierung bieten sich also zwei Möglichkeiten an: beide Schnittstellen in einer Klasse oder für jede Schnittstelle eine eigene Klasse.

Da beide Schnittstellen ohnehin recht umfangreich sind, haben wir uns dafür entschieden, zwei Klassen zu schreiben. Das ist nicht zwingend. Die Servlet-Engine JServ beispielsweise faßt in Version 1.0 beide Schnittstellen in der Klasse `JServConnection` zusammen. Diese ist mit über 1700 Zeilen Code jedoch sehr unübersichtlich.

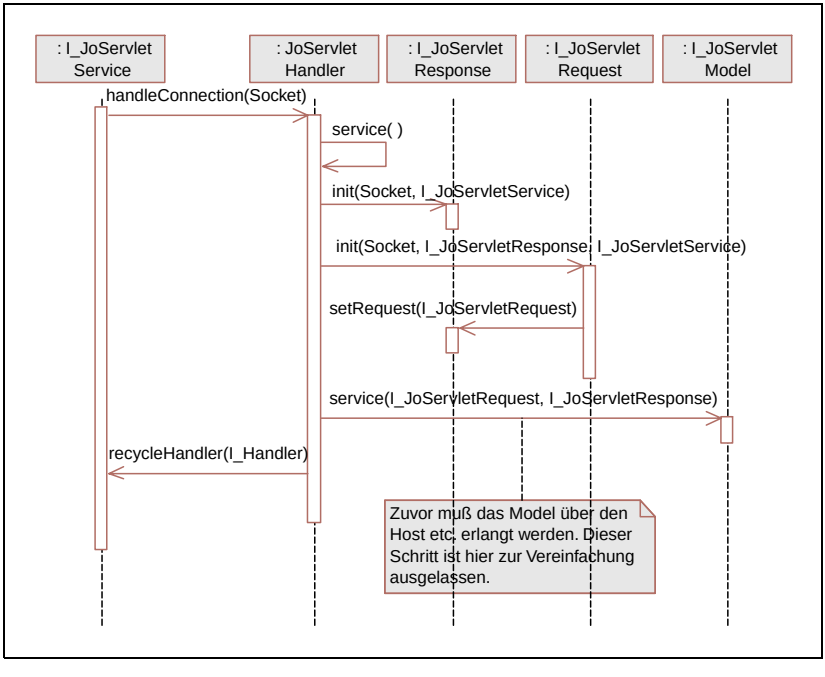


*Abbildung 7.5: Abläufe innerhalb der `handleConnection(Socket)`-Methode des `JoServletHandlers`.*

## 7.8 JoFactory

Nachdem wir uns über den Aufbau der wichtigsten Schnittstellen von jo! einen Überblick verschafft haben, wollen wir kurz auf ein anderes Design-Element eingehen. Wie bereits im einleitenden Abschnitt zum Framework-Teil dieses Buches erwähnt, haben wir bei der Entwicklung Wert darauf gelegt, daß möglichst viele Teile des Frameworks und damit auch von jo! durch Schnittstellen definiert sind. Natürlich existieren auch Realisierungen dieser Schnittstellen. Es ist aber nicht zwingend, diese auch zu benutzen. In jo! ist es besonders leicht, die realisierende Klasse einer Schnittstelle auszutauschen. Es genügt, die Konfigurationsdatei `/webapp/projects/jo/etc/factory.cfg` zu editieren.

```
//Datei "factory.cfg"
{
   I_JoServletConfig = "de.webapp.Framework.ServletEngine.JoServletConfig"
   I_JoServletModel = "de.webapp.Framework.ServletEngine.JoServletModel"
   I_JoServletContext = "de.webapp.Framework.ServletEngine.JoServletContext"
   I_JoServletContextPeer = "de.webapp.Framework.ServletEngine.JoServletContextPeer"
   I_JoServletRequest = "de.webapp.Framework.ServletEngine.JoServletRequest"
   I_JoServletResponse = "de.webapp.Framework.ServletEngine.JoServletResponse"
   I_JoServletService = "de.webapp.Framework.ServletEngine.JoServletService"
   I_JoSession = "de.webapp.Framework.ServletEngine.JoSession"
   I_JoSessionContext = "de.webapp.Framework.ServletEngine.JoSessionContext"
   I_TCPHandler = "de.webapp.Framework.ServletEngine.JoServletHandler"
   I_JoRequestWrapper = "de.webapp.Framework.ServletEngine.JoRequestWrapper"
   I_JoResponseWrapper = "de.webapp.Framework.ServletEngine.JoResponseWrapper"
   I_JoRequestDispatcher = "de.webapp.Framework.ServletEngine.JoRequestDispatcher"
   I_JoHost = "de.webapp.Framework.ServletEngine.JoHost"
}
```

*Listing 7.5: Die Datei `factory.cfg` definiert, welche Schnittstelle durch welche Klasse realisiert werden soll.*

Die Datei `factory.cfg` enthält eine Liste, die aus einem Alias und dem entsprechenden Klassennamen besteht. Fragt man die zugehörige Klasse `JoFactory` nach einem Alias, so erhält man eine Instanz der Klasse mit dem zugewiesenen Klassennamen zurück. Grundsätzlich werden innerhalb von jo! alle wichtigen Klassen auf diese Weise instantiiert. Eine Folge dieser Vorgehensweise ist, daß alle Klassen über einen argumentlosen Konstruktor verfügen müssen. Die Initialisierung erfolgt statt dessen mit einer `init()`-Methode, die bereits in der Schnittstelle festgelegt ist.

Der Gewinn ist eine größere Flexibilität. Ändert sich beispielsweise das Servlet-API derart, daß die neue Version nicht mit der alten kompatibel ist, kann man allein durch Editieren der Konfigurationsdatei die realisierenden Klassen austauschen. Auf diese Weise ließe sich beispielsweise ein komplett anderes Session-Management einbauen. Eine schnelle Migration beziehungsweise das Testen mit verschiedenen Versionen der Engine wird so stark erleichtert.

## 7.9 Was jo! ausmacht

Nachdem wir das Design von jo! besprochen haben, möchten wir uns noch einmal kurz klarmachen, welche Ziele erreicht wurden:

- jo! Basiert auf einem ServerKit. Damit haben wir eine klare Trennung von allgemeiner Server-Funktionalität (Thread-Management) und spezieller Server-Technologie (Webserver, Servlet-Engine) vollzogen. Auf diese Weise ist es einfach, beide Teile sinnvoll zu erweitern. Außerdem implementiert jo! die `I_Service`-Schnittstelle. Das bedeutet, daß jegliche Administrationdienste, die für das ServerKit entwickelt werden, auch jo! steuern können.
- jo! setzt das Servlet-API 2.1 um. Schon im Design ist Unterstützung für verschiedene `ServletContexte` vorgesehen. Es läßt sich über Konfigurationsdateien sogar eine andere Klasse für den `ServletContext` bestimmen. Zudem wird die Schnittstelle `RequestDispatcher` unterstützt.
- jo! trägt dem Bedürfnis nach virtuellen Hosts Rechnung und unterstützt diese explizit bereits in der Architektur.
- jo! ist vollständig über Schnittstellen definiert. Die Klassen, die diese Schnittstellen im Betrieb ausfüllen, lassen sich durch Ändern einer Konfigurationsdatei austauschen.

# 8 Servlet Method Invocation

An den Beispiel-Servlets am Ende von Kapitel 3 konnte man sehen, daß es schwierig ist, mehr als eine Funktion in einem Servlet unterzubringen. In den Beispielen haben wir die reine Anzeige von Seiten zumeist über die `GET`-Methode ausgeführt, während Aktionen wie das Versenden von Mails über die `POST`-Methode angestoßen wurden. Oft reicht die Unterscheidung in zwei Betriebsmodi jedoch nicht aus. Um eine Applikation mit Servlets zu bauen, müssen einzelne Servlets eine beliebige Anzahl an Funktionen bereitstellen oder zumindest ansprechen können. Servlets dienen oft als Mittler zwischen dem Nutzer und einer Applikation: Sie verbinden eine ansonsten unabhängige Anwendung mit einem Webserver.

Natürlich will man nicht Hunderte von Funktionen in einem einzigen Servlet verwirklichen. Vielmehr ist es sinnvoll, das Servlet als Zugang zu anderen Objekten zu nutzen, die jeweils eine überschaubare Anzahl an Funktionen zur Verfügung stellen. Solche Komponenten sollten sinnvoll mit einem Servlet assoziiert werden können. Auf diese Weise lassen sich je nach Anwendung Aggregate aus Standard- und Spezial-Komponenten bilden. Eine Standardkomponente könnte beispielsweise den Zugriff auf eine Datenbank oder das Mail-API kapseln, während eine Spezialkomponente servletspezifische Logik realisiert und eine andere Spezialkomponente den Zugriff auf servletunabhängige Business-Objekte organisiert.

Um den Grad der Wiederverwertbarkeit in verschiedenen Applikationen zu erhöhen, sollen die Komponenten der Beans-Spezifikation (`http://java.sun.com/beans/docs/spec.html`) entsprechen. Zudem halten wir es für opportun, die Kommunikation des Servlets mit den Komponenten über den Beans-Event-Mechanismus abzuwickeln. Zusammen genommen garantiert dieses Vorgehen eine optimale Entkoppelung unserer Komponenten – sowohl von der Servlet-Architektur als auch untereinander.

Um das gemeinsame Nutzen von Ressourcen (Datenbanken, externe Geräte etc.) zu ermöglichen, müssen Komponenten eines Servlets über eine gemeinsam zugängliche Datenstruktur Objekte teilen können. Genau wie in `HttpSession` (Abschnitt 3.9.5) definiert, müssen gebundene Objekte benachrichtigt werden können, wenn sie ge- beziehungsweise entbunden werden. Darüber hinaus soll jede Komponente eine Nachricht in Form eines Events an eine andere Komponente des gleichen Aggregats schicken können.

Damit auch mehrere Servlets beziehungsweise Komponenten-Aggregate Ressourcen teilen können, sollte zudem ein Kontext ähnlich dem `ServletContext` im Servlet-API 2.1 existieren. Wiederum müssen an den Kontext gebundene Objekte benachrichtigt werden, wenn sie ge- beziehungsweise entbunden werden. Diese Funktionalität fehlt leider im Servlet-API 2.1 und erfordert einen eigenen Freigabemechanismus.

Zusammengefaßt soll unser Framework zur Erweiterung des Servlet-APIs folgende Anforderungen erfüllen:

- Realisierung vieler Funktionen durch ein Servlet
- komponentenweise Kapselung zusammengehöriger Funktionen
- Wiederverwendbarkeit von Standard-Komponenten durch explizite Unterstützung von Aggregationen
- Ansteuerung von Funktionen über einen signalorientierten Kommunikationsmechanismus (Events)
- gemeinschaftliche Nutzung von Ressourcen auf unterschiedlichen Ebenen (Servlet, Komponente und Session)
- lebenszyklusorientierte Verwaltung geteilter Ressourcen

Das beschriebene Framework ist im Paket `de.webapp.Framework.SMI` enthalten. In Analogie zu RMI (Remote Method Invocation) steht SMI für Servlet Method Invocation.

## 8.1 Architektur

Wir wollen uns zunächst klarmachen, welche Abläufe es zu verwirklichen gilt. Eine Anfrage soll von einem Servlet angenommen und an eine bestimmte Komponente weitergeleitet werden. Diese Komponente soll die Anfrage abarbeiten und im Zuge dessen die Anfrage gegebenenfalls an eine andere Komponente weiterleiten.

Um sinnvoll arbeiten zu können, reicht es nicht, nur die Anfrage, also das `ServletRequest`-Objekt, weiterzuleiten. Erst zusammen mit dem `ServletResponse`- und `ServletContext`-Objekt ergibt sich eine vollständige Anfrage. Da wir die drei zusammengehörigen Objekte nicht einzeln übergeben wollen, kapseln wir sie in einem `SMIEvent` (Listing 8.1). Dieses `SMIEvent` soll von einem `SMIServlet`, das als Anfrage-Annahmestelle fungiert, instantiiert und an die richtige Komponente weitergeleitet werden. Das Event ist also unser zentraler Kommunikationsmechanismus. Um ihn flexibel zu halten, soll es möglichst leicht sein, das Servlet zu veranlassen, andere Event-Klassen zu instantiieren, die von `SMIEvent` erben. Dazu lassen wir es von einer `I_SMIEventFactory` produzieren, die mit dem Servlet assoziiert ist.

```
package de.webapp.Framework.SMI;

import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

import de.webapp.Framework.ConfigManager.*;

public class SMIEvent extends java.util.EventObject {
/** Values */
protected Hashtable myValues;
/** Das HttpServletRequest-Object  */
protected HttpServletRequest myRequest;
/** Das HttpServletResponse-Object  */
protected HttpServletResponse myResponse;
/** Das ServletContext-Object  */
protected ServletContext myServletContext = null;
/** Name des I_SMIEventListeners, der das Event ausführen soll */
protected String mySMIEventListenerName;

public SMIEvent(Object aSource, ServletContext aServletContext, HttpServletRequest
aRequest, HttpServletResponse aResponse, String aSMIEventListenerName) {
   super(aSource);
   mySMIEventListenerName = aSMIEventListenerName;
   myRequest = aRequest;
   myResponse = aResponse;
   myServletContext = aServletContext;
   myValues = new Hashtable();
   myValues.put("REQUEST", aRequest);
   myValues.put("RESPONSE", aResponse);
   myValues.put("SERVLETCONTEXT", aServletContext);
}
/** Gibt den Servlet-Kontext zurück. */
public ServletContext getServletContext() {
   return myServletContext;
}
/** Gibt das HttpServletRequest-Object zurück. */
public HttpServletRequest getRequest() {
   return myRequest;
}
/** Gibt das HttpServletResponse-Object zurück. */
public HttpServletResponse getResponse() {
   return myResponse;
}
/** Gibt den Namen des ausführenden Listeners zurück. */
public String getSMIEventListenerName() {
   return mySMIEventListenerName;
}
/** Gibt eine Aufzählung der Schlüssel zurück. */
public Enumeration getKeys() {
```

```
   return myValues.keys();
}
/** Gibt einen Parameter zurück. */
public Object getValue(Object aKey) {
   return myValues.get(aKey);
}
/** Setzt einen Wert. */
public void putValue(Object aKey, Object aValue) {
   myValues.put(aKey, aValue);
}
/** Löscht einen Wert. */
public void removeValue(Object aKey) {
   myValues.remove(aKey);
}
} // Ende der Klasse
```

*Listing 8.1: Die Klasse `SMIEvent`*

Neben der leichteren Handhabbarkeit hat die Kapselung noch weitere Vorteile: Da es einen Parameter geben muß, anhand dessen bestimmt werden kann, welche Komponente von diesem Event benachrichtigt werden soll, verfügt `SMIEvent` über die Methode `getSMIEventListenerName()`. Der Listenername ist quasi die Adresse, an die das Event ausgeliefert werden soll. Er bezeichnet einen `SMIEventListener`. Dabei handelt es sich um eine Komponente, welche die Schnittstelle `I_SMIEventListener` (Listing 8.2) realisiert und somit auf `SMIEvents` reagieren kann.

Die Tatsache, daß das `SMIEvent` den Namen seines Ziels kennt, entspricht nur bedingt der Beans-Spezifikation. Die Alternative wäre jedoch, für viele Ereignisse eigene Event-Klassen zu definieren, die jeweils nur an einen speziellen, registrierten Listener geliefert würden. Der Aufwand steht hier in keinem Verhältnis zum Nutzen, daher benutzen wir die universellen `SMIEvents`, die durch ihr Attribut Listenername spezialisiert werden.

Die indirekte Verknüpfung des Events mit seinem Listener über einen Namen (Alias) führt zudem zu einer starken Entkoppelung der einzelnen Komponenten und ermöglicht es, das Verhalten von SMI leicht über Konfigurationen zu beeinflussen. Dies wiederum erhöht den Grad der Wiederverwertbarkeit erheblich.

```
package de.webapp.Framework.SMI;

public interface I_SMIEventListener extends java.util.EventListener {

/** Initialisiert diesen Listener. */
public void init(String aName, I_SMIEventSwitch aSMIEventSwitch);
/** Führt ein Kommando aus. */
public void executeSMIEvent(SMIEvent aSMIEvent);
/** Gibt den Namen dieses Listeners zurück. */
public String getName();
/** Gibt den EventSwitch dieses Listeners zurück. */
```

```
public I_SMIEventSwitch getSMIEventSwitch();
/** Gibt eventuell belegt Ressourcen wieder frei. */
public void destroy();
} // Ende der Schnittstelle
```

*Listing 8.2: Die Schnittstelle `I_SMIEventListener`*

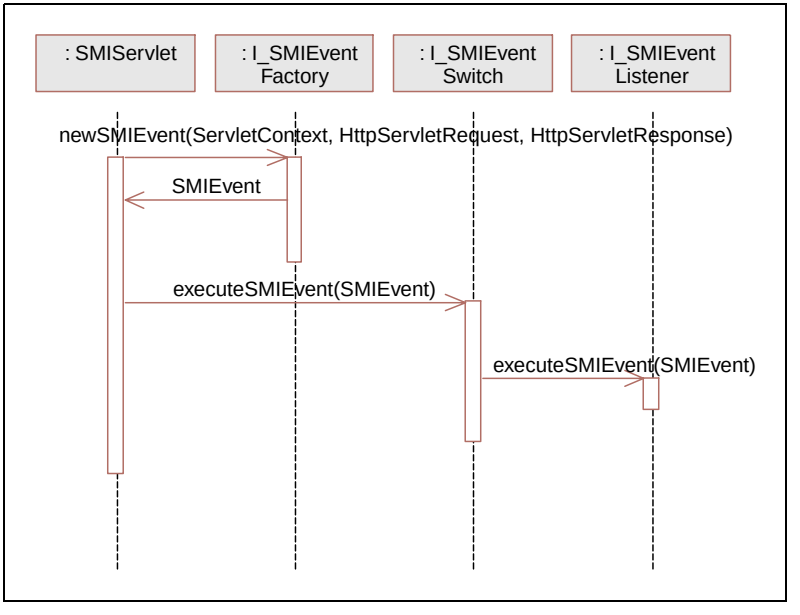


*Abbildung 8.1: Sequenzdiagramm von SMI*

Um ein `SMIEvent` an den richtigen `I_SMIEventListener` weiterzuleiten, verfügt das `SMIServlet` über einen `I_SMIEventSwitch`.

Darüber hinaus ist jeder `I_SMIEventSwitch` bei einem `I_SMIContext` registriert (Abbildung 8.2). Dadurch wird es ermöglicht, Objekte zwischen `I_SMIEventListenern` unterschiedlicher `I_SMIEventSwitches` zu teilen.

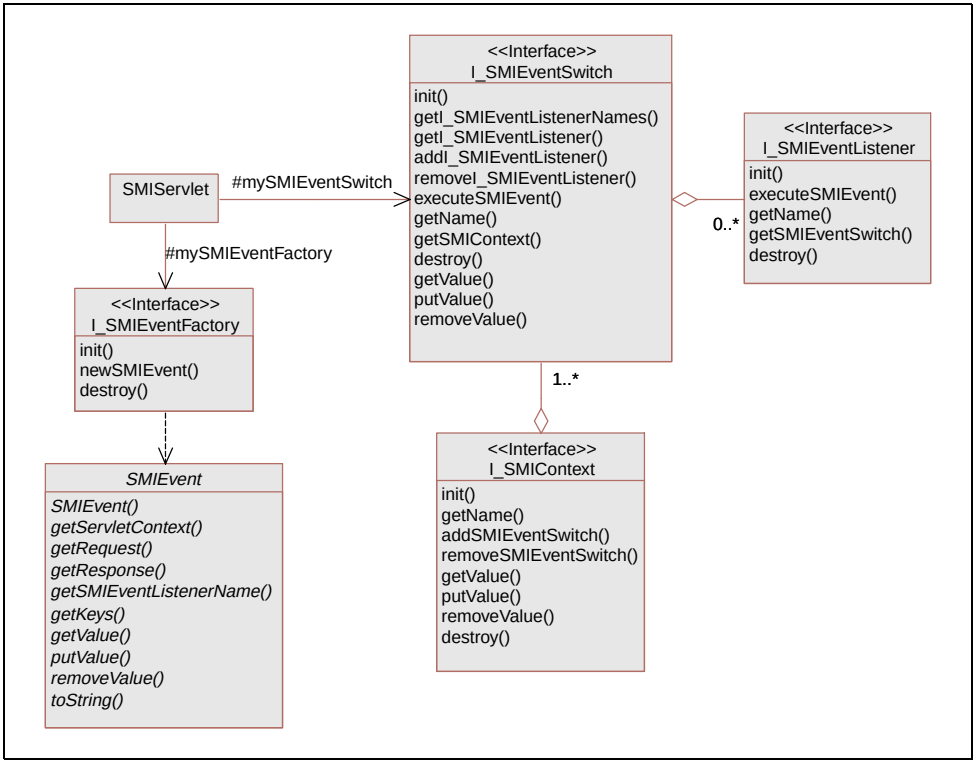


*Abbildung 8.2: Basisarchitektur von SMI*

## 8.2 Lebenszyklus

Mit der vorgestellten Architektur sind wir in der Lage, eine Anfrage zu kapseln und in Form eines `SMIEvents` über einen `I_SMIEventSwitch` an einen `I_SMIEventListener` weiterzuleiten. Dabei ist eine `I_SMIEventFactory` dafür verantwortlich, daß das Event weiß, zu welchem `I_SMIEventListener` es weitergeleitet werden möchte. Zu diesem Zweck ist es sinnvoll, daß die Fabrik über eine Konfiguration verfügt, die ihr zur Initialisierung mitgeteilt wird. Ebenso muß das Servlet wissen, wie sein `I_SMIEventSwitch` heißt, welche `I_SMIEventListener` bei diesem registriert werden und zu welchem `I_SMIContext` das resultierende Gebilde gehören soll.

`SMIServlet` übernimmt also neben seiner Rolle als Schnittstelle zur Servlet-Engine die Aufgabe, eine Instantiierungssituation herzustellen, die durch eine Konfigurationsdatei beschrieben ist. Da das Servlet über einen fremdgesteuerten Lebenszyklus verfügt (Abschnitt 3.4), ist es sinnvoll, daß das Servlet auch für das Abbauen der geschaffenen Instanzen verantwortlich ist. Dementsprechend verfügen alle beteiligten Schnittstellen über Initialisierungs- und Freigabemethoden mit den Bezeichnern `init()` und

`destroy()`. Da `javax.servlet.Servlet` bereits über entsprechende Methoden verfügt, befindet sich im SMI-Paket keine Schnittstelle für ein Servlet.

### 8.2.1 Initialisierung

Natürlich muß der Aufbau in einer geordneten Reihenfolge geschehen (Abbildung 8.3). Zunächst wird die `I_SMIEventFactory` instantiiert, anschließend der `I_SMIEventSwitch` und schließlich die `I_SMIEventListener`. Der `I_SMIEventSwitch` muß zudem noch bei seinem `I_SMIContext` registriert werden.

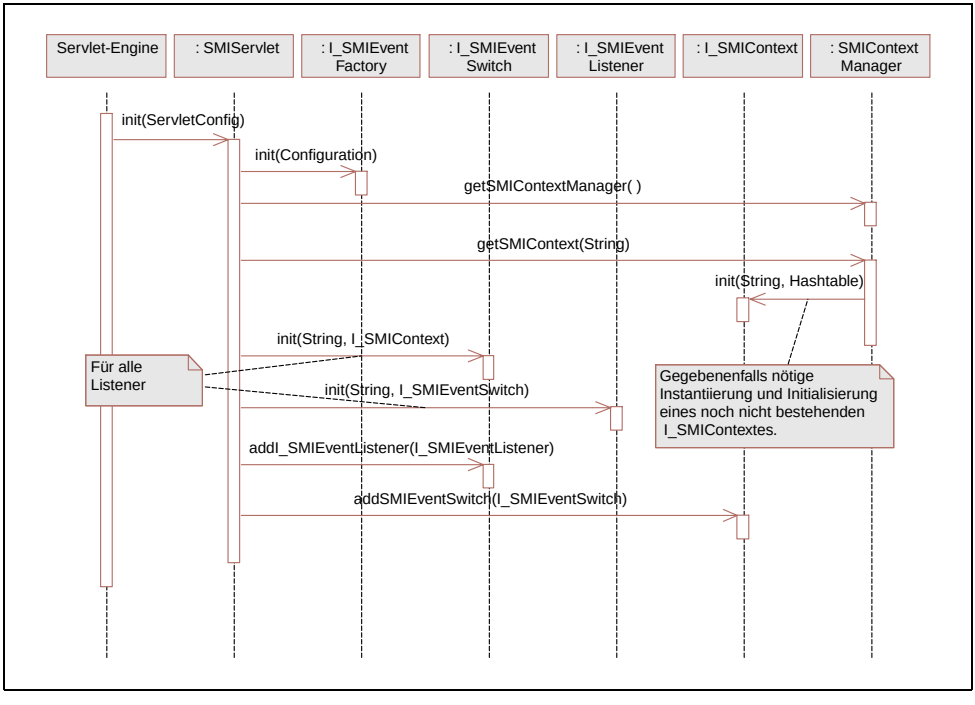


*Abbildung 8.3: Sequenzdiagramm der Initialisierung eines `SMIServlets`*

Da ein `I_SMIContext` von mehreren `I_SMIEventSwitches` beziehungsweise `SMIServlets` geteilt wird, kann er nicht einfach vom Servlet instantiiert werden. Vielmehr muß geprüft werden, ob der Kontext nicht bereits schon existiert. Nur, wenn er nicht existiert, muß er instantiiert werden. Idealerweise sollte die Verwaltung des Kontextes von der Engine übernommen werden. Da Servlet-Engines jedoch in der Regel nur schlecht erweiterbar sind, müssen wir einen anderen Weg finden.

In SMI übernimmt daher der Singleton `SMIContextManager` die Verwaltung der Kontexte. Wird ein Kontext von ihm angefordert, gibt er entweder einen bestehenden zurück oder instantiiert ihn nötigenfalls.

### 8.2.2 Freigabe

Auch die Freigabe von `I_SMIContexten` verläuft etwas anders als die der anderen beteiligten Objekte (Abbildung 8.4). Der Kontext kann nicht einfach freigegeben werden, wenn eines der registrierten Servlets freigegeben wird. Erst die Freigabe des letzten beim `I_SMIContext` registrierten `I_SMIEventSwitch` darf zur Freigabe des Kontextes führen. Das bedeutet, daß sich `I_SMIEventSwitches` bei einem Aufruf ihrer `destroy()`-Methode aus ihrem `I_SMIContext` entfernen müssen. Hat sich der letzte `I_SMIEventSwitch` entfernt, muß der `I_SMIContext` sich beim `SMIContextManager` abmelden und alle an ihn gebundenen Objekte freigeben.

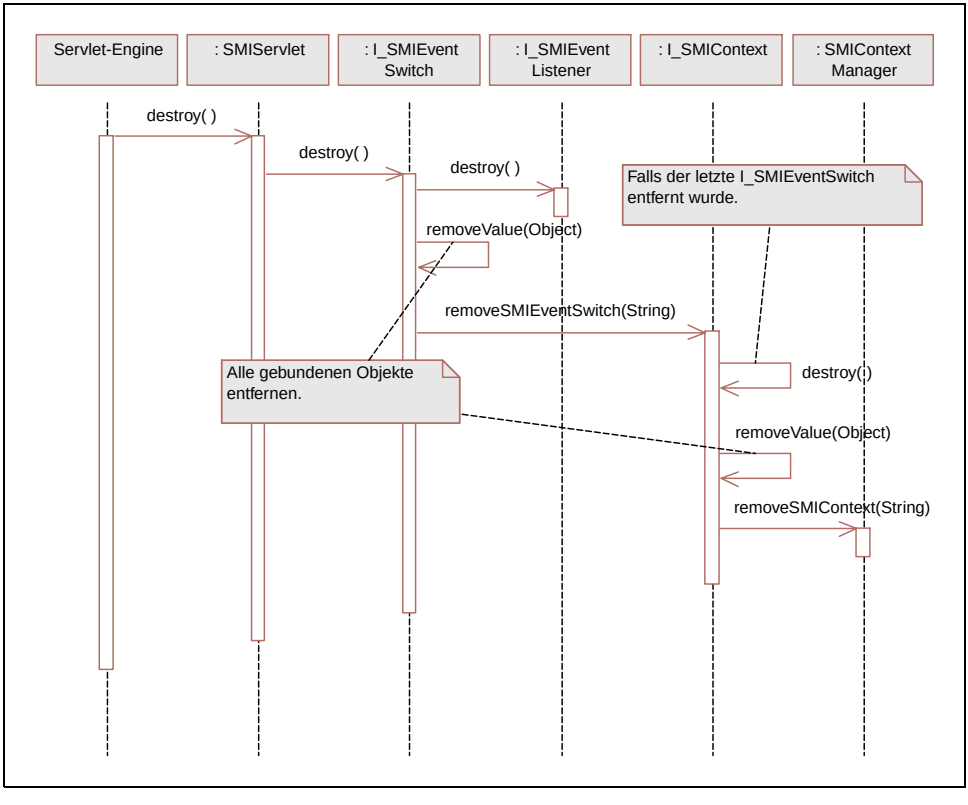


*Abbildung 8.4: Freigabe des* `SMIServlets`

## 8.3 Gebundene Objekte

Objekte, die die Schnittstellen `HttpSessionBindingListener`, `I_SMIContextBindingListener` oder `I_SMIEventSwitchBindingListener` realisieren, werden beim Binden und Entbinden an Session, Eventswitch oder Kontext durch ein Ereignis-Objekt benachrichtigt. Die möglichen Ereignisse heißen entsprechend `HttpSessionBindingEvent`, `SMIContextBinding-Event` und `SMIEventSwitchBindingEvent`. Die aufgerufenen Methoden besitzen immer die Signatur `valueBound(<x>BindingEvent)` und `valueUnbound(<x>BindingEvent)`:

```
package de.webapp.Framework.SMI;
public interface I_SMIEventSwitchBindingListener {
public void valueBound(SMIEventSwitchBindingEvent event);
public void valueUnbound(SMIEventSwitchBindingEvent event);
} // Ende der Schnittstelle
```

*Listing 8.3: Schnittstelle `I_SMIEventSwitchBindingListener`*

```
package de.webapp.Framework.SMI;
public interface I_SMIContextBindingListener {
public void valueBound(SMIContextBindingEvent event);
public void valueUnbound(SMIContextBindingEvent event);
} // Ende der Schnittstelle
```

*Listing 8.4: Schnittstelle `I_SMIContextBindingListener`*

In der Praxis sind die vorgestellten Schnittstellen und Ereignisse besonders praktisch, wenn eine Ressource automatisch auf ihre Freigabe reagieren soll. Zum Beispiel könnte sich eine Datenbankverbindung selbständig schließen, sobald sie aus dem `I_SMIContext` entbunden wird.

## 8.4 Realisierung

Das vorliegende Modell erlaubt recht unterschiedliche Realisierungen und damit auch Spezialisierungen. Letztendlich handelt es sich lediglich um eine Infrastruktur, die Nachrichten transportiert. Um diese sinnvoll einzusetzen, lohnt es sich, sie zu spezialisieren.

### 8.4.1 Kommandoorientierung

Die im Paket SMI vorliegende Realisierung beschreitet folgenden Weg: Eine Anfrage enthält immer Anfrage-Parameter, die über die Methode `ServletRequest.getParameter(String)` zugänglich sind. Ziel ist es, eine Anfrage mit einem speziellen Parameter an unser `SMIServlet` schicken zu können, welches dann veranlaßt, daß die korrespondierende Methode eines `I_SMIEventListeners` ausgeführt wird. Hierzu definieren wir, daß der Parameter mit dem Namen `Command` ein Alias für eine Methode eines `I_SMIEventListeners` ist (Abbildung 8.5).

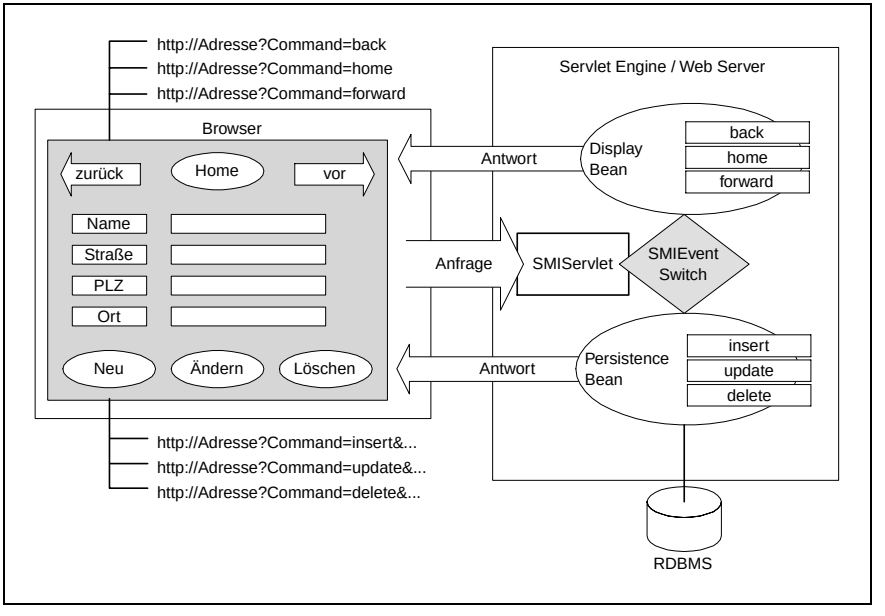


*Abbildung 8.5: Vereinfachter Ablauf eines Kommandoaufrufs:* `DisplayBean` *und* `PersistenceBean` *sind* `I_SMIEventListener`

Um komfortabel arbeiten zu können, bietet es sich an, das `SMIEvent` zu spezialisieren. Über eine Methode `getCommand()` soll bequem auf das Kommando zugegriffen werden können. Zusätzlich erweitern wir `SMIEvent` noch um Methoden, um einfacher auf das Session-Objekt der Anfrage zugreifen zu können. Wir nennen die resultierende Klasse `SMICommand` (Abbildung 8.6 und Listing 8.5):

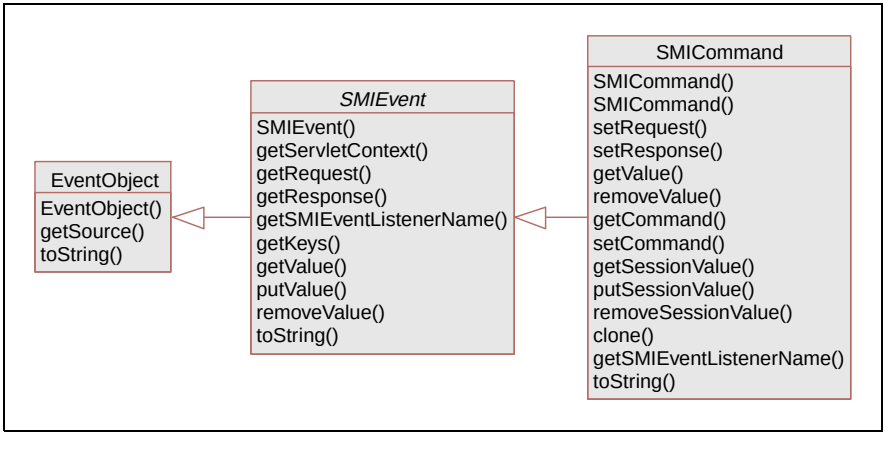


*Abbildung 8.6: Beziehungen zwischen den Klassen* `java.util.EventObject`, `SMIEvent` *und* `SMICommand`

```
package de.webapp.Framework.SMI;

import java.util.*;

import javax.servlet.*;
import javax.servlet.http.*;

import de.webapp.Framework.ConfigManager.*;

public class SMICommand extends SMIEvent implements C_SMI, Cloneable {

/** Das Kommando  */
protected String myCommand;
/** Defaultparameter  */
protected Hashtable myDefaultParameter;
/** Initparameter  */
protected Hashtable myInitParameter;
/** Zuordnung von Kommandos zu Listenern */
protected Hashtable myCommand2Listener;

public SMICommand (
      Object aSource,
      ServletContext aServletContext,
      HttpServletRequest req,
      HttpServletResponse res,
      String aCommand,
      Hashtable aValues,
      Hashtable aInitParameter,
      Hashtable aDefaultParameter,
      Hashtable aCommand2Listener
   ) {
   super(aSource, aServletContext, req, res, null);
   myCommand = aCommand;
   if (aInitParameter != null) myInitParameter = (Hashtable)aInitParameter.clone();
   if (aDefaultParameter != null) myDefaultParameter =
      (Hashtable)aDefaultParameter.clone();
   myValues = aValues;
   myCommand2Listener = aCommand2Listener;
   // Listenernamen überschreiben
   mySMIEventListenerName = getSMIEventListenerName(myCommand);
}

// Konstruktor zum Erstellen von Clones.
public SMICommand (SMICommand aSMICommand, Hashtable aValues, Hashtable
aInitParameter, Hashtable aDefaultParameter, Hashtable aCommand2Listener) {
   super(
      aSMICommand.getSource(),
      aSMICommand.getServletContext(),
      aSMICommand.getRequest(),
      aSMICommand.getResponse(),
      aSMICommand.getSMIEventListenerName()
```

```
   );
   myCommand = aSMICommand.getCommand();
   myInitParameter = aInitParameter;
   myDefaultParameter = aDefaultParameter;
   myValues = (Hashtable)aValues.clone();
   myCommand2Listener = aCommand2Listener;
}
/** Setzt das HttpServletRequest. */
public void setRequest(HttpServletRequest aRequest) {
   myRequest = aRequest;
   myValues.put("REQUEST", aRequest);
}
/** Setzt das HttpServletResponse. */
public void setResponse(HttpServletResponse aResponse) {
   myResponse = aResponse;
   myValues.put("RESPONSE", aResponse);
}
/** Gibt einen Parameter zurück. */
public Object getValue(Object aKey) {
   Object value = super.getValue(aKey);
   if (value == null) {
      if (myDefaultParameter != null) {
         value = myDefaultParameter.get(aKey);
      }
      if (value == null) {
         if (myInitParameter != null) {
            value = myInitParameter.get(aKey);
         }
      }
   }
   return value;
}
/** Löscht einen Wert. */
public void removeValue(Object aKey) {
   myValues.remove(aKey);
   if (aInitParameter != null) myInitParameter.remove(aKey);
   if (aDefaultParameter != null) myDefaultParameter.remove(aKey);
}
/** Gibt das Kommando zurück. */
public String getCommand() {
   return myCommand;
}
/** Setzt das Kommando. */
public String setCommand(String aCommand) {
   String oldCommand = myCommand;
   myCommand = aCommand;
   // Listenername setzen
   mySMIEventListenerName = getSMIEventListenerName(myCommand);
   return oldCommand;
}
/** Gibt den Wert eines an die Session gehängtes Objekt zurück. */
```

```
public Object getSessionValue(String aKey) {
   return getRequest().getSession(false).getValue(aKey);
}
/** Fügt der Session ein Objekt hinzu. */
public void putSessionValue(String aKey, Object aValue) {
   getRequest().getSession(false).putValue(aKey, aValue);
}
/** Löscht ein an die Session gehängtes Objekt. */
public void removeSessionValue(String aKey) {
   getRequest().getSession(false).removeValue(aKey);
}
/** Gibt eine flache Kopie des Objekts zurück. */
public Object clone() {
   return new SMICommand(this, myValues, myInitParameter, myDefaultParameter,
     myCommand2Listener);
}
/** Gibt den Listenernamen für ein Kommando zurück. */
protected String getSMIEventListenerName(String aCommand) {
   return (String)myCommand2Listener.get(aCommand);
}
} // Ende der Klasse
```

*Listing 8.5: Die Klasse `SMICommand`*

### 8.4.2 Konfiguration

Genau wie `SMIEvent` soll `SMICommand` von einer `I_SMIEventFactory` instantiiert werden. Die realisierende Klasse heißt in diesem Fall `SMICommandFactory`. Zur Instantiierung eines `SMICommands` interpretiert die Fabrik eine Konfiguration, die ihr vom `SMIServlet` übergeben wird (Listing 8.6). Die Konfiguration definiert, welchem Kommando welcher `SMIEventListener` zugeordnet wird, welche Methode des Listeners zu einem Kommando gehört, sowie über welche voreingestellten Werte das `SMICommand` verfügen soll.

Darüber hinaus wird in der Konfigurationsdatei definiert, welche `I_SMIEventListener` überhaupt zu diesem Servlet gehören. Dies geschieht im ersten Teil. Hinter dem Schlüsselwort `Listener` können in einer Hashtabelle Namen von Listenern eingetragen werden, die wiederum in einer Hashtabelle durch das `Class`-Attribut näher definiert werden. Optional können mit den Parametern `EventFactoryClass` und `EventSwitchClass` noch die Klassennamen der `I_SMIEventFactory` und des `I_SMIEventSwitches` angegeben werden. Der erste Teil der Konfiguration definiert also in erster Linie die Instantiierungssituation der SMI-Schicht.

Im zweiten Teil werden kommandospezifische Standardwerte gesetzt. Dies ist zunächst einmal das Standard-Kommando `DefaultCommand`, durch das definiert wird, was passiert, wenn eine Anfrage keinen Parameter mit dem Namen `Command` enthält. Außerdem kann man über `Values` Standardwerte definieren, auf die später über die `getValue()`-Methode des `SMICommands` zugegriffen werden kann.

Im dritten Teil werden die Kommandos definiert. Ein Kommando muß über die Attribute `MethodName` und `Listener` verfügen. Optional kann man wiederum Standardwerte setzen, auf die ebenfalls über `getValue()` zugegriffen wird. Im dritten Teil gesetzte Standardwerte überdecken grundsätzlich im zweiten Teil gesetzte Werte. Im dritten Teil gesetzte Werte werden wiederum von den Parametern des `HttpServletRequests` überdeckt. Sie sind aus Gründen der Bequemlichkeit ebenfalls über `getValue()` zugänglich – mit dem Unterschied, daß sämtliche `String`-Arrays in Vektoren umgewandelt wurden.

```
{
   // Teil I: Beschreibung der Instantiierungssituation der SMI-Schicht
   Listener = {
      PersistenceBean = {
         Class = "de.webapp.Framework.SMICommandListener.PersistenceBean" ;
      },
      DisplayBean = {
         Class = "de.webapp.Framework.SMICommandListener.DisplayBean";
      }
   };
   // Optional:
   // EventFactoryClass = "de.webapp.Framework.SMI.SMICommandFactory";
   // EventSwitchClass = " de.webapp.Framework.SMI.SMIEventSwitch";
   // Teil II: Standardwerte für Kommandos
   DefaultCommand = "display";
   Values = {
      _FollowUpCommand = "display";
      _TemplateName = "/htdocs/index.html";
   }
   // Teil III: Definition der Kommandos
   Commands = {
      display = {
         MethodName = "display";
         Listener = "DisplayBean" ;
      }
      NewObject= {
         MethodName = "getStore";
         Listener = "PersistenceBean" ;
         Values = {
            _TemplateName = "/jsp/object.jsp"
         }
      }
   }
}
```

*Listing 8.6: Beispielhafte SMI-Datei*

Welche Konfigurationsdatei das Servlet interpretieren muß, wird ihm über die Initialisierungsargumente `EventSwitch` und `Context` mitgeteilt. Der Kontext muß einem Eintrag in der Datei `Registry.cfg` entsprechen. Durch diese Beziehung wird implizit das Verzeichnis der Konfigurationsdatei festgelegt. Der Name des `EventSwitches` sollte dem

Namen einer Datei mit dem Suffix `.smi` in diesem Verzeichnis entsprechen. Würden also die beiden Konfigurationsdateien aus Listing 8.7 und Listing 8.8 interpretiert, müßte die Datei `/projects/OnlineShop/OnlineShop.smi` gelesen werden.

```
...
# OnlineShop
servlet.OnlineShop.code=de.webapp.OnlineShop.OnlineShopServlet
servlet.OnlineShop.initArgs=EventSwitch=OnlineShop,Context=OnlineShop
...
```

*Listing 8.7: Ausschnitt aus der Datei `servlet.properties`*

```
{
   ...
   OnlineShop = {
      PATH = "/projects/OnlineShop";
   };
   ...
}
```

*Listing 8.8: Ausschnitt aus der Datei `Registry.cfg`*

Neben den SMI-Definitionsdateien existiert noch eine weitere Konfigurationsdatei für den `SMIContext`. Sie befindet sich im gleichen Verzeichnis wie die SMI-Dateien und heißt genau wie der Kontext mit dem angehängten Suffix `.con`. Aus `OnlineShop` würde also `OnlineShop.con`. In ihr können der Klassenname des `I_SMIContext` und Standardwerte definiert werden:

```
{
   Class = "de.webapp.Framework.SMI.SMIContext"
   Values = {
      einSchluessel = "einWert"
   }
}
```

*Listing 8.9: Beispielhafte Kontext-Definitionsdatei*

### 8.4.3 SMICommandListener

Ist das `SMICommand` einmal erzeugt, wird es über den oben beschriebenen Mechanismus an einen `I_SMIEventListener` (Listing 8.2) weitergeleitet, indem dessen `executeSMIEvent()`-Methode aufgerufen wird. Nun ist in der Schnittstelle `I_SMIEventListener` nichts definiert, das die Interpretation eines Kommandos erlaubt. Daher erweitern wir `I_SMIEventListener` zu `I_SMICommandListener`:

```
package de.webapp.Framework.SMI;

import java.lang.reflect.*;
import java.util.Enumeration;

public interface I_SMICommandListener extends I_SMIEventListener {
```

```
/** Fügt ein Alias für eine Methode hinzu. */
public void addMethodAlias(String aAlias, String aMethodName) throws
NoSuchMethodException;
/** Entfernt einen Alias für eine Methode. */
public void removeMethodAlias(String aAlias);
/** Gibt alle Aliases zurück. */
public Enumeration getAliases();
} // Ende der Schnittstelle
```

*Listing 8.10: Die Schnittstelle `I_SMICommandListener`.*

`I_SMICommandListener` stellt Methoden zur Verfügung, mit denen sich Aliase für Methodennamen verwalten lassen. Das Alias steht in diesem Fall für ein Kommando. Bemerkenswert ist, daß eine Methode durchaus verschiedenen Kommandos zugeordnet werden kann und so eine Entkoppelung des Mechanismus erreicht wird. Dies wäre bei einer Gleichheit von Kommando und Methodenname nicht möglich.

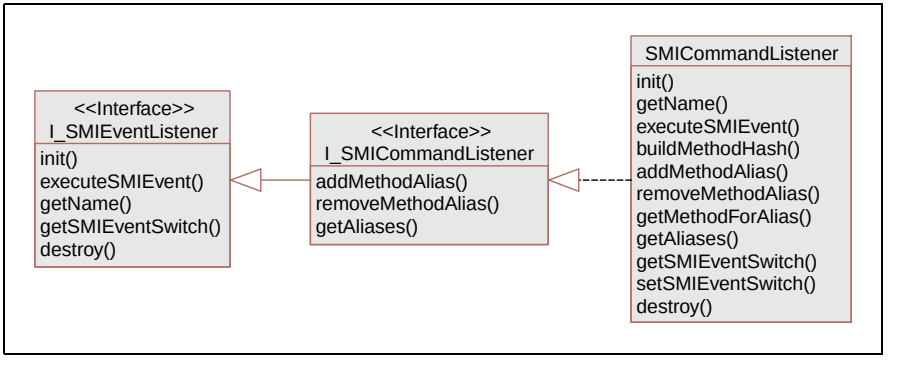


*Abbildung 8.7: Beziehung zwischen `I_SMIEventListener`, `I_SMICommandListener` und `SMICommandListener`*

Verwirklicht wird die Schnittstelle von der Klasse `SMICommandListener` (Listing 8.11 und Abbildung 8.7). Sie stellt einen Automatismus zum Aufruf der Methoden zur Verfügung. Dazu wird Javas Reflect-API (siehe JDK-Dokumentation zu dem Paket `java.lang.reflect`) benutzt.

```
package de.webapp.Framework.SMI;

import java.util.*;
import java.lang.reflect.*;

public class SMICommandListener implements I_SMICommandListener, java.io.Serializable
{
/** EventSwitch dieses SMICommandListener */
protected transient I_SMIEventSwitch mySMIEventSwitch;
/** Aliases */
```

```
protected transient Hashtable myAliases;
/** Methoden-Objekte */
protected transient Hashtable myMethods;
/** Name des Listeners */
protected String myName;
/** Parameter dieses CommandListeners */
protected Hashtable myParameters;

/** Initialisiert diesen Listener. */
public void init(String aName, Hashtable aParameters, I_SMIEventSwitch
aSMIEventSwitch) {
   myName = aName;
   mySMIEventSwitch = aSMIEventSwitch;
   myAliases = new Hashtable();
   myMethods = new Hashtable();
   myParameters = aParameters;
   buildMethodHash();
}
/** Gibt den Namen des Listeners zurück. */
public String getName() {
   return myName;
}
/** Führt ein Kommando per invoke aus. */
public void executeSMIEvent(SMIEvent aSMIEvent) {
   SMICommand aSMICommand = (SMICommand)aSMIEvent;
   Method theMethod = getMethodForAlias(aSMICommand.getCommand());
   if (theMethod != null) {
      Object[] args = {aSMICommand};
      try {
         theMethod.invoke(this, args);
      }
      catch (Throwable t) {
         if (t instanceof InvocationTargetException) {
            throw new SMIEventListenerException
               (((InvocationTargetException)t).getTargetException());
         }
         throw new SMIEventListenerException(t);
      }
   }
}
/** Speichert die Methoden-Objekte in myMethods. */
protected void buildMethodHash() {
   myMethods.clear();
   Class[] aSignature = { SMIEvent.class };
   Class[] types;
   Method[] methods = this.getClass().getMethods();
   for (int i=0; i<methods.length; i++) {
      types = methods[i].getParameterTypes();
      if (types.length == 1) {
         if (aSignature[0].isAssignableFrom(types[0])) {
            myMethods.put(methods[i].getName(), methods[i]);
```

```
            }
        }
    }
}
/** Fügt ein Alias für eine Methode hinzu. */
public void addMethodAlias(String aAlias, String aMethodName) throws
NoSuchMethodException {
    Method theMethod = null;
    theMethod = (Method)myMethods.get(aMethodName);
    if (theMethod == null) {
        throw new NoSuchMethodException("There is no method " + aMethodName
            + " in receiver " + this.getClass().getName());
    }
    myAliases.put(aAlias, theMethod);
}
/** Entfernt einen Alias für eine Methode. */
public void removeMethodAlias(String aAlias) {
    myAliases.remove(aAlias);
}
/** Gibt eine Methode für einen Alias zurück. */
public Method getMethodForAlias(String aAlias) {
    return (Method)myAliases.get(aAlias);
}
/** Gibt alle Aliases zurück. */
public Enumeration getAliases() {
    return myAliases.keys();
}
/** Gibt den EventSwitch dieses CommandListeners zurück. */
public I_SMIEventSwitch getSMIEventSwitch() {
    return mySMIEventSwitch;
}
/** Setzt den EventSwitch dieses CommandListeners. */
public void setSMIEventSwitch(I_SMIEventSwitch aSMIEventSwitch) {
    mySMIEventSwitch = aSMIEventSwitch;
}
/** Gibt alle benutzten Ressourcen wieder frei. */
public void destroy() { }
} // Ende der Klasse
```

*Listing 8.11: Die Klasse `SMICommandListener`*

Während der Initialisierung baut `SMICommandListener` über `buildMethodHash()` eine Hashtabelle aller seiner Methoden und ihrer Namen auf, die als Argument ein `SMIEvent` oder eine Unterklasse akzeptieren. Über `addMethodForAlias()` kann anschließend eine Zuordnung eines Aliases zu einem der Methoden-Objekte registriert werden. Dies geschieht während der Initialisierung durch das `SMIServlet`.

Der zentrale Trick zum Ausführen beliebiger Kommandos ist in der `executeSMIEvent()`-Methode kodiert: Bei ihrem Aufruf wird das `SMIEvent` in ein `SMICommand` gewandelt (Typecast) und über `getCommand()` nach seinem Kommando gefragt. Per `getMethodFor-`

Alias() wird für das Kommando ein Method-Objekt gefunden und mittels invoke() mit dem SMICommand als Argument ausgeführt. Schlägt invoke() fehl, wird das ausgelöste Throwable in einer SMIEventListenerException gekapselt. Für den Fall, daß es sich um eine InvocationTargetException handelt, wird lediglich die Zielausnahme gekapselt.

Solange SMICommandListener nicht erweitert wurde, ist dieses Vorgehen natürlich sinnlos – schließlich ist die einzige Methode, die SMIEvent als Argument akzeptiert, die Methode executeSMIEvent(). Hat eine Klasse jedoch von SMICommandListener geerbt, werden ihre Methoden automatisch registriert und dem Ausführen über ein SMICommand steht nichts mehr im Wege – zumindest, wenn ein Alias für die Methode registriert worden ist.

Der Einsatz von SMICommandListenern beziehungsweise der gesamten SMI-Schicht wird ausführlich im dritten Teil des Buches demonstriert. Daher steht an dieser Stelle nur ein kleines Beispiel.

## 8.5 DisplayBean

Als einfaches Beispiel für einen I_SMICommandListener wollen wir das DisplayBean (Listing 8.12) vorstellen. Es soll den vorgegebenen Parameter C_ConfigTemplateName als URI interpretieren und die Anfrage über einen RequestDispatcher weiterleiten. Um auf das SMIEvent auch in der angesprochenen Ressource zugreifen zu können, wird das SMIEvent als Attribut der Anfrage gesetzt.

```
package de.webapp.Framework.SMICommandListener;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.Hashtable;
import de.webapp.Framework.SMI.*;

public class DisplayBean extends SMICommandListener implements Serializable {

/** Konstante zum Auslesen des Templatenames aus dem I_SMIEvent. */
public static String C_ConfigTemplateName = "_TemplateName";
/**
  * Stellt eine Seite dar, die im Command-Attribut
  * _TemplateName spezifiziert ist.
  * In der Anfrage wird als Attribut das Kommando-Objekt
  * unter dem Schlüssel SMIEvent hinterlegt.
  */
public void display (SMIEvent aSMIEvent)
      throws IOException, ServletException {
   aSMIEvent.getRequest().setAttribute("SMIEvent", aSMIEvent);
   String theTemplateName = (String)aSMIEvent.getValue(C_ConfigTemplateName);
   RequestDispatcher theRequestDispatcher =
```

```
aSMIEvent.getServletContext().getRequestDispatcher(theTemplateName);
   theRequestDispatcher.forward(aSMIEvent.getRequest(), aSMIEvent.getResponse());
}
} // Ende der Klasse
```

*Listing 8.12: Die Klasse `DisplayBean` aus dem Paket `de.webapp.Framework.SMICommandListener`*

Die Konfiguration in Listing 8.13 bewirkt, daß das angesprochene Servlet den `I_SMIEventListener DisplayBean` instantiiert und ihm zwei Kommandos zuordnet: `display` und `welcome`. Wird das Servlet mit dem Command `display` aufgerufen, zeigt es die Seite `/index.html` an:.

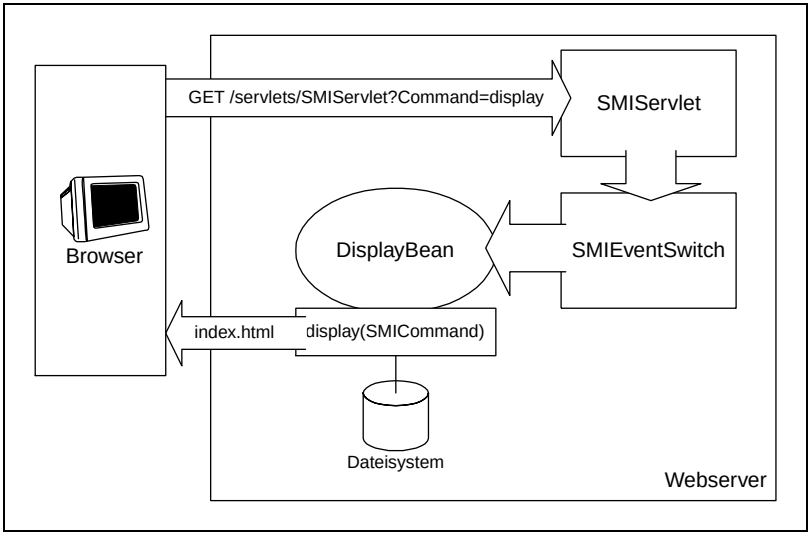


*Abbildung 8.8: Ausführung des Kommandos `display` mit dem `DisplayBean`*

Dies geschieht ebenfalls, wenn kein Kommando angegeben wurde, da `display` das `DefaultCommand` ist. Wird also das Kommando `welcome` angegeben, zeigt das `DisplayBean` die Seite `/welcome.html` an.

```
{
   // Teil I
   Listener = {
      DisplayBean = {
         Class = "de.webapp.Framework.SMICommandListener.DisplayBean";
      }
   };
   // Teil II
   DefaultCommand = "display";
   Values = {
      _TemplateName = "/index.html";
   }
```

```
   // Teil III
   Commands = {
      display = {
         MethodName = "display";
         Listener = "DisplayBean" ;
      }
      welcome = {
         MethodName = "display";
         Listener = "DisplayBean" ;
         Values = {
             _TemplateName = "/welcome.html"
         }
      }
   }
}
```

*Listing 8.13: Beispielhafte Konfiguration eines `SMIServlets` für den Einsatz des `DisplayBeans`*

## 8.6 SMI weitergedacht

Die Realisierung des SMI-Konzepts mittels des beschriebenen `SMICommandListeners` ist nur eine von vielen Möglichkeiten. Die Klasse `de.webapp.Framework.SMI.SMIBean`, eine Ableitung des `SMICommandListeners`, verwirklicht einen universellen Ansatz. Sie ermöglicht, beinahe beliebige Methoden per Konfiguration festgelegter Objekte aufzurufen. Um dies zu ermöglichen, müssen über den `Command`-Parameter hinaus noch einige Konventionen getroffen werden.

Der Name der aufzurufenden Methode ist weiterhin in der SMI-Definitionsdatei mit dem Parameter `MethodName` angegeben. Zusätzlich definiert werden muß lediglich die Art und Weise der Argumentübergabe. Wir legen fest, daß diese über die Parameter `_SMIArg<x> erfolgt`. Der Platzhalter `<x>` bezeichnet dabei die Nummer des Arguments beginnend mit `_SMIArg0`. Die Anzahl der direkt aufeinander folgenden Argumente bestimmt die Anzahl der Argumente für den Methodenaufruf.

Über die Parameter `_SMIArgType<x>` läßt sich zudem für jedes Argument ein Typ angeben. Zulässig sind alle Java-Klassen, die über einen Konstruktor verfügen, der als einziges Argument einen `String` erwartet. Dazu zählen unter anderem alle Basis-Objekte wie `java.lang.Integer`, `java.lang.Double`, `java.lang.Boolean` etc. Die Einschränkung ist nötig, da der Parameter zur Instantiierung der Klasse als Parameter der Anfrage übergeben wird. Als `_SMIArgType<x>` muß der vollständige Klassenname angegeben werden. Wird kein `_SMIArgType<x> angeben, interpretiert das SMIBean das Argument automatisch als String`.

Oft gibt es Situationen, in denen die direkt zugänglichen Argument-Typen nicht ausreichen und in denen beispielsweise ein Objekt der `HttpSession` als Argument dienen soll. Auch dies ist mit dem `SMIBean` möglich. Als `_SMIArgType<x>` muß man dazu ledig-

lich einen der reservierten Typnamen _SessionValue, _CommandValue, _SwitchValue, oder _ContextValue angeben. Der Wert von _SMIArg<x> dient dann als Schlüssel zur jeweiligen Datenstruktur. Über diese indirekte Adressierung hinaus kann man zusätzlich zu den Werten eines dieser Objekte auch die Objekte selbst übergeben. Dazu muß einer der Typen _SessionObject, _CommandObject, _SwitchObject oder _ContextObject angegeben werden. Auf gleiche Art ist es auch möglich, null als Argument zu setzen. Der entsprechende Argument-Typ heißt _Null. Generell gilt: Es wird immer die gemäß der Java-Sprachspezifikation spezifischste Methode [Gosling et al. 97, S. 313ff] für die angegebenen Argument-Typen aufgerufen.

Natürlich ist nicht nur der Aufruf einer Methode interessant, sondern auch ihr Rückgabewert. Dieser kann in einer der Datenstrukturen HttpSession, SMICommand, SMIEvent-Switch und SMIContext hinterlegt werden, indem man als _SMIResultScope einen der Werte Session, Command, Switch oder Context angibt. Das Ergebnis der aufgerufenen Methode wird dann unter dem Schlüssel _SMIResult hinterlegt. Ein anderer Schlüssel läßt sich mit dem Parameter _SMIResultIdentifier angeben.

Um den Rückgabewert interpretieren zu können, ist es zudem möglich, ein Folgekommando mittels des Parameters _FollowUpCommand zu setzen. Vor dem Ausführen des Folgekommandos wird der Parameter _FollowUpCommand aus dem Kommando-Objekt gelöscht.

| Parameter | Bedeutung |
|---|---|
| _SMIArg<#> | Wert des Arguments Nummer <#> |
| _SMIArgType<#> | Typ des Arguments Nummer <#>. Möglich sind zum Beispiel die Klassen java.lang.Integer, java.lang.Double etc. Zusätzlich sind die reservierten Typnamen _CommandValue, _SessionValue, _SwitchValue, _ContextValue, _CommandObject, _SessionObject, _SwitchObject, _ContextObject und _Null erlaubt. |
| _SMIResultScope | Gibt den Bereich an, in dem der Rückgabewert der aufgerufenen Methode hinterlegt werden soll. Mögliche Werte sind Session, Command, Switch und Context. |
| _SMIResultIdentifier | Gibt den Schlüssel an, unter dem der Rückgabewert hinterlegt werden soll. Wird dieser Parameter nicht angegeben, wird der Standard-Schlüssel _SMIResult benutzt. |
| _FollowUpCommand | Gibt das Kommando an, das nach der Ausführung des aktuellen Kommandos aufgerufen werden soll. |

*Tabelle 8.1: Mögliche Parameter des SMIBean*

Zum konkreten Einsatz des SMIBean muß noch definiert werden, wessen Methoden es aufrufen soll. Standardmäßig ruft SMIBean seine eigenen Methoden auf. Das macht natürlich nur Sinn, wenn SMIBean zuvor spezialisiert wurde. Um auch die Methoden

anderer Objekte aufrufen zu können, kann man in der SMI-Definitionsdatei durch den Parameter `TargetObject` die Klasse des Zielobjekts festlegen. Während der Initialisierung des SMIBeans wird automatisch eine Instanz der Klasse erzeugt und als Zielobjekt gesetzt. Als Voraussetzung für dieses Verfahren muß das Zielobjekt über einen argumentlosen Konstruktor verfügen. Listing 8.14 zeigt die Konfiguration eines SMI-Beans, das als Zielobjekt ein `DisplayBean` instantiiert:

```
{
   // Teil I
   Listener = {
      DisplayBean = {
         Class = " de.webapp.Framework.SMI.SMIBean";
         TargetObject = "de.webapp.Framework.SMICommandListener.DisplayBean";
      }
   };
   // Teil II
   DefaultCommand = "display";
   Values = {
      _SMIArgType0 = "_CommandObject"
      _TemplateName = "/index.html"
   }
   // Teil III
   Commands = {
      display = {
         MethodName = "display";
         Listener = "DisplayBean" ;
      }
      welcome = {
         MethodName = "display";
         Listener = "DisplayBean" ;
         Values = {
            _TemplateName = "/welcome.html"
         }
      }
   }
}
```

*Listing 8.14: Beispiel-Konfigurationsdatei für `SMIBean`*

Besonders interessant an der Realisierung von `SMIBean` sind ihre Erweiterungsmöglichkeiten. Die `SMIBean`-Ableitung `SMI2RMIBean` (Listing 8.15) interpretiert den Parameter `TargetObject` als URL eines RMI-Objekts und ist so in der Lage, dessen Methoden aufzurufen, sofern die Argument-Objekte die Schnittstelle `java.io.Serializable` implementieren. Eine ähnliche Komponente zur Anbindung an CORBA steht mit dem `SMI2CORBABean` ebenso zur Verfügung. Lediglich die Konfiguration ist etwas anders (Listing 8.16). Damit steht der generischen Anbindung eines Enterprise-JavaBeans-Servers (EJB-Server), eines Legacy-Systems oder einem anderen Hintergrundserver nichts mehr im Wege.

```
package de.webapp.Framework.SMI;

import java.rmi.*;
import java.util.*;

public class SMI2RMIBean extends SMIBean implements I_SMICommandListener, C_SMI,
java.io.Serializable  {

/** Initialisiert diesen Listener. Dabei wird das RMI-Objekt gebunden. */
public void init(String aName, Hashtable aParameters, I_SMIEventSwitch
aSMIEventSwitch) {
   myName = aName;
   mySMIEventSwitch = aSMIEventSwitch;
   myAliases = new Hashtable();
   myParameters = aParameters;
   String theTargetName = (String)myParameters.get(_ConfigListenerTargetObject);
   if (theTargetName == null) {
      setTargetObject(this);
   }
   else {
      try {
         setTargetObject(Naming.lookup(theTargetName));
      }
      catch(Throwable t) {
         throw new SMIEventListenerException(t);
      }
   }
}
} // Ende der Klasse
```

*Listing 8.15: Die Klasse* `SMI2RMIBean`

```
package de.webapp.Framework.SMI;

import java.util.*;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class SMI2CORBABean extends SMIBean implements I_SMICommandListener, C_SMI{
/** Helferklassen-Suffix */
public static final String _HelperSuffix="Helper";
/**
  * Initialisiert diesen Listener.
  * Dabei wird das CORBA-Objekt gebunden.
  * In der SMI-Konfigurationsdatei muß ein folgendem Muster
  * entsprechender Eintrag stehen:
  *    ...
  *    Listener = {
  *       meinCORBABean = {
  *          Class = "de.webapp.Framework.SMI.SMI2CORBABean" ;
  *          TargetObject = {
  *              Name = "meinCORBAObjekt" ;
```

```
  *             Class = "de.webapp.meinCORBAObjekt" ;
  *             ORBInitialPort = "1050" ;
  *             ORBInitialHost = "localhost" ;
  *       }
  *    }
  *    ...
  */
public void init(String aName, Hashtable aParameters, I_SMIEventSwitch
aSMIEventSwitch) {
   myName = aName;
   mySMIEventSwitch = aSMIEventSwitch;
   myAliases = new Hashtable();
   myParameters = aParameters;
   Hashtable theCORBAParameters =
      (Hashtable)myParameters.get(_ConfigListenerTargetObject);
   if (theCORBAParameters == null) {
      setTargetObject(this);
   }
   else {
      try {
         String theTargetName = (String) theCORBAParameters.get(_ConfigName);
         String theTargetClass = (String) theCORBAParameters.get(_ConfigClass);
         String theORBInitialHost = (String)theCORBAParameters.get(
           _ConfigORBInitialHost);
         String theORBInitialPort = (String) theCORBAParameters.get(
           _ConfigORBInitialPort);
         Properties props = new Properties();
         if(theORBInitialHost != null) {
            props.put("org.omg.CORBA.ORBInitialHost", theORBInitialHost);
         }
         if(theORBInitialPort != null) {
            props.put("org.omg.CORBA.ORBInitialPort", theORBInitialPort);
         }
         ORB theOrb = ORB.init(new String[0], props);
         NamingContext theNamingContext = NamingContextHelper.narrow(
            theOrb.resolve_initial_references("NameService"));
         NameComponent theNameComponent = new NameComponent(theTargetName, "");
         NameComponent path[] = {theNameComponent};
         Class theHelperClass = Class.forName(theTargetClass + _HelperSuffix);
         Class argTypes[] = {org.omg.CORBA.Object.class};
         java.lang.Object args[] = {theNamingContext.resolve(path)};
         java.lang.Object target = theHelperClass.getMethod("narrow",
         argTypes).invoke(null, args);
         setTargetObject(target);
      }
      catch(Throwable t) {
         throw new SMIEventListenerException(t);
      }
   }
}
} // Ende der Klasse
```

*Listing 8.16: Die Klasse* `SMI2CORBABean`

## 8.7 SMI zusammengefaßt

Nachdem wir das Design von SMI besprochen haben, möchten wir noch einmal die wesentlichen Eigenschaften aufzählen:

- SMI ist eine Erweiterung des Servlet-APIs.
- Es ermöglicht den modularen Aufbau von servletbasierten Applikationen.
- Die direkte Kommunikation zwischen Komponenten erfolgt über generische Events.
- Objekte können zusätzlich auf verschiedenen Ebenen (`SMIContext`, `SMIEventSwitch` und `SMIEvent`) in Datenstrukturen zur Verfügung gestellt werden.
- SMI fördert die Trennung von Anwendungslogik und Anzeige.
- Der Einsatz von Schnittstellen garantiert Flexibilität und Erweiterungsfähigkeit.

SMI ermöglicht den direkten Aufruf von Java-Methoden über HTTP.

# 9 Java-Objekte in einer relationalen Datenbank

Die Geschäftswelt drängt mit immer mehr Informationen ins Web. Mega- und Gigabytes umfassende Informationssysteme und Produktdatenbanken müssen schnell und überall in der Welt verfügbar sein. Daher ist eine Website für geschäftliche Zwecke ohne die Nutzung von Datenbanken mittlerweile undenkbar. Nötig sind möglichst schnelle und einfache Ansätze, die die Nutzung selbst komplexer Business-Prozesse über das Web ermöglichen. Dies ist eine neue Herausforderung für die Entwickler.

Um diesen Anforderungen gerecht werden zu können, ist es sinnvoll, die Daten auch weiterhin mit schnellen und robusten relationalen Datenbanken zu verwalten. Gleichzeitig drängt sich Java zur einfachen, sicheren und portablen Realisierung von Netzwerkanwendungen geradezu auf. Aus diesen Überlegungen heraus ergibt sich die Notwendigkeit, einen Weg zu finden, von Java aus möglichst komfortabel auf relationale Datenbanken zuzugreifen. Unweigerlich stößt man dabei auf JDBC, die Java Database Connectivity [Hamilton et al. 97]. Dieses API stellt eine gelungene Abstraktion für die diversen Datenbankprodukte dar. JDBC 2.0 unterstützt sogar den aktuellen Entwurf des SQL3 Standards (Structured Query Language). Soweit die Hersteller den notwendigen Treiber, als 100%-Pure-Java-Version zur Verfügung stellen, ist damit bisher unerreichte Portabilität garantiert.

JDBC ist jedoch nicht sehr komfortabel. Es fehlt vor allem eine echte objektorientierte Sicht auf die Daten. Als Programmierer muß man sich immer noch mit `ResultSets`, Spalten und Zeilen beschäftigen, obwohl man viel lieber mit Objekten, Methoden und Attributen arbeiten würde. Doch was tun?

Wir benötigen ein Framework, das es uns ermöglicht, Objekte in einer relationalen Datenbank zu speichern, diese Objekte wieder zu laden sowie gezielt nach bestimmten Objekten zu suchen.

Solche speicherbaren Objekte werden *persistente* Objekte genannt. Im Gegensatz zu *transienten* Objekten besitzen sie die Fähigkeit, die Lebensdauer eines Prozesses zu überdauern. Diese Objekte bezeichnen wir im WebApp-Framework als `Persistence`. Sie zeichnen sich durch folgende Eigenschaften aus: `Persistence` lassen sich in einer relationalen Datenbank speichern und jederzeit wieder laden. Sie besitzen eine Identität und

einen Zustand. Außerdem kann jedes `Persistence` mit anderen `Persistence` assoziiert sein. Der Zugriff auf assoziierte `Persistence` unterscheidet sich dabei in keiner Weise vom Zugriff auf normale assoziierte Objekte (Abschnitt 9.6).

Das in diesem Kapitel beschriebene Persistence-Framework legt den Fokus darauf, Java-Objekte in einer relationalen Datenbank zu speichern und nicht etwa ein bestehendes relationales Schema auf Java-Objekte abzubilden. Mit der hier vorgestellten Lösung versuchen wir, dem Entwickler alle Vorteile von Objekten zu erhalten, ohne dabei auf die liebgewonnene Eigenschaften der Nutzung und der niedrigen Kosten einer relationalen Datenbank verzichten zu müssen.

Das Persistence-Framework soll folgenden Anforderungen genügen:

- Eindeutigkeit der Objekte
- Suche nach einer Menge von Objekten mit gleichen Eigenschaften
- Datenbankanfragen mit SQL
- Navigation über die Assoziationen eines Objektes
- Erzeugen neuer Objekte
- Speichern veränderter Objekte
- Speicherstrukturen von Mengen und Objektnetzen
- Unterstützung von Transaktionen
- Anbindung an die relationale Datenbank über das JDBC-API
- Einfache Erweiterbarkeit
- Zwischenspeichern von Objekten (Caching)

An dieser Stelle sei nochmals darauf hingewiesen, daß sich der gesamte Quellcode auf unserer Website befindet. Da der Umfang der Realisierung dieses Buch sprengen würde, werden wir in diesem Kapitel ausschließlich auf den Entwurf und die Anwendung eingehen.

Bevor wir das Framework im Detail vorstellen, möchten wir kurz die Grundlagen relationaler Datenbanken erklären. Wer in dieses Thema schon eingearbeitet ist, kann den Abschnitt 9.1 getrost überschlagen. Abschnitt 9.2 sollte allerdings jeder lesen, denn dort definieren wir einige notwendige Vorbereitungen der Datenbank für die Verwendung des Persistence-Frameworks.

Im Framework werden die SQL-Anweisungen `CREATE`, `SELECT`, `INSERT`, `UPDATE`, und `DELETE` gebraucht. Etwas über den Verbundausdruck (`JOIN`) und seine Modellierung zu wissen, ist jedoch hilfreich, um die Konfigurationsdateien zur Definition von Assoziationen besser zu verstehen [Date/Darwen 98, Abschnitt 11.2].

## 9.1 Grundlagen relationaler Datenbanken

Relationale Datenbanken verwalten Daten in Tabellen [Heuer/Saake 97]. Jede Zeile einer Tabelle definiert einen Datensatz, der aus Datenfeldern von Standarddatentypen besteht, die wir als Spalten bezeichnen. Wer schon mal mit einer Tabellenkalkulation gearbeitet hat, findet sich hiermit leicht zurecht. Ein Beispiel ist die folgende Restauranttabelle:

| **Nummer** | **Name** | **Beschreibung** |
|---|---|---|
| 123440 | Pizza Don Camillo und Peppone | Die Hombrucher Pizzeria |
| 123451 | Brunos Grillpfanne | Heiße Pommes und mehr |
| 123462 | Lung Sing | Die östliche Offenbarung |

*Tabelle 9.1: Die Tabelle »Restaurant«*

Um mit den Daten arbeiten zu können, bietet eine Datenbank Operationen zur Manipulation dieser Tabellen. Das Schema, die Struktur einer Tabelle, kann erzeugt und manipuliert werden. Ferner stellt die Datenbank – ähnlich wie ein Dateisystem – einfache Operationen zum Erzeugen, Manipulieren und Löschen der Datensätze einer Tabelle zur Verfügung.

Tatsächlich könnte man eine einzelne Tabelle genauso gut in einer Datei abspeichern. Der Vorteil einer relationalen Datenbank gegenüber einem Dateisystem liegt darin, viele verschiedene Tabellenstrukturen verwalten zu können und dem Nutzer trotzdem einen einheitlichen Zugriff auf die Tabellen zu gewähren. Besonders, wenn Sie spezielle Zugriffsarten auf Ihre Tabellen benötigen, welche eine logische Auswertung von Datenfeldern unterschiedlicher Tabellen voraussetzen, kommen die Vorteile eines relationalen Datenbanksystems zur Geltung. Natürlich sind die Behandlung von Transaktionen oder die Zugriffsgeschwindigkeit weitere Gründe für die Verwendung von relationalen Datenbanksystemen (Abschnitt 9.1.2). Mit einer zusätzlichen Tabelle zur Definition der Gerichte des Restaurants, wären wir zum Beispiel in der Lage herauszufinden, in welchem Restaurant wir welche Pizza essen könnten.

| **Restaurantnummer** | **Gericht** |
|---|---|
| 123440 | Pizza Inferno |
| 123451 | Pizza Vegetaria |
| 123462 | Pizza Hawaii |

*Tabelle 9.2: Die Tabelle »Gericht«*

Um mit begrenztem Aufwand zwischen verschiedenen Datenbankprodukten wechseln zu können, wurde SQL als Anfrage- und Definitionssprache für relationale Datenbank-Management-Systeme (RDBMS) definiert [Date/Darwen 98]. SQL ist im Standard SQL/92 der ISO und ANSI-Gremien definiert. Weiterhin existiert ein Entwurf SQL3. Im folgenden werden wir die für das Verständnis des Buches notwendigen Anweisungen kurz erklären.

### 9.1.1 SQL-Anweisungen

Eine komplette Beschreibung von SQL füllt viele dicke Bücher und würde diese Einführung sprengen [Date/Darwen 98, Heuer/Saake 97]. Trotzdem sind die grundlegenden Elemente relativ schnell erlernbar. Prinzipiell können wir neue Daten in eine Datenbank hineinschreiben, sie ändern oder löschen und selbstverständlich auch wieder auslesen. Im folgenden werden wir die wichtigsten Anweisungen kurz vorstellen.

#### *SQL-Anweisung CREATE*

Bevor wir Daten speichern können, müssen wir die entsprechenden Tabellen erzeugen. Dies geschieht mit der `CREATE`-Anweisung:

```
CREATE TABLE <Tabelle> (
   <Datenfeld> <Datentyp>,
   <Datenfeld> <Datentyp>,
   ...);
```

Genau wie in Java enden alle Befehle meist mit einem Semikolon. Bei einigen Herstellern ist es der Befehl `go`, oder das Endzeichen ist implementierungsabhängig. Groß-/Kleinschreibung ist meist irrelevant, kann aber bei einigen Produkten konfiguriert werden (z.B. Sybase und Oracle). Unsere einfache Restauranttabelle mit Nummer, Name und Beschreibung wird mit folgendem Befehl erzeugt:

```
CREATE TABLE Restaurant (
   Nummer INTEGER,
   Name VARCHAR(30),
   Beschreibung VARCHAR(255));
```

Eine Tabelle der Gerichte für jedes Restaurant wird folgendermaßen erzeugt:

```
CREATE TABLE Gericht (Nummer INTEGER, RestaurantNummer INTEGER, Beschreibung
VARCHAR(255));
```

Zusätzlich zu dieser einfachen Definition lassen sich Schlüsselfelder angeben, die der Identifikation eines Datensatzes dienen. Grundsätzlich gibt es zwei Arten von Schlüsseln. Der Primärschlüssel (`Primary Key`) ist eine eindeutige Eigenschaft eines Datensatzes einer Tabelle. Der Fremdschlüssel (`Foreign Key`) hingegen dient der Zuordnung eines Datensatzes zu einem oder mehreren Datensätzen einer anderen Tabelle. Er definiert also eine Beziehung zwischen den Datensätzen zweier Tabellen. Dabei wird nicht festgelegt, ob es sich um eine 1:1-Beziehung oder 1:N-Beziehung handelt, sondern nur

grundsätzlich, daß eine Beziehung über bestimmte Attribute besteht. Der CREATE-Anweisung mit diesen Definitionen hat folgende Form:

```
CREATE TABLE <Tabelle> (
   <Datenfeld>   <Datenfeldtyp>   <Datenfeldzusatz>,
   ...
   <Datenfeld>   <Datenfeldtyp>   <Datenfeldzusatz>,
   PRIMARY KEY (<Liste der Datenfelder des Primärschlüssels>),
   FOREIGN KEY (<Liste der Datenfelder>)
      REFERENCES <Name der Fremdtabelle>
      (<List der Datenfelder in der Fremdtabelle>));
```

Unser Beispiel lautet also korrekterweise:

```
CREATE TABLE Restaurant (
   Nummer INTEGER NOT NULL,
   Name VARCHAR(30) NOT NULL,
   Beschreibung VARCHAR(255),
   PRIMARY KEY (Nummer));
CREATE TABLE Gericht (
   Nummer INTEGER NOT NULL,
   RestaurantNummer INTEGER NOT NULL,
   Beschreibung VARCHAR(255) NOT NULL,
   PRIMARY KEY (Nummer),
   FOREIGN KEY(RestaurantNummer) REFERENCES Restaurant(Nummer) );
```

Mit dem Datenfeldzusatz kann zusätzlich noch die Bedingung festgelegt werden, ob eine Datenfeld null sein darf oder nicht (not null). In der Datenbank hat ein Attribut mit dem Wert null keinen Inhalt. Dabei gilt, daß ein Primärschlüssel niemals null sein darf. Mit der Angabe der Schlüssel ist eine Kontrolle von modifizierenden Befehlen durch die Datenbank verbunden. Ein Datensatz kann nicht gelöscht werden, solange noch andere Datensätze auf ihn verweisen.

#### *SQL-Anweisung INSERT*

Um einen neuen Datensatz anzulegen, existiert die INSERT-Anweisung:

```
INSERT INTO <Tabelle> (<Datenfeld1>,<Datenfeld2>,...)
   VALUES (<Wert1>, <Wert2>, ...);
```

Die Reihenfolge der Werte muß exakt der Reihenfolge der Datenfelder entsprechen.

```
INSERT INTO Restaurant (Nummer, Name, Beschreibung)
   VALUES (1234567,
   'Der kleine Spinatgenuß',
   'Eine vegetarische Verführung');
INSERT INTO Gericht (Nummer, RestaurantNummer, Beschreibung)
   VALUES (1,1234567,'Pizza Inferno');
```

#### *SQL-Anweisung UPDATE*

Die UPDATE-Anweisung ändert einen oder mehrere Datensätze:

```
UPDATE <Tabelle>
   SET <Datenfeld>=<Wert>, <Datenfeld>=<Wert>,...
   WHERE <Bedingung> AND|OR <Bedingung> ... ;
```

Die Änderung des eben eingefügten Datensatzes geschieht also folgendermaßen:

```
UPDATE Restaurant
   SET Name = 'Der große Spinatfeinschmecker'
   WHERE Nummer = 1234567 ;
```

### *SQL-Anweisung DELETE*

Zum Löschen dient die DELETE-Anweisung:

```
DELETE FROM <Tabelle>
   WHERE <Bedingung> AND|OR <Bedingung> ... ;
```

Mit Hilfe der folgenden Anweisung wird unser Beispiel-Datensatz wieder aus der Datenbank entfernt:

```
DELETE FROM Gericht
   WHERE Nummer = 1 ;
```

Das Löschen des Restaurantdatensatzes wäre zu diesem Zeitpunkt nicht möglich, da noch ein Datensatz der Gerichte mit einer Referenz auf das Restaurant existiert. Sie müssen in diesem Fall zuerst die zum Restaurant gehörenden Gerichte löschen.

### *SQL-Anweisung SELECT*

Nachdem die Tabelle erzeugt wurde, können wir mit ihr arbeiten. Die dazu verwendete Anweisung ist `SELECT`. Sie dient zur Auswahl bestimmter Datensätze aus der Datenbank.

```
SELECT <Datenfeld>, <Datenfeld>, ...
   FROM <Liste von Tabellen>
   [WHERE <Bedingung> [AND|OR <Bedingung> ...]] ;
```

Die Bedingungen sind Ausdrücke, welche die Anfrage an die Datenbank einschränken. Beispielsweise könnten wir unsere Restaurantdatenbank nach allen Restaurants mit einem bestimmten Namensteil fragen. Neben Gleichheit kann man zumeist auch auf Ungleichheit, Ähnlichkeit, einen bestimmten Wertebereich etc. testen. So können beispielsweise folgende Anfragen an eine Restauranttabelle gestellt werden:

```
SELECT *
   FROM Restaurant;
SELECT Nummer, Name
   FROM Restaurant
   WHERE Beschreibung like '%Spinat%' ;
SELECT Nummer, Name
   FROM Restaurant
   WHERE Nummer = 1234567;
SELECT t0.Nummer, t0.Name
```

```
FROM Restaurant t0, Gericht t1
WHERE t0.Nummer = t1.RestaurantNummer
         AND t1.Beschreibung like 'Pizza%' ;
```

Die erste Anfrage ermittelt alle Datensätze der Tabelle Restaurant. Der Stern dient dabei als Joker für alle Datenfelder. In der zweiten Anfrage werden alle Nummern und Namen der Restaurants gesucht, deren Beschreibung das Wort Spinat enthält. Die dritte Anfrage bestimmt das Restaurant mit der Nummer 1234. In der vierten Anfrage werden alle Restaurants gesucht, die Pizzen anbieten. Dazu haben wir die Aliasnamen `t0` und `t1` für die beiden Tabellen benutzt. Mit Hilfe dieser Aliasnamen können wir die in der Anfrage und dem Ergebnis verwendeten Datenfelder einfacher ihren Tabellen zuordnen. Zudem ermöglicht dies, Datenfelder gleichen Namens, jedoch unterschiedlicher Tabellen, auseinanderzuhalten.

### 9.1.2 Transaktionen

Nachdem wir die grundlegenden SQL-Anweisungen erklärt haben, möchten wir noch kurz auf das Transaktions-Konzept eingehen. Wenn mehrere Anweisungen einen Datenbankzustand ändern oder mehrere Nutzer gleichzeitig auf die selben Datenbestände zugreifen, ist es notwendig, diese Anweisungen als untrennbare Einheit (atomar) begreifen zu können, um unerwünschte Nebeneffekte, wie Inkonsistenz der Daten oder eine falsche Sicht auf die Daten, auszuschalten. Diese Einheit heißt Transaktion.

Transaktionen sind unverzichtbar, wenn es darum geht, den Datenbestand in einem konsistenten Zustand zu halten. Kann eine Anweisung, die Teil einer Transaktion ist, nicht korrekt ausgeführt werden, müssen alle durch diese Transaktion bereits erfolgten Änderungen der Datenbank wieder zurückgenommen werden, ohne daß dafür weitere Anweisungen vom Nutzer erzeugt werden.

Transaktionen werden von allen ernsthaften SQL-Datenbanken unterstützt. Gewöhnlich wird vor jeder Anweisung eine Transaktion geöffnet (`BEGIN`, dies geschieht oft implizit, ist jedoch implementierungsabhängig), alle folgenden Anweisungen werden durch eine Bestätigung (`COMMIT`) oder Rücknahme (`ROLLBACK`) abgeschlossen. Leider ist im SQL-Standard die Behandlung von Transaktionen nicht genau festgeschrieben, so daß sich verschiedene Produkte durchaus unterschiedlich verhalten (Einstellung des Isolierungsgrads). Die von uns in den Beispielen verwandte finnische Datenbank Solid sperrt zu Beginn einer Transaktion alle Tabellen, so daß bis zum Schluß der Transaktion die Daten für keine andere, gleichzeitige Transaktion sichtbar werden (Abbildung 9.1). Dies gilt insbesondere auch für die `SELECT`-Anweisung. Für andere Datenbankprodukte trifft dies nicht unbedingt zu (`Dirty Read`)[Date/Darwen 98, S.81ff].

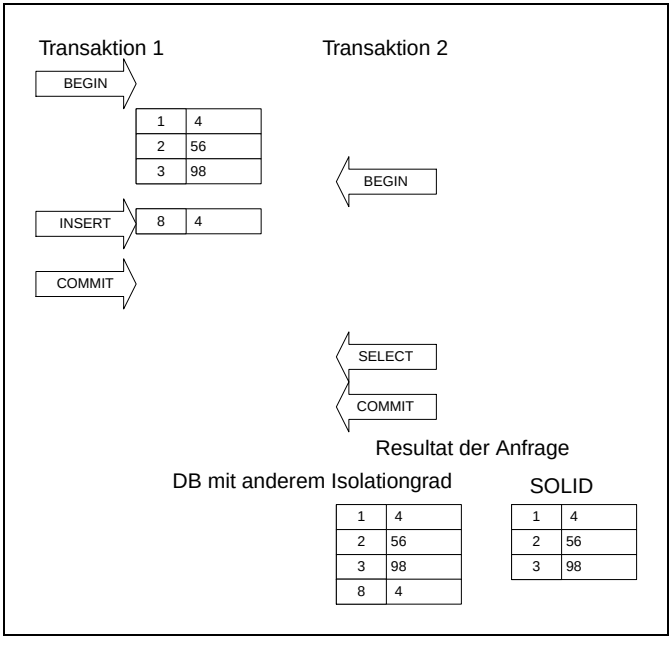


*Abbildung 9.1: Die unterschiedlichen Ergebnisse verschiedener Datenbanken bei gleichzeitigen Transaktionen*

Ansonsten gibt es jede Menge weitere Anweisungen, welche die Datenbank steuern, Datentypen manipulieren, Autorisierungen erteilen oder die Nutzer verwalten. Die Autorisierung erteilt verschiedenen Nutzern der Datenbank Rechte zum Betrachten und Ändern einzelner Tabellen. Für ein tieferes Verständnis sei auf die entsprechenden Handbücher oder die einschlägige Literatur zum Thema SQL und Datenbanken verwiesen [Date/Darwen 98, Heuer 92].

## 9.2 Tabellen für Persistence

Für unser Ziel, `Persistence` in einer relationalen Datenbank speichern zu können, bedarf es einiger zusätzlicher Konventionen.

### 9.2.1 Generierung von Identitätsschlüsseln

Eine grundsätzliche Eigenschaft von Objekten ist ihr eindeutiger Bezeichner [Booch 92, Holl/Schorsch 98]. Auch unsere persistenten Objekte sollen über diese Eigenschaft verfügen. Insbesondere sollte jedes Objekt über alle Tabellen hinweg diese Eindeutigkeit behalten. Was nützt es, wenn der Wert `12340` ein Restaurant, einen Lieferanten und eine Person bezeichnet? Wir möchten, daß aus dem Bezeichner der Typ eines Objektes, also seine Klasse, ersichtlich wird. Mit dieser Eigenschaft läßt sich ein sehr einfacher und genereller Zugriff auf verschiedene `Persistence` mit einer Methode verwirklichen

(Abschnitt 9.4.1 Methode `newPersistence()`). Da es sich damit bei diesem Bezeichner um ein internes Datenfeld handelt, müssen die Werte automatisch bestimmt werden.

Zum Generieren dieser Identitätsschlüssel ist es notwendig, in der Datenbank eine Zuordnung von Typennamen zu Nummern zu hinterlegen. Damit die Nummer für jedes einzelne Objekt verschieden ist, müssen wir einen Zähler in der Datenbank speichern.

Da wir diesen Zähler nicht bei jedem Erzeugen eines Datensatzes extra speichern möchten, soll jede Verbindung zur Datenbank zu Beginn eine Anzahl von Zählern zugeteilt bekommen. Wir wenden dazu das Konzept der Zählerkonten an.

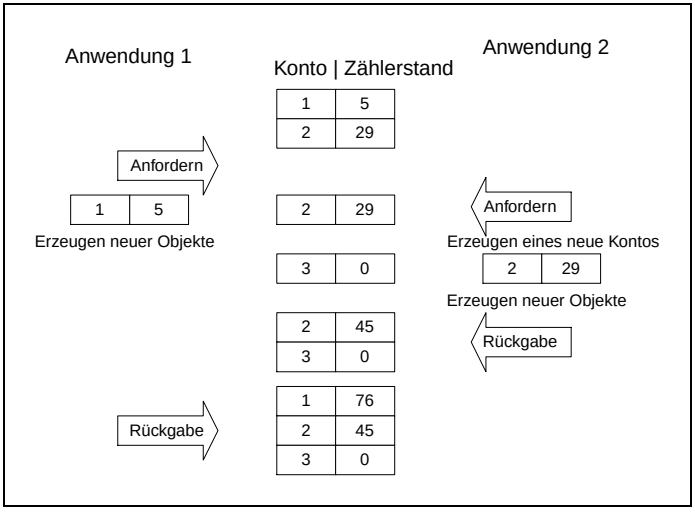


*Abbildung 9.2: Anforderung und Rückgabe von Zählerkonten für die Generierung von Objektidentitäten*

Jede Verbindung entnimmt dabei das kleinste verfügbare Konto. Falls kein Konto verfügbar ist, wird ein Konto mit der nächsthöheren Kontonummer in der Datenbank angelegt (Abbildung 9.2). Für diese Operation benötigt die Verbindung exklusiven Zugriff auf die Tabelle `AvailableAccount`, die die Zählerkonten verwaltet. Die folgenden SQL-Anweisungen legen die Tabelle an und erzeugen das erste Konto:

```
CREATE TABLE AvailableAccount (
   AccountNumber INTEGER NOT NULL,
   Counter INTEGER NOT NULL);
INSERT INTO AvailableAccount
   VALUES (1, 0);
```

Der Algorithmus zur Entnahme eines Kontos lautet:

1. Setze Schreibsperre auf verfügbare Konten, durch `UPDATE` auf alle Sätze von `AvailableAccount`.
2. Bestimme kleinstes Konto.
3. Nehme das kleinste Konto.
4. Falls kleinstes und größtes Konto gleich sind, füge ein neues größtes Konto hinzu.
5. Entnehme und lösche das kleinste Konto.

Besitzt eine Anwendung ein Konto, so erhält sie durch einfaches Erhöhen des Zählers einen für dieses Konto eindeutigen Bezeichner. Da es keine zwei gleichen Kontonummern gibt, die Kontonummern also eindeutig in der Datenbank sind, ist ein aus Kontonummer und Zähler gewonnener Bezeichner ebenfalls eindeutig.

Um den verfügbaren Zahlenraum möglichst effizient zu nutzen, wird das von einer Datenbankverbindung genutzte Konto beim Schließen der Verbindung wieder in der Tabelle `AvailableAccount` eingetragen. Beim Absturz einer Anwendung geht das Konto verloren. Mit etwas Aufwand können diese Konten natürlich wiedergewonnen werden. Langfristig zu nutzende Anwendungen sollten unbedingt einen größeren Zahlenraum verwenden (Abschnitt 9.2.3).

### 9.2.2 Kodierung des Objekttyps

Zum Generieren einer Objektidentität, die den oben genannten Anforderungen genügt, müssen wir zu Kontonummer und Zählerstand die Typinformation hinzufügen. Dazu wird die Tabelle `TypeNumber` (Tabelle 9.3) benutzt. Die folgende SQL-Anweisung legt die Tabelle an:

```
CREATE TABLE TypeNumber (
   Type VARCHAR NOT NULL,
   TypeNumber INTEGER NOT NULL);
```

In ihr wird eine einfache Zuordnung von Typnamen zu Typnummern definiert. Bei den Typnamen darf es sich um beliebige Namen für Klassen, Klassengruppen oder Schnittstellen handeln. Letztlich sind es nur Platzhalter, bzw. Aliasname. Genauso verhält es sich mit der Typnummer, die beliebig gewählt werden kann. Einzige Bedingung ist, daß einer Typnummer nicht mehrere Typnamen zugewiesen werden.

| Type | TypeNumber |
|---|---|
| Restaurant | 10 |
| Gericht | 11 |
| ... | ... |

*Tabelle 9.3: Zuordnung von Typnamen zu Typnummern*

### 9.2.3 Generierung der Objektidentität

Die Kodierung der Objektidentität erfolgt so:

```
<Zähler 9-stellig><Konto 4-stellig><Typ 3-stellig>
```

Die Werte jedes Zählers, Kontos und Typs werden einzeln rechtsbündig in einer 16stelligen Zahl gespeichert. Die fehlenden Stellen jedes Blocks werden mit einem Unterstrich aufgefüllt. Zur Zeit werden nur Ziffern verwendet.

Falls der hier angestrebte Zahlenraum sich als zu klein erweisen sollte, kann er durch die Kodierung zu einer anderen Basis erweitert werden. Zum Beispiel wäre die Basis 36 durch die Hinzunahme aller Buchstaben möglich [Polar 98]. Natürlich ist auch die Verlängerung der Zeichenkette möglich. Das Format zur Identitätsgenerierung sollte vor der Erzeugung des ersten Objektes festgelegt werden, da sonst später alle Objekte und Referenzen angepaßt werden müssen.

### 9.2.4 Verhindern von unbemerkten Änderungen durch andere Anwendungen

Jede Anwendung hat ihre eigene unabhängige Kopie eines Objekts. Da zu einer Datenbank oft mehrere Verbindungen gleichzeitig existieren, kann es zu Konflikten beim Speichern von Objekten kommen. Problematisch wird es, wenn mehrere Anwendungen das selbe Objekt modifizieren.

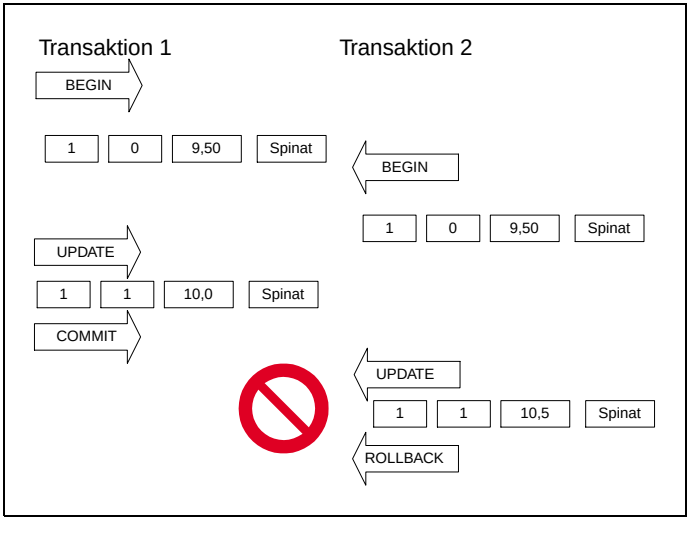


*Abbildung 9.3: Änderungskonflikt aus anderen Transaktionen*

Damit wir bemerken, ob eine andere Anwendung vor uns das Objekt geändert hat, speichert jedes Objekt seine Versionsnummer, wenn es modifiziert wird. Jede Anwendung muß bei einer Modifikation prüfen, ob sie die richtige Version des Objekts ändert

oder löscht. Dafür wird jeder Tabelle das Datenfeld `ObjectVersion` hinzugefügt. Die Version wird bei jeder Änderung um Eins hochgezählt. Zudem wird ein `UPDATE` immer mit der Bedingung `'AND ObjectVersion = <Versionswert>'` ausgeführt. Eine Änderung erfolgt mit der Anweisung:

```
UPDATE Restaurant
   SET ObjectVersion = 1,
   Nummer = 1234567,
   Beschreibung = 'Die vegetarische Versuchung',
   Name = 'Die Spinatfalle'
   WHERE ObjectIdentifier = '________1___1_10' AND ObjectVersion = 0 ;
```

Bei jeder Änderung eines Objektes werden alle Attribute bis auf die Identität immer mitverändert – egal ob sie sich wirklich geändert haben oder nicht. Ein spezieller Abgleich kostet nur Speicherplatz und Rechenzeit. Das hier genutzte Verfahren vereinfacht zudem die Generierung der Befehle massiv und ermöglicht die Verwendung der JDBC-Anweisung `PreparedStatement`. Falls die Datenbank es unterstützt, wird der Befehl vor seiner Anwendung übersetzt und bei der Ausführung müssen nur noch die Werte zur Datenbank übertragen werden [Hamilton et al. 97, S.158].

## 9.3 Definition der Java-Objekte

Wenn wir bisher davon sprachen, Java-Objekte in der Datenbank zu speichern, so bezeichneten wir diese immer als *bestimmte* Java-Objekte. Die Einschränkung bezieht sich hauptsächlich darauf, daß wir als Attributtypen konsequent Objekttypen und keine skalaren beziehungsweise atomare Datentypen verwenden, da skalare Datentypen zu Problemen mit dem Datenbankwert `null` führen würden. In einem skalaren Java-Datentyp `int`, `double` oder `boolean` ist der Wert `null` nicht erlaubt [Gosling et al. 96, S. 21ff]. Um trotz skalarer Typen nicht auf `null` zu verzichten, müßte man einen Wert aus dem Wertebereich des skalaren Datentyps bestimmen, der die `null` abbildet, zum Beispiel für `int` den Wert `-1`. Wenn man aber genau diesen Wert trotzdem benötigt, hat man ein größeres Problem. Mit der Verwendung von echten Objekten ist ein eindeutiger Wert für `null` gegeben und kann entsprechend einfach in der Datenbank abgebildet werden.

Die Menge der Datentypen der Attribute ist in Tabelle 9.4 aufgeführt. Dabei sind die Objektdatentypen durch JDBC praktisch vorgegeben:

| JDBC-Typ/DB-Typ | Java-Objekttyp |
|---|---|
| BIGINT | java.lang.Long |
| BIT | java.lang.Boolean |

*Tabelle 9.4: Abbildung der Datenbanktypen auf Attributdatentypen*

| JDBC-Typ/DB-Typ | Java-Objekttyp |
|---|---|
| CHAR | java.lang.String |
| DATE | java.sql.Date |
| DECIMAL | java.math.BigDecimal |
| DOUBLE | java.lang.Double |
| FLOAT | java.lang.Double |
| INTEGER | java.lang.Integer |
| LONGVARCHAR | java.lang.String |
| NUMERIC | java.math.BigDecimal |
| REAL | java.lang.Float |
| SMALLINT | java.lang.Integer |
| TIME | java.sql.Time |
| TIMESTAMP | java.sql.Timestamp |
| TINYINT | java.lang.Integer |
| VARCHAR | java.lang.String |

*Tabelle 9.4: Abbildung der Datenbanktypen auf Attributdatentypen (Fortsetzung)*

Die Tabellen aus dem Beispiel des Restaurants müßten für unser Persistence-Framework also folgendermaßen erstellt werden:

```
CREATE TABLE Restaurant (
   ObjectIdentifier CHAR(16) NOT NULL,
   ObjectVersion INTEGER,
   Nummer INTEGER NOT NULL,
   Name VARCHAR(30) NOT NULL,
   Beschreibung VARCHAR(255),
   PRIMARY KEY (ObjectIdentifier));
INSERT INTO TypeNumber ('Restaurant',10) ;
CREATE TABLE Gericht (
   ObjectIdentifier CHAR(16) NOT NULL,
   ObjectVersion INTEGER,
   RestaurantID CHAR(16) NOT NULL,
   Beschreibung VARCHAR(255) NOT NULL,
   PRIMARY KEY (ObjectIdentifier),
   FOREIGN KEY (RestaurantID) REFERENCES Restaurant(ObjectIdentifier));
INSERT INTO TypeNumber ('Gericht',11) ;
```

Um Ihre Anwendung komfortabler zu gestalten, müssen Sie ein bestehendes Schema ändern. Es gibt zur Zeit keine Möglichkeit, direkt eine bestehende Datenbank zu nutzen. Eine sanfte, aber aufwendige Migration ist allerdings möglich, indem Sie in allen Tabellen die Datenfelder `ObjectIdentifier` und `ObjectVersion` mit den entsprechenden Werten einfügen [Holl/Schorsch 98]. Natürlich müssen danach alle – auch die vorher bestehenden – Anwendungen gültige Identitäten und Versionsnummern erzeugen. Dies lohnt sich sicher nur selten.

## 9.4 Architektur des Persistence-Frameworks

Die Aufgabe eines Persistence-Frameworks ist es, den Zustand bestimmter Objekte einer Anwendung über die Lebenszeit eines Prozesses hinaus zu speichern. Solche Objekte werden als *persistente* Objekte bezeichnet. Persistente Objekte besitzen durch das Persistence-Framework eine Repräsentation in einer relationalen Datenbank. Die Anwendung kann so nach einem Neustart wieder auf ihre persistenten Objekt zugreifen, als wäre sie nie terminiert. Das Persistence-Framework ist also in der Lage, den Zustand eines Objekts, das durch eine Menge bestimmter Attribute beschrieben ist, in einer Datenbank zu speichern und es daraus jederzeit wieder zu laden.

Im Persistence-Framework werden die gesamten Dienstleistungen über das zentrale Objekt `Store` zur Verfügung gestellt. Über den `Store` können neue `persistente` Objekte erzeugt, von der Datenbank geladen und modifiziert werden. Der Name `Store` ist eine Analogie zu einem Warenhaus mit einer sortierten Warenpalette.

Neben dem `Store` bilden die Klassen `PersistencePeer` und `Persistence` das Fundament des Persistence-Frameworks (Abbildung 9.4). Im folgenden erhalten Sie einen Überblick über die Rollen dieser drei Klassen.

Der `Store` verwaltet alle allgemeinen, *nicht typspezifischen* Dienste. Das sind:

- der Transaktionsdienst `StoreTransactor`
- die Datenbankverbindung `StoreConnection`
- die typunspezifische Fabrik für persistente Objekte `StorePersistenceFactory`
- der typunspezifische Dienst `StoreModifier` zum Löschen, Ändern und Speichern von persistenten Objekten
- der typunspezifische Dienst zum Generieren von eindeutigen Objektidentitäten `StoreOIDAutonumber`

Darüber hinaus vermittelt `Store` die typspezifischen `PersistencePeers`. Dabei gilt, daß für jeden persistenten Objekttyp genau ein `PersistencePeer` existiert. Genau wie der `Store` vermittelt auch der `PersistencePeer` Dienste. Im Gegensatz zum `Store` sind seine Dienste jedoch *typspezifisch*. Diese sind:

- der Reflect-API-ähnliche Metadienst `PersistenceType`
- der typspezifische Suchdienst `PersistenceRetriever`
- die typspezifische Fabrik `PersistenceFactory`
- der Zwischenspeicher `PersistenceCache`
- der Dienst `PersistenceModifier` zum Löschen, Ändern und Speichern eines persistenten Objekts

Über seinen Suchdienst `PersistenceRetriever` ist ein `PersistencePeer` in der Lage, persistente Objekte seines Typs zu vermitteln. Diese nennen wir `Persistence`. Jedes `Persistence` verfügt über eine Repräsentation in der Datenbank, die über die Methoden `create()`, `delete()` und `update()` manipuliert wird. Es kann seine Repräsentation also erzeugen, löschen und aktualisieren. Darüber hinaus kennt jedes `Persistence` seinen `PersistencePeer` und kann über ihn auf den `Store` und alle seine Dienste zugreifen.

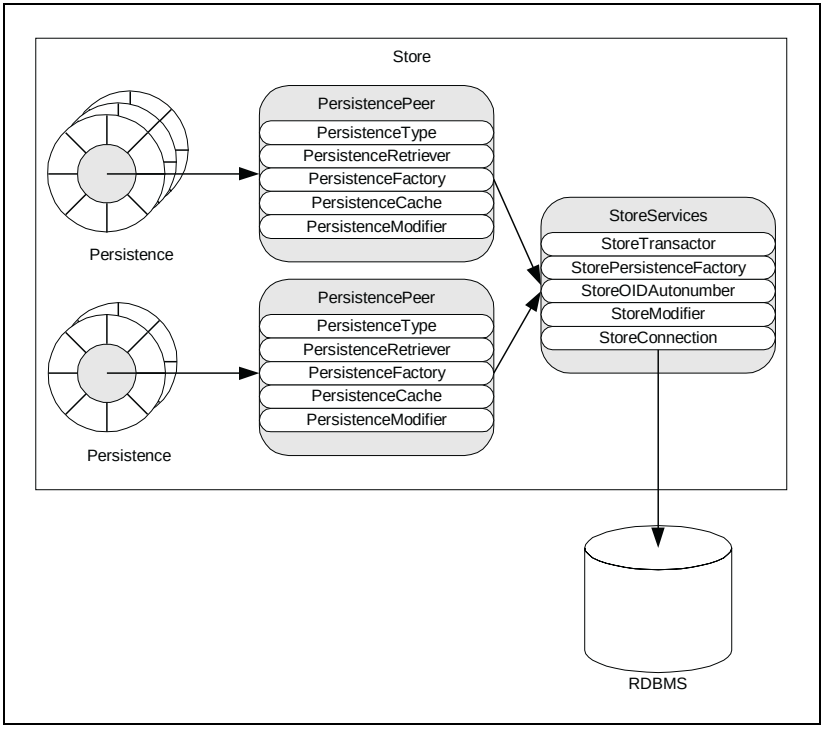


*Abbildung 9.4: Die Komponenten des Persistence-Frameworks*

### 9.4.1 Die Klasse Store

Von außen betrachtet hat der `Store` die Aufgabe, für eine Anwendung die Verbindung zur Datenbank herzustellen und diese zu kapseln. Er ist die zentrale Schnittstelle des Persistence-Frameworks. Über ihn werden Dienstleister vermittelt, welche die eigentliche Arbeit realisieren (Abbildung 9.4 und Abbildung 9.5).

So vermittelt der `Store` die Datenbankverbindung. Von dieser `StoreConnection` werden die Objekte `StorePersistenceFactory`, `StoreModifier` und `StoreOIDAutonumber` versorgt. Einmal initialisiert versehen sie kooperativ mit den anderen Diensten eine bestimmte, klar abgegrenzte Aufgabe.

Die `StorePersistenceFactory` ist für sämtliche lesenden Zugriffe auf die Datenbank zuständig. Sie realisiert die Anweisungen zum Laden von Tabelleninhalten. Dazu generiert sie die entsprechende SQL-`Select`-Anweisung für den verlangten `Persistence`-Typ und greift damit über die `StoreConnection` auf die Datenbank zu. Zusammen mit einer typspezifischen Fabrik werden die Objekte der Ergebnismenge erzeugt. Der genaue Vorgang ist im Abschnitt »Die Klasse `PersistenceRetriever`« beschrieben. Festzuhalten ist, daß jedes `Persistence`-Objekt, das durch eine Suche in die Anwendung gelangt, dabei zwangsläufig die `StorePersistenceFactory` passiert.

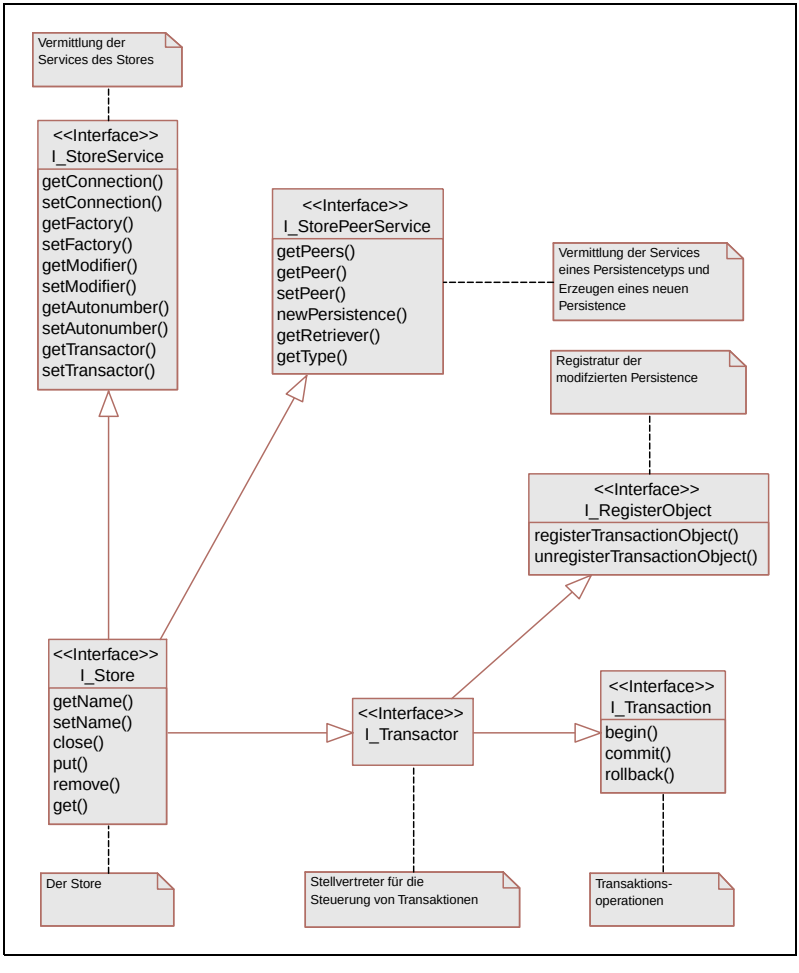


*Abbildung 9.5: Der `Store` und seine Dienstleister*

Die Klasse `StoreModifier` wird dazu benötigt, Modifikationen an einem `Persistence` in der Datenbank zu speichern. Das bedeutet, daß sämtliche schreibenden Zugriffe – also

`insert()`, `update()` und `delete()` – auf die Datenbank über diesen Dienst abgewickelt werden.

Um jedes `Persistence` eindeutig identifizieren zu können, existiert der Dienst `StoreOID-Autonumber` zur Generierung dieser Identität. Er generiert für jedes neue persistentes Objekt einen `ObjectIdentifier`, gemäß den in Abschnitt 9.2.3 beschriebenen Regeln. Soll der `ObjectIdentifier` anders generiert werden, muß diese Klasse geändert werden (Anhang B.1).

Um im Fehlerfalle alle `Persistence`-Objekte, die im Laufe einer Transaktion modifiziert wurden, nachladen zu können, wird der `Transactor` benötigt. Sämtliche `Persistence`-Objekte, die modifiziert, aber noch nicht gespeichert wurden, werden dazu beim `Transactor` registriert.

Über seine eigenen Dienste hinaus vermittelt der `Store` zentral für jeden `Persistence`-Typ einen `PersistencePeer`. Der `PersistencePeer` ist ebenfalls ein Vermittler von Diensten, nur sind seine Dienste auf einen spezifischen Typ konzentriert. Der `PersistencePeer` wird im Abschnitt »Die Klasse `PersistenceType`« genau erklärt. Um das Arbeiten mit dem `Store` zu vereinfachen, kann der `Store` den eigentlich typspezifischen Typ- und Suchdienst eines `PersistencePeers` auch direkt vermitteln.

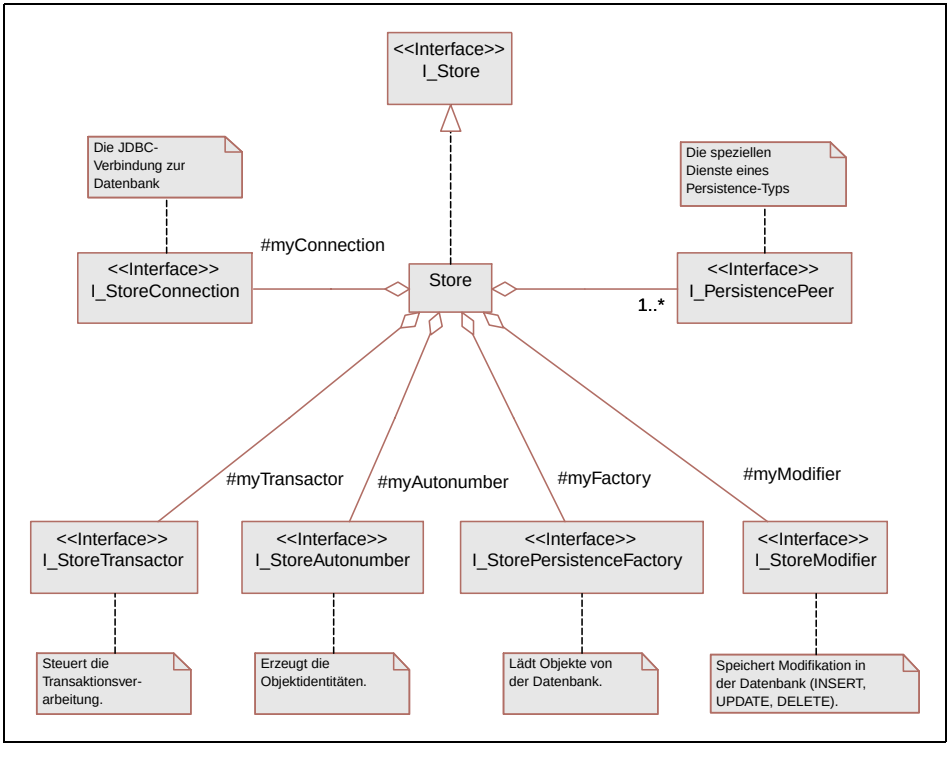


*Abbildung 9.6: Die Klasse `Store`*

Die Erzeugung einer `Store`-Instanz wird über ein Objekt vom Typ `StoreFactory` geleistet. Die `StoreFactory` ist in der Lage, einen `Store` anhand einer Konfigurations-Beschreibung zu instantiieren und zu initialisieren (Anhang B.1).

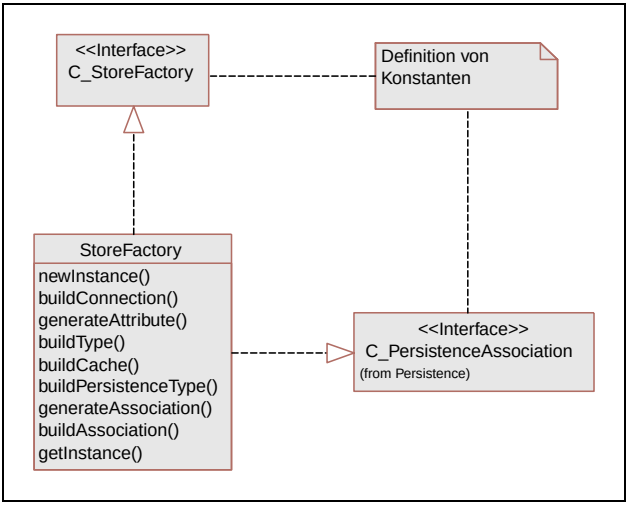


*Abbildung 9.7: Die Klasse `StoreFactory`*

Zunächst erzeugt `StoreFactory` (Abbildung 9.7) einen gänzlich leeren `Store`. Dann wird ein Objekt der `Store`-Klasse für die Datenbankverbindung instantiiert und an die Objekte `StorePersistenceFactory`, `StoreModifier` und `StoreOIDAutonumber` übergeben (Abbildung 9.8). Während der Initialisierung des `StoreOIDAutonumber`-Objekts wird die Liste der Typen gelesen und ein Konto zur Numerierung neuer `Persistence` belegt.

Nachdem die Basis-Dienste des `Stores` konfiguriert sind, werden alle `PersistencePeers` konfiguriert und beim `Store` registriert. Für die Registrierung der `PersistenceTypes` werden zwei Durchläufe benötigt. Im ersten Lauf werden die Typen mit ihren Attributen angelegt. Im zweiten Lauf werden für alle `PersistencePeers` die Assoziationen aufgelöst. Auf diese Weise kann man selbst kreisförmige Assoziationen konfliktlos anlegen.

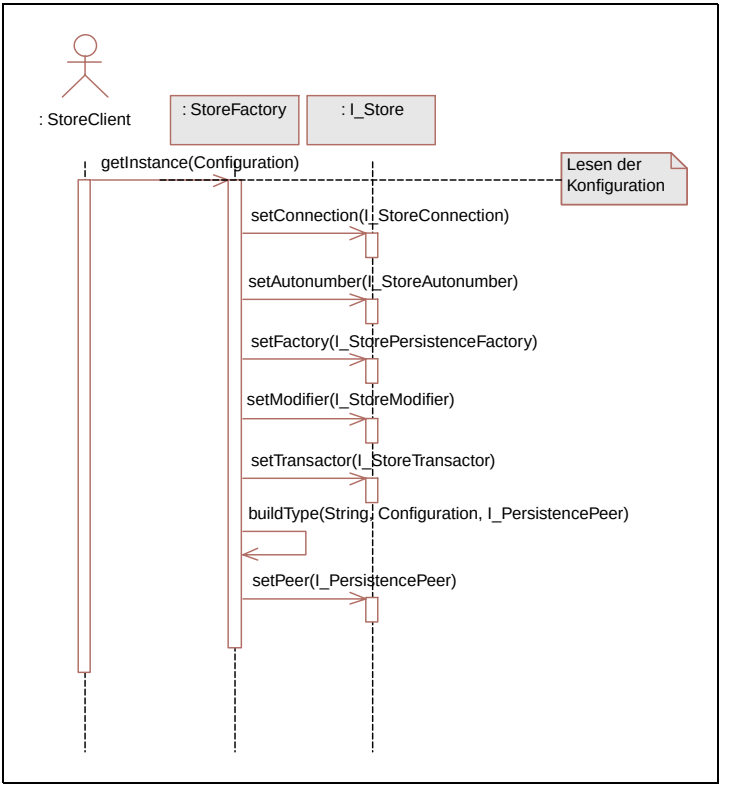


*Abbildung 9.8: Generierung eines* `Stores`

Nach der Konfiguration ist der `Store` betriebsbereit. Jeder Vorgang beziehungsweise jeder Zugriff auf ein `Persistence` des `Stores` muß in einem `synchronized`-Block stattfinden. So ist garantiert, daß nur ein `Thread` den `Store` gleichzeitig nutzt. Dies ist gerade in einer *multithreaded* Umgebung wie der Java-VM besonders wichtig. Ebenfalls, um die Konsistenz der Daten schützen zu können, muß vor der Nutzung des `Stores` eine Transaktion begonnen werden. Die erste Transaktion öffnet eine Datenbank-Transaktion. Jetzt können beliebig viele Operationen des `Stores` angewendet werden. Um Teilvorgänge sicher nutzen zu können, ist es möglich, weitere Transaktionen zu öffnen. Falls es zu einer Ausnahme kommt, wird die umschließende Datenbanktransaktion zurückgesetzt, und die letzte konsistente Version modifizierter Objekte wird neu aus der Datenbank geladen. Alle erfolgten Änderungen verfallen (Abschnitt 9.1.2).

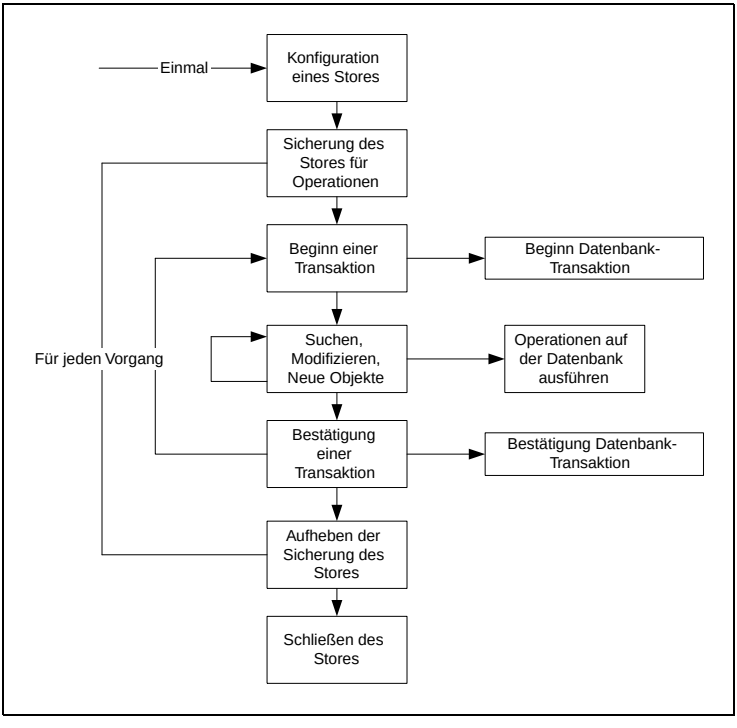


*Abbildung 9.9: Initialisierung und Zugriff eines* `Stores`

Nach Beendigung aller Anwendungsvorgänge muß der `Store` geschlossen werden, um benutzte Ressourcen wieder korrekt freizugeben. Falls nicht, wird dies durch die Speicherbereinigung in der Methode `finalize()` nachgeholt.

### 9.4.2 Der PersistencePeer

Ein `Persistence` ist immer genau einem `PersistencePeer` zugeordnet. Der `PersistencePeer` bietet die zentralen typspezifischen Dienste zum Erzeugen neuer Objekte, Suchen nach Objekten in der Datenbank und Speichern der Modifikationen. Weiterhin gehört eine Typbeschreibung und ein Cache zum Funktionsumfang. Im folgenden werden die Realisierungen der einzelnen Schnittstellen vorgestellt.

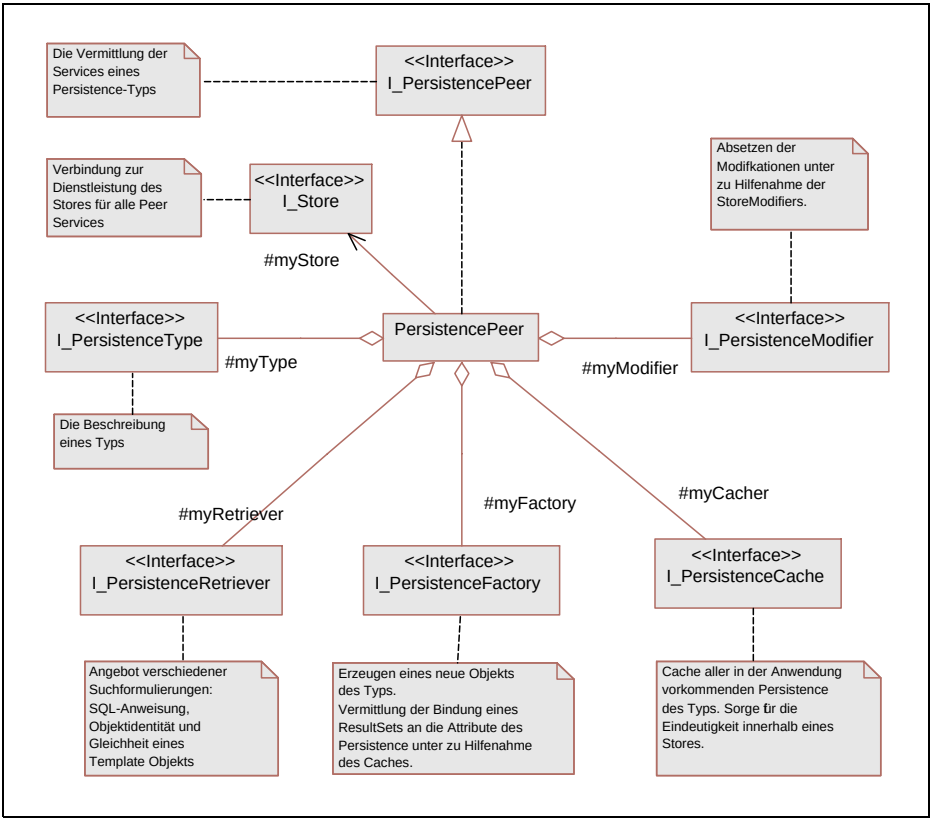


*Abbildung 9.10: Der `PersistencePeer` und seine Dienstleister*

### *Die Klasse PersistenceType*

Die Beschreibung des Typs eines `Persistence` ist eine echte Erweiterung des Java-Typsystems. Das Typ-Objekt eines `Persistence` beschreibt dessen persistente Attribute und Assoziationen (Abbildung 9.11). Ähnlich dem Reflect-API (`java.lang.reflect` – insbesondere `java.lang.reflect.Method` und `java.lang.reflect.Field`) ermöglichen die beschreibenden Objekte `PersistenceAttribute` und `PersistenceAssociation` den Zugriff auf konkrete Relationen und Attribute. So kann jedes `PersistenceAttribute` auf den Wert eines `Persistence` zugreifen oder ein `PersistenceAssociation`-Objekt ein Referenzobjekt der Assoziation besorgen (mehr dazu im Abschnitt 9.6). Durch diese Typbeschreibung verfügen wir über eine sehr große Flexibilität im Zugriff auf `Persistence`-Objekte (siehe das Beispiel StoreBrowser in Kapitel 11).

`PersistenceType` selbst beschreibt den Typ und die Abbildung auf die Datenbank eines `Persistence` auf mehrere Arten. Jeder Persistence-Typ verfügt über einen symbolischen Namen, einen Tabellennamen und einen Klassennamen. Der symbolische Name ist

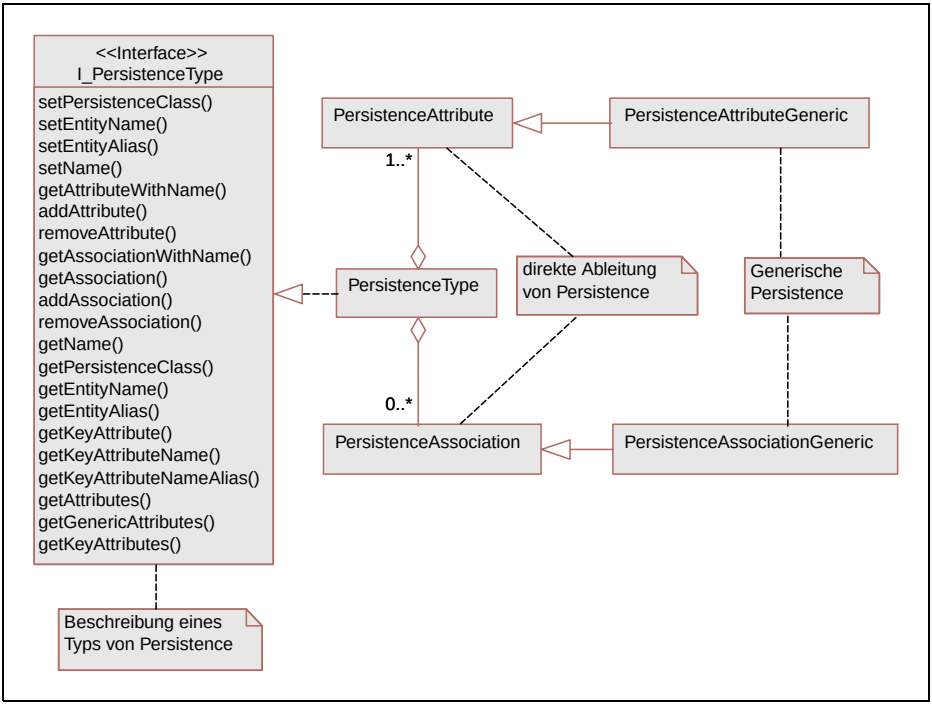


*Abbildung 9.11: Die Klasse* `PersistenceType`

gleich dem Typ-Namen in der Tabelle `TypeNumber`, auf die wir bereits in Abschnitt 9.2.2 eingegangen sind. Der Tabellenname – auch Entity genannt – gibt die Tabelle an, in der ein persistentes Objekt dieses Typs gespeichert wird. Und der Klassenname gibt die Klasse an, die instantiiert wird, um ein Objekt dieses Typs zu erzeugen.

Der Vorteil bei der Verwendung von symbolischen Namen, liegt in der starken Entkoppelung der beteiligten Objekte. Ein `Persistence` erhält seine Bindung zu einer konkreten Klasse zu einer Tabelle durch seine Konfigurationsdaten (Anhang B.1). Das bedeutet, daß die Tabellenbezeichnung auch einfach geändert werden kann. Ebenso kann die zu instantiierende Klasse eines `Persistence` mit einem bestimmten symbolischen Bezeichner einfach ausgetauscht werden. Dies ist besonders dann sinnvoll einzusetzen, wenn eine Klasse im nachhinein spezialisiert wurde. Langfristig garantiert dieses Vorgehen große Flexibilität.

*Die Klasse PersistenceRetriever*

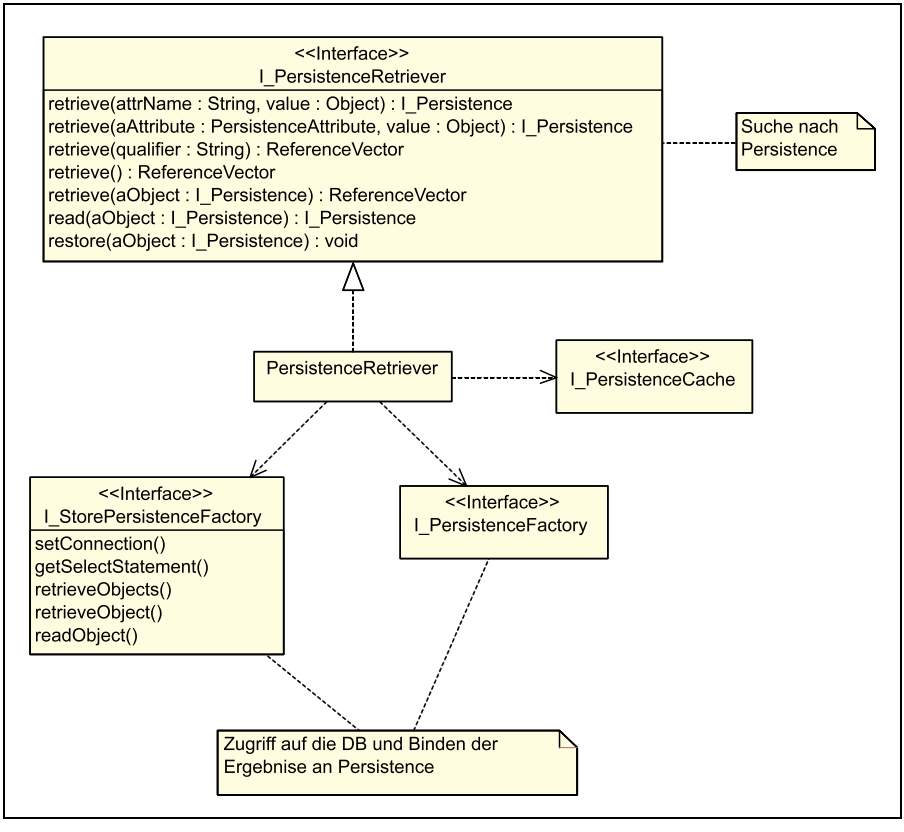


*Abbildung 9.12: Die Klasse* `PersistenceRetriever`

Zur Suche nach `Persistence`-Objekten wird zumeist eine SQL-Teilanweisung verwendet. Auf unser Restaurant-Beispiel könnte man beispielsweise folgende Anweisungen anwenden:

```
WHERE t0.Name LIKE 'Pizza%'
WHERE t0.Name LIKE 'Pizza%' AND t0.Beschreibung LIKE '%Spinat%'
```

Das Tabellenalias `t0` dient dabei als Stellvertreter der eigentlichen Tabellennamen. Für die Verbindung von Tabellen durch einen SQL-Verbundausdruck (`JOIN`) können vor dem `WHERE` die entsprechenden Tabellen angegeben werden. Wenn eine Menge von Gerichten zu dem Restaurant in der Tabelle Gerichte existiert, sieht die SQL-Anweisung folgendermaßen aus:

```
, Gericht t1 WHERE t0.ObjectIdentifier = t1.RestaurantID AND t1.Beschreibung LIKE
'%Pizza%'
```

Das Komma verbindet die Tabellen in der FROM-Angabe der SELECT-Anweisung. Die Suche wird über die StorePersistenceFactory abgewickelt.

Eine Qualifikation wird mit der Methode retrieve() realisiert:

```
public ReferenceVector retrieve(String qualifier)
    throws PersistenceException
{
    String command;
    if(qualifier != null)
      command = getSelectStatement() + qualifier ;
    else
      command = getSelectStatement() ;
   I_StorePersistenceFactory theFactory = myPeer.getStore().getFactory() ;
   return theFactory.retrieveObjects(command,myPeer.getFactory()) ;
}
```

*Listing 9.1: Die Methode PersistenceRetriever.retrieve(String)*

Fehlt die Qualifizierung, wird nach allen Persistence-Objekten dieses Typs gesucht. Die SELECT-Anweisung wird der Qualifizierung voran gestellt. Die Methode getSelectStatement() besorgt sich für ihren Typ den SELECT-Anteil bei der StorePersistenceFactory und speichert diese Anweisung für weitere Anfragen zwischen. Die vollständige Suchanweisung für Restaurants mit Pizzen lautet:

```
SELECT t0.ObjectIndentifier, t0.ObjectVersion, t0.Name, t0.Beschreibung
   FROM Restaurant t0, Gericht t1
   WHERE t0.ObjectIdentifier = t1.RestaurantID AND t1.Beschreibung LIKE '%Pizza%'
```

Die eigentliche Anfrage wird mittels StorePersistenceFactory.retrieveObjects() veranlaßt. Neben der Suchanweisung wird die Fabrik des Persistence-Typs übergeben (Abbildung 9.13). Diese Fabrik sorgt nach dem Bestimmen der Datenbank-Spalten für die Generierung der Objekte.

Zuerst wird in der Methode retrieveObjects(String,I_PersistenceFactory) geprüft, ob eine Verbindung zur Datenbank, der SELECT-Anweisung und die PersistenceFactory vorhanden sind (Listing 9.2). Es wird ein ReferenceVector für die Ergebnisse angelegt (Abschnitt »Das Zählen von Referenzen«). Nun wird über die JDBC-Connection ein java.sql.Statement erzeugt und an die Datenbank gesendet. Für jede Spalte des ResultSets wird dann über PersistenceFactory.getObject(ResultSet) ein Persistence-Objekt angefordert und zur Ergebnismenge hinzugefügt. Nachdem alle Spalten der Anfrage bearbeitet sind, wird das Statement mittels close() freigegeben und das Ergebnis zurückgegeben:

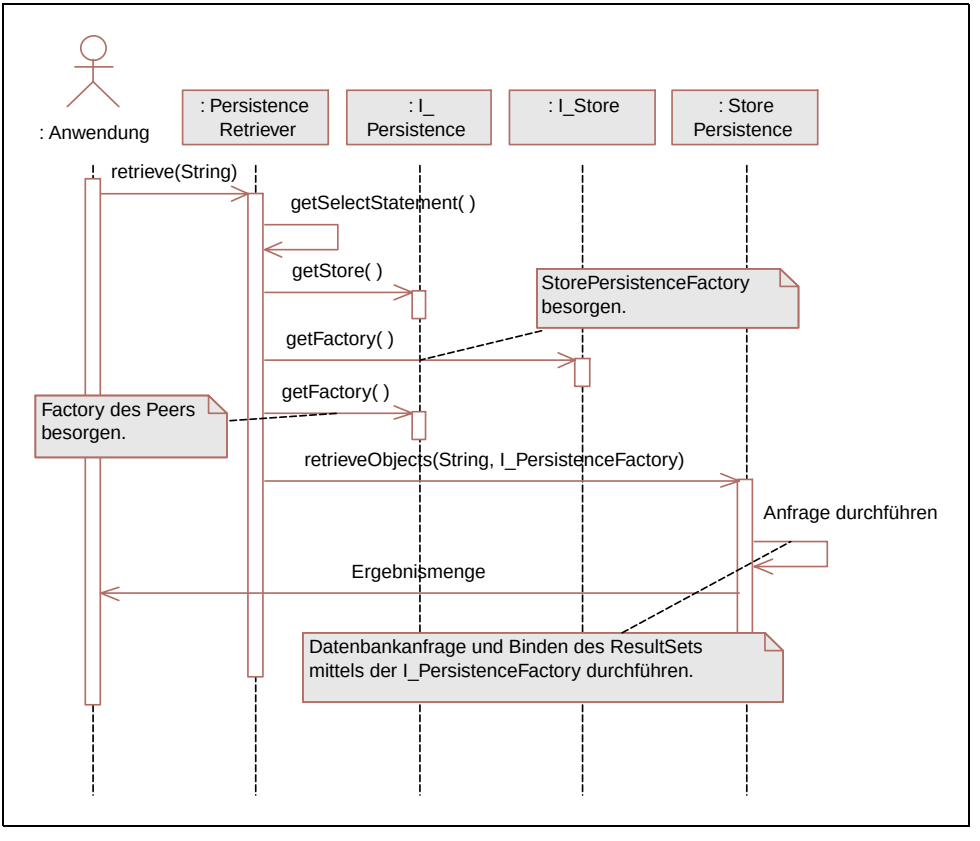


*Abbildung 9.13: Sequenzdigramm zur Methode* `PersistenceRetriever.retrieve(String)`

```
public ReferenceVector retrieveObjects(
   String command,
   I_PersistenceFactory factory )
throws PersistenceException
{
   if (myConnection == null || factory == null || command == null )
      throw new PersistenceException("Keine Suche für "
         + command + " möglich" );
   ReferenceVector objects = new ReferenceVector() ;
   Statement statement = null;
   try {
      if(Log.isLog())
         Log.log("START JDBC " + command);
       statement = myConnection.createStatement() ;
       ResultSet rs = statement.executeQuery(command);
       if(rs != null) {
          while (rs.next()) {
             I_Persistence obj;
```

```
            obj = factory.getObject(rs);
            if(obj != null)
               objects.addElement(obj);
         }
      }
      if(Log.isLog())
         Log.log("END JDBC - Ergebnis #" + objects.size() +" objects" );
   } catch (SQLException sqe) {
       throw new PersistenceException("Fehler beim Binden des Kommandos "
          + command + " aufgetreten.",sqe);
   } finally {
      if(statement != null )
         try {
            statement.close() ;
         } catch (SQLException sqe1) {
            throw new PersistenceException(
              "Fehler beim Schliessen des Kommandos " + command
             + " aufgetreten.",sqe1);
         }
      }
     return objects ;
}
```

*Listing 9.2: Die Methode `StorePersistenceFactory.retrieveObjects(String, I_PersistenceFactory)`*

Es gibt noch weitere Suchmethoden: Es existiert die Möglichkeit, mit der Methode `retrieve(I_Persistence)` alle `Persistence`-Objekte zu finden, die exakt die gleichen Attributwerte wie das übergebene Platzhalterobjekt (Template) haben. Weiterhin ist es mit der Methode `read(I_Persistence)` möglich, eine transiente, also nicht mehr speicherbare Kopie eines `Persistence` zu besorgen, beziehungsweise ein bereits bestehendes persistentes Objekt an die in der Datenbank gespeicherten Werte anzugleichen. Für das Nachladen von `Persistence` nach einer fehlgeschlagenen Transaktion ist die Methode `restore(I_Persistence)` zuständig.

An der Realisierung des `PersistenceRetriever` ist zu sehen, wie der Framework die verschiedenen Dienstleister des `Stores` und `PersistencePeers` nutzt. Es gibt eine klare Aufgabentrennung zwischen allgemeinen und typspezifischen Diensten.

### *Die Klasse PersistenceCache*

Aus zweierlei Gründen ist ein Cache im Persistence-Framework unverzichtbar. Zum einen möchte man Datenbankzugriffe vermeiden. Objekte, die also bereits existieren, sollten nicht noch einmal aus der Datenbank gelesen werden müssen. Zum anderen sollen es keine zwei Objekte geben, die das selbe Objekt in der Datenbank beschreiben. Nur so kann gewährleistet werden, daß Änderungen des Objekts innerhalb einer Anwendung sofort für alle sichtbar werden.

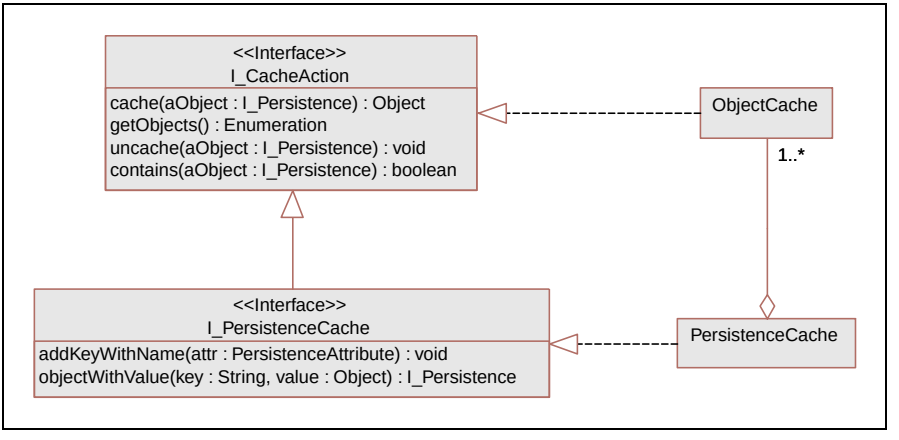


*Abbildung 9.14: Die Klasse* `PersistenceCache`

Die Klasse `PersistenceCache` dient dazu, diese Anforderungen umzusetzen. Aus Entwicklersicht ist sie ein Stellvertreter (Proxy) für mehrere `ObjectCaches` [Gamma et al. 96, Buschmann et al. 96]. Dabei existiert für jedes eindeutige Attribut eines `Persistence` ein separater Cache (Listing 9.3). Bei jedem Einfügen eines `Persistence` in den `PersistenceCache` wird es in die betreffenden `ObjectCaches` eingetragen.

```
public Object cache(I_Persistence aObject)
   throws PersistenceException {
   int     i;
   Object  theObject = null;
   if(aObject != null) {
      typeCheck(aObject) ;
      for(i=0; i<myKeyCount; i++)
          theObject =
          ((I_CacheAction)myKeyList.elementAt(i)).cache(aObject);
   }
   return theObject;
}
```

*Listing 9.3: Die Methode* `PersistenceCache.cache(I_Persistence)`

Dabei wird jedes `Persistence` vor der Übernahme in den Cache durch die Methode `typeCheck(I_Persistence)` auf seinen korrekten Typ überprüft. Die Gemeinsamkeiten von `PersistenceCache` und `ObjectCache` sind in der Schnittstelle `I_CacheAction` (Listing 9.4) beschrieben.

```
package de.webapp.Framework.Persistence;

import java.util.Enumeration ;
public interface I_CacheAction {
// Füge Persistence in den Cache ein
public Object cache(I_Persistence aObject) throws PersistenceException ;
```

```
// Gib alle Objekte aus dem Cache bekannt
public Enumeration getObjects();
// Entferne Objekt aus dem Cache
public void uncache(I_Persistence aObject) throws PersistenceException ;
// Ist Objekt im Cache?
public boolean contains(I_Persistence aObject) throws PersistenceException ;
} // Ende der Schnittstelle
```

*Listing 9.4: Die Schnittstelle `I_CacheAction`*

```
public I_Persistence objectWithValue(String key, Object identifier) {
   ObjectCache keyObj;
   I_Persistence ret = null;
   if(key != null) {
      keyObj = (ObjectCache) myKeyDict.get(key);
      if(keyObj != null)
         ret = keyObj.objectWithValue(identifier);
   }
   return ret;
}
```

*Listing 9.5: Die Methode `PersistenceCache.objectWithValue(String, Object)`*

Mit der Methode `objectWithValue(String key, Object identifier)` kann der Cache nach einem bestimmten `Persistence` befragt werden (Listing 9.5). Dabei wird durch den Schlüssel `key` gesteuert, welcher `ObjectCache` gewählt wird. Mit dem `identifier` wird der Wert des Schlüssels beschrieben. Im folgenden Abschnitt wird beschrieben, wie die `Persistence`-Objekte aus der Datenbank entstehen.

### *Die Klasse PersistenceFactory*

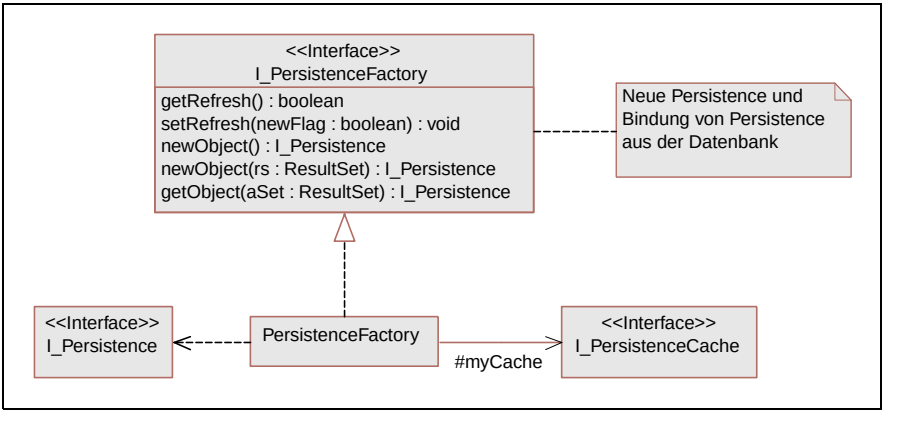


*Abbildung 9.15: Die Klasse `PersistenceFactory`*

Die Fabrik eines `Persistence`-Typs hat die Aufgabe, neue Objekte dieses Typs zu erzeugen. Dabei ist es möglich, ein vollkommen neues `Persistence` oder eines auf Basis einer

Zeile eines `ResultSets` aus der Datenbank zu erzeugen. Die `PersistenceFactory` bedient sich für das Binden eines `Persistence` eines `PersistenceCache`. Im Cache werden die `Persistence`-Objekte so lange gehalten, bis auf sie keine Referenz mehr aus der Anwendung besteht. Die Methode `getObject(ResultSet)` der `PersistenceFactory` erzeugt aus einer Zeile eines `ResultSets` ein `Persistence`-Objekt:

```
public I_Persistence getObject(ResultSet rs)
   throws PersistenceException {
   I_Persistence new_Obj = null;
   if(myCache != null) {
      try {
         // Key Attribute aus erster Stelle im ResultSet holen
         new_Obj = myCache.objectWithValue(
         myPeer.getType().getKeyAttributeName(),rs.getObject(1)) ;
      } catch( Exception e) {
          throw new PersistenceException("Binden eines Objekts des Typs "
             + getPeer().getType().getName() + " mit dem Resultset " + rs
             + "an den Cache fehlerhaft.",e) ;
      }
      if(new_Obj == null) {
         try {
            new_Obj = newObject(rs) ;
         } catch (PersistenceException pe) {
            throw pe ;
         } catch ( Exception e) {
               throw new PersistenceException("Objekt "
                  + myPeer.getType().getPersistenceClass()
                + " konnte nicht allokiert werden.",e) ;
         }
         myCache.cache(new_Obj);
      } else {
         switch(new_Obj.getObjectState()) {
         case C_Persistence.STATE_PERSISTENT:
         case C_Persistence.STATE_REFERENCEMODIFIED:
         case C_Persistence.STATE_OUTDATED:
            try {
               // Version aus der zweiten Stelle des ResultSet holen
               if(myRefreshFlag &&
                  !new_Obj.getObjectVersion().equals(
                     (Integer) rs.getObject(2))) {
                  if(Log.isLog())
                     Log.log("Erneutes Binden des Objekts " +
                        new_Obj.getObjectIdentifier()) ;
                  new_Obj.bind(rs);
               }
            } catch ( SQLException e) {
               throw new PersistenceException(
                  "Versionszugriff auf das Objekt "
                  + new_Obj + " mit ResultSet " + rs
                  + " fehlerhaft ", e) ;
```

```
            }
            break;
         }
       }
    } else {
      // Erzeuge ein neues Objekt mit dem ResultSet
      try {
         new_Obj = newObject(rs) ;
      } catch ( Exception e) {
         throw new PersistenceException("Objekt "
            + myPeer.getType().getPersistenceClass()
            + " konnte nicht allokiert werden ",e) ;
      }
    }
    return new_Obj ;
}
```

*Listing 9.6: Die Methode* `PersistenceFactory.getObject(ResultSet)`

Zuerst wird geprüft, ob das `Persistence` schon im Cache vorhanden ist. Dies ist möglich, da die erste Spalte eines `ResultSets` immer den eindeutigen `ObjectIdentifier` enthält und dieser als Schlüssel für den Cache genutzt werden kann. Wenn das `Persistence` noch nicht im Cache enthalten ist, wird ein neues `Persistence` mittels `newObject(ResultSet)` generiert. Das Binden der Attribute erfolgt in der Methode `Persistence.bind(I_PersistencePeer,ResultSet)`. Falls das `Persistence` im Cache vorhanden ist, wird geprüft, ob es nicht modifiziert wurde und in einer anderen Version vorliegt, um ein unsinniges erneutes Binden zu unterdrücken. Wenn kein Cache existiert, wird immer ein neues Objekt erzeugt.

### *Die Klasse PersistenceModifier*

Das Verändern von `Persistence` wird durch die Klasse `PersistenceModifier` realisiert. Beim ersten Gebrauch der modifizierenden Funktionen `create()`, `update()` oder `delete()`, wird das entsprechende `PreparedStatement` von einer der Methoden `getCreateStatement(I_PersistenceType)` (Listing 9.7), `getUpdateStatement(I_PersistenceType)` oder `getDeleteStatement(I_PersistenceType)` der Klasse `StoreModifier` generiert und für die weitere Verwendung hinterlegt:

```
public PreparedStatement getCreateStatement(I_PersistenceType aType)
   throws PersistenceException {
   PreparedStatement stst = null ;
   // Aufbau der Anweisung
   StringBuffer statement = new StringBuffer("INSERT INTO "
      + aType.getEntityName() + " " ) ;
   generateAttributes(statement,aType.getAttributes()) ;
   statement.append( " VALUES ");
   addParameter(statement,aType.getAttributes().size()) ;
   if(Log.isLog())
      Log.log( "Statement " + statement ) ;
```

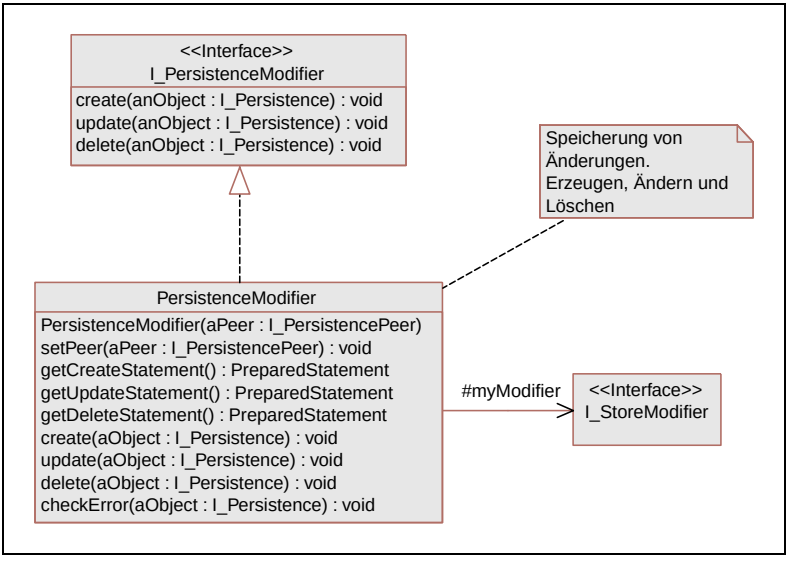


*Abbildung 9.16: Die Klasse* `PersistenceModifier`

```
   try {
      // Übergabe der Werte für die Anweisung
      stst = myConnection.prepareStatement(new String(statement)) ;
   } catch (SQLException sqe) {
      throw new PersistenceException(
         "Kommando Insert -- aTyp " + aType.getName()
         + " konnte nicht durchgeführt werden " ,sqe) ;
   }
   return stst ;
}
```

*Listing 9.7: Die Methode* `StoreModifier.getCreateStatement(I_PersistenceType)`

Die resultierende Anweisung für die SQL-`INSERT`-Anweisung sieht für die Tabelle `Restaurant` folgendermaßen aus:

```
INSERT INTO Restaurant
   (ObjectIdentifier,ObjectVersion,Name,Bezeichung)
   VALUES (?,?,?,?) ;
```

Mit dieser SQL-Anweisung wird über die JDBC-`Connection` des `Stores` ein `PreparedStatement`-Objekt erzeugt und an den `PersistenceModifier` zurückgegeben.

```
public int create(I_Persistence aObject)
throws PersistenceException {
   myPeer.getStore().registerTransactionObject(aObject) ;
   if(aObject.getObjectIdentifier() == null) {
      I_StoreAutonumber theNumber = myPeer.getStore().getAutonumber() ;
      aObject.setObjectIdentifier(theNumber,
```

```
            newObjectIdentifier(aObject)) ;
      }
      if(!myModifier.create(getCreateStatement(),aObject))
         return checkError(aObject) ;
      myPeer.getCacher().cache(aObject) ;
      return DBStateInDB ;
   }
```

*Listing 9.8: Die Methode `PersistenceModifier.create(I_Persistence)`*

Erzeugt wird ein `Persistence` mit der Methode `create(I_Persistence)` (Listing 9.8). Bevor der Zustand des `Persistence` in der Datenbank gespeichert wird, erfolgt die Registrierung beim `Transactor`. Der `Store` dient hier als Stellvertreter. Falls der `ObjectIdentifier` nicht gesetzt ist, wird eine neue Identität über die Autonumerierung angefordert. Über den `StoreModifier` wird das `Persistence` nun in der Datenbank gespeichert. Falls dies erfolgreich ist, wird das neue `Persistence` in den Cache aufgenommen.

```
public boolean create(PreparedStatement statement, I_Persistence aObject)
   throws PersistenceException {
   try {
      fillValues(statement,aObject,0) ;
      if(Log.isLog())
        Log.log( "Befehl create " + statement.toString()
           + " / " + aObject) ;
      statement.executeUpdate() ;
      return statement.getUpdateCount() != -1 ;
   } catch (SQLException sqe) {
      throw new PersistenceException(
        "Kommando Create --  Object " + aObject
        + " konnte nicht durchgeführt werden " , sqe) ;
   }
}
```

*Listing 9.9: Die Methode `StoreModifier.create(PreparedStatement,I_Persistence)`*

Die Methode `create(PreparedStatement,I_Persistence)` des `StoreModifiers` füllt die konkreten Werte der Attribute des `Persistence` in das vorgefertigte `PreparedStatement` (Listing 9.8). Danach wird der Befehl durch `executeUpdate()` an die Datenbank gesendet.

Bei einer `UPDATE`- oder `DELETE`-Anweisung liegt ein Fehler vor, wenn keine Zeile in der Datenbank durch die Anweisung modifiziert wird, da zumindestens die Objekt-Version um Eins erhöht werden muß.

### 9.4.3 Persistence-Objekte

Nachdem wir in den letzten beiden Abschnitten die Werkzeuge zur Verwaltung von persistenten Objekten genauer untersucht haben, wollen wir uns nun den persistenten Objekten selber widmen. Üblicherweise sind alle persistenten Objekte von der Basis-Klasse `de.webapp.Framework.Persistence.Persistence` abgeleitet. Die Klasse bietet die In-

frastruktur, mit deren Hilfe sich Geschäftsobjekte in einer relationale Datenbank abbilden lassen. Es ist dafür gesorgt, daß auch die Assoziationen zu anderen Geschäftsobjekten einfach realisiert werden können. Um sicher mit den Objekten arbeiten zu können, verfügen sie zudem über einen definierten Zustandsraum.

*Die Schnittstelle I_Persistence*

Um nicht an die `Persistence`-Klasse gebunden zu sein, sind alle wesentlichen Eigenschaften in der Schnittstelle `I_Persistence` (Abbildung 9.17) definiert. Alle persistenten Objekte realisieren die Schnittstelle `I_Persistence` und erfüllen damit folgende Bedingungen:

- Ein `Persistence` hat Zugriff auf seinen `PersistencePeer`-Dienst.
- Es besitzt eine Objektidentität, Objektversion und einen Zustand.
- Es kann von einer Fabrik erzeugt werden.
- Jede der Modifikationsoperationen `create()`, `update()` und `delete()` (`<op>`) ist durch eine `Hook`-Methode ergänzt. Diese Methode `can<op>()` kann dazu genutzt werden, spezielle Tests eines `Persistence` zu verankern, die vor der Operation die Attribute auf ihre Konsistenz prüfen.
- Über die `clone()`-Methode kann eine Kopie des Objekts angefertigt werden. Diese Kopie besitzt noch keine Identität und befindet sich im Zustand `UNDEFINED`.
- Über die Methode `transientClone()` wird ein vollständiges, transientes Objekt erzeugt.
- Die `read()`-Methode liest eine aktuelle Kopie von der Datenbank. Dieser Vorgang wird nicht über den Cache abgewickelt. Das Objekt ist danach `TRANSIENT`, soweit das Objekt nicht schon vorher `PERSISTENT` war.
- Die `restore()`-Methode wird für die Rücknahme einer Transaktion benötigt und liest ein persistentes Objekt erneut von der Datenbank. Dabei wird das Objekt zwischenzeitlich aus dem Cache entfernt. Dies geschieht, damit eventuell vorhandene weitere eindeutige Schlüssel ebenfalls bei der Veränderung korrekt berücksichtigt werden.

Die Schnittstelle `I_Persistence` erbt noch die Methoden der Schnittstellen `I_GenericValue` und `I_ReferenceCount`. Die Schnittstelle `I_GenericValue` stellt den Zugriff auf Werte aller persistenten Attribute einer Klasse zur Verfügung. Dies kann auf Basis des Attributnamens oder der Attribut-Beschreibung (`PersistenceAttribute`) erfolgen. Die zweite Schnittstelle `I_ReferenceCount` läßt das Zählen von Verwendungen des `Persistence` in der Anwendung zu. Diese Funktionalität wird ausgiebig im Abschnitt »Das Zählen von Referenzen« erläutert.

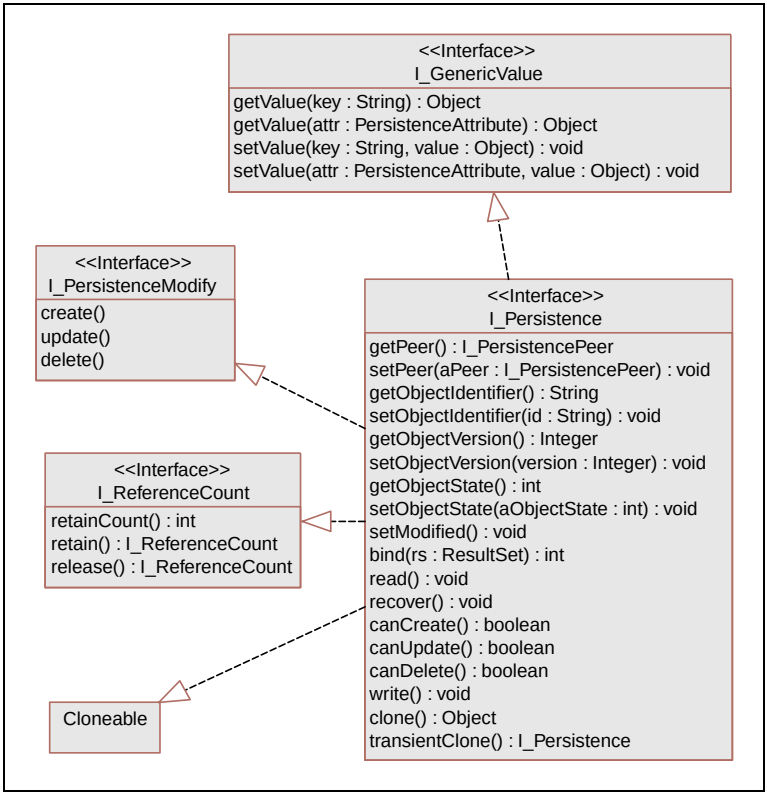


*Abbildung 9.17: Die Schnittstelle `I_Persistence`*

### *Die Klasse Persistence und ihre Ableitungen*

Die Klasse `Persistence` erfüllt die Anforderungen der Schnittstelle `I_Persistence`. Für die Modifikationsmethoden sind zwei weitere `Hook`-Methoden, `before<op>()` und `after<op>()`, hinzugenommen worden. In diesen Methoden werden zum Beispiel die Operationen zur Änderung von Assoziationen kodiert (Abschnitt 9.6.3). Wenn ein Objekt einer Assoziation hinzugefügt wird, wird die Änderung in der `afterCreate()`- beziehungsweise `afterUpdate()`-Methode entsprechend nachgezogen. Jedes `Persistence` stützt sich bei einer Modifikation auf den `PersistenceModifier`-Dienst seines `PersistencePeers` ab.

Für die generelle Anwendung eines `Persistence` ist die Klasse `PersistenceGeneric` bestimmt (Abbildung 9.18). Mit ihrer Hilfe müssen in der Prototypenphase eines Projekts keine echten Java-Klassen erzeugt und dann immer wieder geändert werden. Die Klasse `PersistenceLink` dient der Modellierungsunterstützung von `Link`-Klassen einer M:N-Assoziation (Abschnitt 9.6).

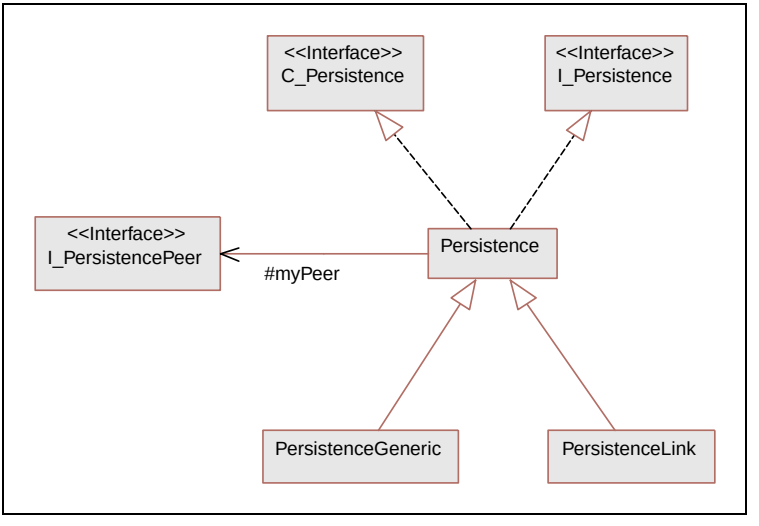


*Abbildung 9.18: Ableitungen der Klasse* `Persistence`

### *Zustandsraum von Persistence*

Um einen reibungslosen Ablauf beim Arbeiten mit persistenten Objekten gewährleisten zu können, ist es nötig, ihren Zustand zu verwalten (Tabelle 9.5 und Abbildung 9.19). Grundsätzlich ist dabei zu unterscheiden, ob ein `Persistence` in der Datenbank gespeichert ist und somit die Anwendung lediglich eine Kopie besitzt, oder ob das Objekt ausschließlich in der Anwendung benötigt wird, ohne jemals in der Datenbank gespeichert zu werden. Im ersten Fall sprechen wir von einem *persistenten* Objekt, das die Laufzeit der Anwendung überlebt, im zweiten von einem *transienten* Objekt, dessen Lebensdauer auf die Laufzeit der Anwendung beschränkt ist.

Beim Erzeugen befindet sich das `Persistence`-Objekt immer im Zustand `UNDEFINED`. Wenn es aus der Datenbank mittels einer `I_PersistenceFactory` geladen wird, überführt es die Methode `bind()` automatisch in den Zustand `PERSISTENT`. Wird ein Objekt jedoch gänzlich neu erzeugt, gelangt es erst nach erfolgreichem Einfügen in die Datenbank in den Zustand `PERSISTENT`.

Die Methode `read()` überführt ein `Persistence` in den Zustand `TRANSIENT`. Außerdem kann ein Objekt in den Zustand `TRANSIENT` durch das Anlegen einer Kopie mittels der Methode `transientClone()` gelangen. Ein transientes Objekt kann also durchaus aus der Datenbank stammen, seine Veränderungen können dann jedoch nicht mehr zurückgeschrieben werden.

| Zustand | Beschreibung |
|---|---|
| UNDEFINED | Ein Persistence ist in der Anwendung erzeugt worden. |
| PERSISTENT | Das Objekt befindet sich in der Anwendung und in der Datenbank. |
| TRANSIENT | Das Objekt befindet sich nur in der Anwendung und wird keinen persistenten Zustand erlangen. |
| MODIFIED | Das Objekt ist in der Anwendung modifziert worden und benötigt eine update()-Operation, damit die Datenbank auf den aktuellen Stand kommt. |
| REFERENCEMODIFIED | Mindestens ein Objekt in einer Assoziation ist in der Anwendung modifziert worden und benötigt eine create()-, update()- oder delete()-Operation damit die Datenbank auf den aktuellen Stand kommt. Der Zustand wird sonst wie MODIFIED behandelt. |
| DELETED | Das Objekt befindet sich nicht mehr in der Datenbank, aber die Anwendung hat es noch im Zugriff. |
| UNREFERENCED | Das Objekt befindet sich noch im Speicher der Anwendung, aber es gibt keinen sinnvollen Nutzen für das Objekt. Ein weiterer Zugriff auf dieses Objekt kann zu Problemen führen. |

*Tabelle 9.5: Zustände eines Persistence*

Ein persistentes Objekt kann jederzeit neu von der Datenbank geladen und Änderungen können auf die Datenbank übertragen werden. Die Modifikationsoperationen berücksichtigen dabei die Version, mit der das Objekt in die Anwendung gelangt ist, um zu verhindern, daß Änderungen anderer Anwendungen unberücksichtigt überschrieben werden. Der Framework erzeugt auf der Datenbank keine Sperren, sondern kontrolliert beim Ändern oder Löschen, ob die Version dieselbe ist. Falls das Objekt nicht mehr in der Datenbank vorliegt, wird der Zustand DELETED erreicht.

Jede Modifikation eines Attributs sollte mit der Methode setModified() abschließen und das Objekt in den Zustand MODIFIED überführen. Nachdem keine Anwendungsklasse mehr eine Referenz auf das Objekt hat, wird das Objekt in den Zustand UNREFERENCED überführt (Abschnitt »Das Zählen von Referenzen«).

### *Das Zählen von Referenzen*

Da das Framework alle Persistence-Objekte in einen Cache einträgt, stellt sich die Frage, wann und wie sie wieder entfernt werden. Sinnvoll ist es, sie zu entfernen, sobald sie nicht mehr von anderen Objekten als dem Cache referenziert werden. Erst Java 2 bietet für ein solches Verhalten einen robusten Mechanismus (JDK-API-Dokumentation zum Paket java.lang.ref). Da das Persistence-Framework aber auch das JDK 1.1 unterstützen soll, müssen sämtliche Persistence-Objekte die auf sie verweisenden Referenzen zählen.

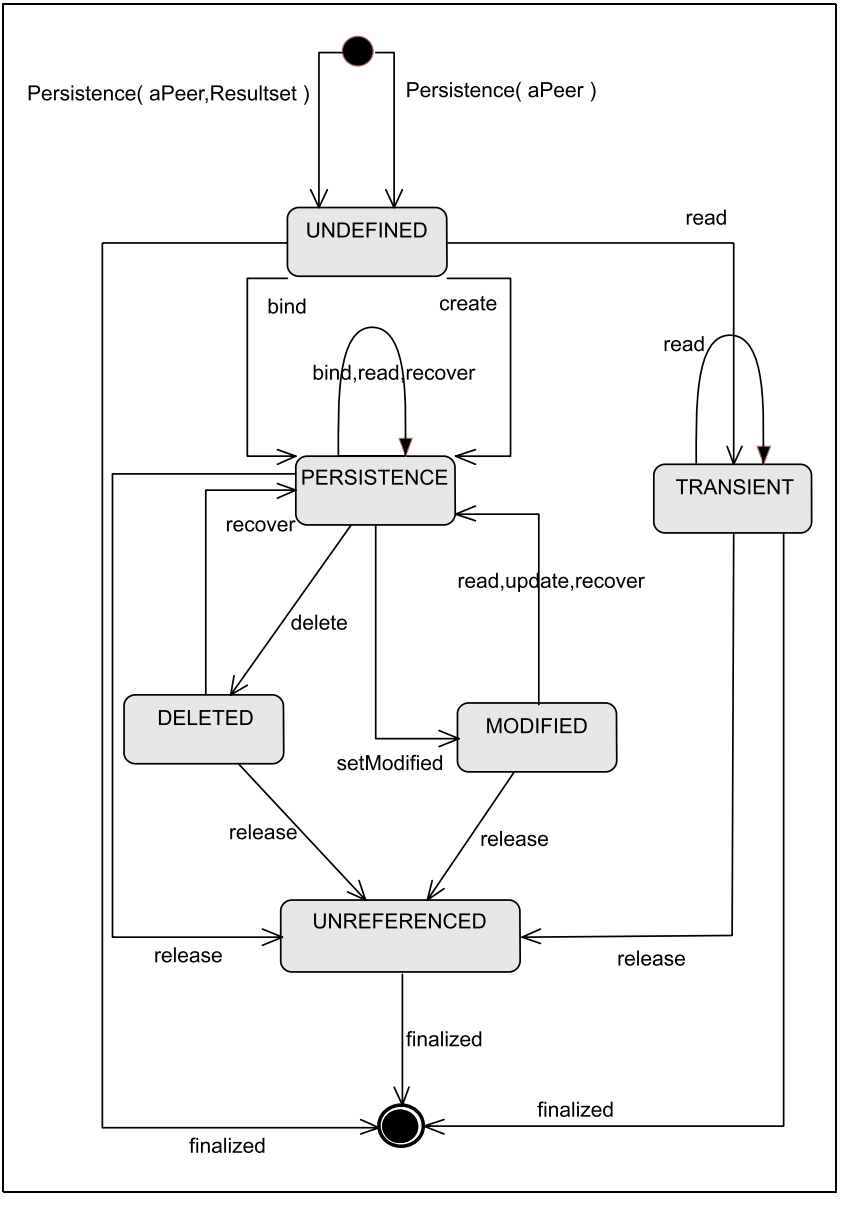


*Abbildung 9.19: Zustandsdiagram eines* `Persistence`

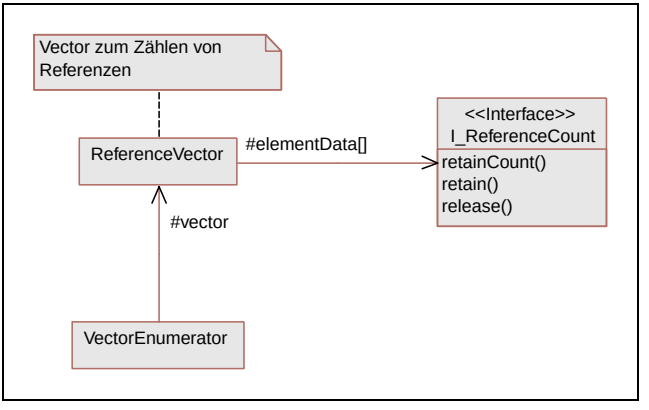


*Abbildung 9.20: Die Klasse* `ReferenceVector`

Dies geschieht im Persistence-Framework mit dem `ReferenceVector`. Dabei handelt es sich um eine Reimplementierung der Klasse `java.lang.Vector`. Jedes Objekt, das in einen `ReferenceVector` gelangt, muß die Schnittstelle `I_ReferenceCount` erfüllen. Sie ermöglicht einem Anwendungsobjekt die Referenzzählung. Dazu muß jedes Objekt, das ein Referenzzählerobjekt nutzt, vor dessen Nutzung ein `retain()` auf dieses Objekt ausüben (Listing 9.10). Wenn das Anwendungsobjekt das Referenzzählerobjekt nicht mehr benötigt, gibt es dieses mit `release()` einfach wieder frei.

```
package de.webapp.Framework.Utilities;
public interface I_ReferenceCount {
// Gebe Zählerstand bekannt
public int retainCount() ;
// Verwendung anmelden
public I_ReferenceCount retain() ;
// Freigabe einer Verwendung erteilen
public I_ReferenceCount release() ;
// Ausgabe für Log des Zählers
public String toString() ;
}
```

*Listing 9.10: Die Schnittstelle* `I_ReferenceCount`

Bei allen Manipulationen des `ReferenceVectors` wird genau dieses Verhalten umgesetzt. Da alle Resultate des `PersistenceRetrievers` in einem `ReferenceVector` gespeichert werden, braucht der Anwender sich über das Referenzzählen in der Regel keine Gedanken zu machen. Bei der Freigabe des `ReferenceVectors` erfolgt die automatische Freigabe aller Referenzzählerobjekte.

Die Klasse `Persistence` implementiert die Schnittstelle `I_ReferenceCount` (Listing 9.11 und Listing 9.12).

```
public I_ReferenceCount retain() {
   if( myObjectState != STATE_UNREFERENCED ) {
      if( myReferenceCount >= 0) {
         myReferenceCount++ ;
         if(Log.isLog())
            Log.log(" retain " + getObjectIdentifier() + " "
               + myReferenceCount + " " + myPeer.getType().getName() ) ;
      } else {
         if(Log.isLog())
            Log.log(" retain Fehler im Zähler" + getObjectIdentifier()
               + " " + myReferenceCount
               + " " + myPeer.getType().getName() ) ;
         throw new PersistenceRuntimeException (
            "Der Referenzzähler ist kleiner 0 bei dem Objekt "
            + getObjectIdentifier() + " " + myReferenceCount
            + " " + myPeer.getType().getName() ) ;
      }
   } else {
      if(Log.isLog())
         Log.log(" retain Fehler unreferenziert" + getObjectIdentifier()
            + " " + myReferenceCount + " "
            + myPeer.getType().getName() ) ;
         throw new PersistenceRuntimeException (
            "Die Applikationslogik stimmt nicht. "
            + " Ein unreferenziertes Objekt wurde wieder referenziert!"
            + getObjectIdentifier() + " " + myReferenceCount
            + " " + myPeer.getType().getName() ) ;
   }
   return this ;
}
```

*Listing 9.11: Die Realisierung der Methode* `Persistence.retain()`

In der Methode `retain()` von `Persistence` wird zuerst kontrolliert, ob das `Persistence` nicht schon aus der Anwendung entfernt worden ist. Falls ein `retain()` auf ein `Persistence`-Objekt im Zustand `UNREFERENCED` auftritt, oder der Zähler negativ ist, liegt ein schwerwiegender Kodierungsfehler vor. Falls alles normal verläuft, wird der Zähler `myReferenceCount` um Eins erhöht und darüber mittels `Log.log()` berichtet.

Die Freigabe wird natürlich genauso abgesichert wie die Verwendung. Jede Freigabe verringert den Zähler um Eins. Nach der letzten Freigabe wird das `Persistence` aktiv aus dem Cache seines `PersistencePeers` entfernt (Listing 9.12).

```
public I_ReferenceCount release() {
   if( myObjectState != STATE_UNREFERENCED ) {
      if(Log.isLog())
        Log.log(" release " + getObjectIdentifier()
          + " " + myReferenceCount + " " + myPeer.getType().getName());
      myReferenceCount--;
      if (myReferenceCount <= 0 && myPeer != null) {
         try {
```

```
            if(Log.isLog())
               Log.log(" Freigabe des Objekts "
                  + getObjectIdentifier()) ;
            myPeer.getCacher().uncache(this);
         } catch ( PersistenceException e ) {
            throw new PersistenceRuntimeException (
               "Austragen im Cache für das Objekt " + this
                + " ist fehlgeschlagen!",e) ;
         }
         myObjectState = STATE_UNREFERENCED;
         return null ;
      }
   } else {
      if(Log.isLog())
         Log.log( " release Fehler " + getObjectIdentifier()
            + " " + myReferenceCount + " "
            + myPeer.getType().getName());
         throw new PersistenceRuntimeException(
            "Die Applikationslogik stimmt nicht. "
            + " Ein unreferenziertes Objekt wurde nochmals freigeben! "
            + getObjectIdentifier() + " " + myReferenceCount + " "
            + myPeer.getType().getName() ) ;
   }
   return this ;
}
```

*Listing 9.12: Die Realisierung der Methode* `Persistence.release()`

Diese sicherlich gewöhnungsbedürftige Art und Weise mit Objekten umzugehen, ist im Framework gekapselt. Alle Referenzklassen des Frameworks kapseln ebenfalls das Zählen der Referenzen. In der Standardanwendung erfolgt die Verwendung also vollständig transparent.

## 9.5 Ein einfaches Beispiel

Nachdem die grundsätzliche Struktur des Stores erklärt ist, nun ein erstes Beispiel. Um ein persistentes Objekt nutzen zu können, müssen drei Schritte unternommen werden:

1. Anlegen einer Datenbank-Tabelle
2. Einfügen des Objekttyps in die Tabelle `TypeNumber`
3. Erstellen einer `Store`-Konfigurationsdatei

In unserem Beispiel wollen wir dem Objekt `Restaurant` zu Persistenz verhelfen. Das Objekt soll über zwei Attribute verfügen: `Name` und `Beschreibung`. Beide Attribute müssen also als Datensatzfelder in der zu generierenden Tabelle enthalten sein. Darüber hinaus müssen in jeder Tabelle, die von einem `Store` genutzt werden soll, die Attribute

ObjectIdentifier und ObjectVersion existieren (Abschnitt 9.2). Die folgende SQL-Anweisung generiert die benötigte Tabelle:

```
CREATE TABLE Restaurant (
   ObjectIdentifier CHAR(16) NOT NULL,
   ObjectVersion INTEGER,
   Name VARCAHR,
   Beschreibung VARCHAR,
   PRIMARY KEY (ObjectIdentifier)
);
```

Wie in Abschnitt 9.2 beschrieben, ist in der Objektidentität der Typ eines Objektes kodiert. Für die Tabelle Restaurant muß daher auf folgende Art ein Eintrag in der Tabelle TypeNumber erzeugt werden:

```
INSERT TypeNumber VALUES ('Restaurant', 10);
```

Damit ein Store auf dieses Persistence zugreifen kann, wird die Store-Konfigurationsdatei GenericRestaurant.store benötigt:

```
{
Name = "Restaurant" ;
JDBCConnection = {
   DriverClass = "solid.jdbc.SolidDriver" ;
   URLConnect = "jdbc:solid://localhost:1313" ;
   Properties = {
      user = "web" ;
      password = "app" ;
   }
} ;
Types = {
    Restaurant = {
      Type = {
        Name = "Restaurant" ;
        Class= "de.webapp.Framework.Persistence.PersistenceGeneric";
        Entity = "Restaurant";
        EntityAlias = "t0" ;
        Attributes = (
          "ObjectIdentifier",
          "ObjectVersion",
          { Name = "Name" ; Generic = "YES"; },
          { Name = "Beschreibung" ; Generic = "YES"; }
        ) ;
      }
    }
  }
}
```

*Listing 9.13: Die Konfiguration GenericRestaurant.store*

In ihr ist die Verbindung zu einer JDBC-Datenquelle und die Persistence-Typen des Stores beschrieben. Auf die Details dieser Konfiguration wird in den Abschnitten 9.6.3

und Anhang B.1 genauer eingegangen An dieser Stelle genügt es zu verstehen, daß alle Eigenschaften des `Stores` und seiner persistenten Objekte in dieser Konfiguration beschrieben werden. Eines sei jedoch angemerkt: Der Einfachheit halber wird hier auf die Kodierung einer speziellen Ableitung des `Persistence`-Objekts verzichtet und dafür das im Persistence-Framework enthaltene `PersistenceGeneric` verwendet. `PersistenceGeneric` ermöglicht es, Attribute über Universalmethoden anhand ihres Attributnamens zu setzen und zu lesen. Beispielsweise wird also statt einer Methode `getBeschreibung()` die Methode `getValue("Beschreibung")` verwendet. Ebenso wird statt `setBeschreibung("eine Beschreibung")` die Methode `setValue("Beschreibung", "eine Beschreibung")` benutzt.

Das Programm `GenericRestaurant` gibt je nach Modus entweder alle Restaurants der Datenbank aus oder fügt ein neues hinzu:

```
package de.webapp.Examples.Persistence.Simple;

import java.util.*;
import de.webapp.Framework.ConfigManager.* ;
import de.webapp.Framework.Persistence.* ;
import de.webapp.Framework.StoreFactory.StoreFactory ;
import de.webapp.Framework.Log.Log;
import de.webapp.Framework.Utilities.ReferenceVector;

public class GenericRestaurant{
protected I_Store myStore ;
// Zugang zum Store
public static void main(String args[])
   throws PersistenceException {
   GenericRestaurant theAccess = new GenericRestaurant ("RestaurantStore");
   Integer aCommand  = new Integer(args[0]) ;
   theAccess.action(aCommand.intValue(),args);
}
// Konstruktion des Stores für das Restaurant
public GenericRestaurant(String aStore)
   throws PersistenceException {
   Configuration theStore =
   ConfigManager.getConfigManager().getConfiguration(aStore);
   // Store Config
   myStore = StoreFactory.getInstance(theStore) ;
   if(myStore == null) {
      Log.log("Store konnte nicht angelegt werden") ;
      throw new PersistenceException("Store ist nicht zu konfigurieren" ) ;
   }
}
// Erzeuge ein neues Restaurant
protected void newRestaurant (String aName, String aBeschreibung)
   throws PersistenceException {
   I_Persistence newRestaurant = myStore.newPersistence("Restaurant") ;
   newRestaurant.setValue("Name",aName) ;
   newRestaurant.setValue("Beschreibung",aBeschreibung) ;
```

```
   newRestaurant.create() ;
}
// Suche alle Restaurants des Stores
protected void searchAllRestaurant()
   throws PersistenceException {
   ReferenceVector aVector =
      myStore.getRetriever("Restaurant").retrieve();
   for(Enumeration e=aVector.elements();e.hasMoreElements() ; ) {
      I_Persistence aObject=(I_Persistence)e.nextElement() ;
      Log.log("Restaurant/ Name - " + aObject.getValue("Name") +
        " - Beschreibung - " + aObject.getValue("Beschreibung") );
   }
}
// Ausführen eines Befehls auf einem Store
public void action(int aCommand, String[] args) {
   if(myStore != null) {
      // Umgebung für einen Store-Zugriff schaffen
      synchronized( myStore) {
         try {
             try {
                myStore.begin() ;
                switch(aCommand) {
                case 0:
                   searchAllRestaurant() ;
                   break ;
                case 1 :
                   newRestaurant(args[1],args[2]) ;
                   break ;
                default :
                   Log.log("Kommando " + aCommand + args
                       + " existiert nicht " ) ;
                }
                myStore.commit() ;
             } catch (PersistenceException e) {
                if(myStore != null)
                   myStore.rollback() ;
                throw e ;
             } finally {
                myStore.close() ;
             }
         } catch (Exception e ) {
            e.printStackTrace() ;
         }
      }
   }
}
} // Ende der Klasse
```

*Listing 9.14: Testumgebung für das Restaurant-Beispiel*

Im Programm wird zunächst eine Instanz von `GenericRestaurant` erzeugt. Dabei wird aus der Konfiguration `RestaurantStore` ein `Store` generiert. Anschließend wird eine Aktion mittels der Methode `action(int, String[])` ausgeführt (Listing 9.14). In der `action()`-Methode wird der `Store` durch `synchronized` vor der Ausführung durch einen anderen `Thread` geschützt und ein Transaktionsrahmen bereitgestellt. Jede Ausnahme führt direkt zum Abbruch (`rollback()`) der Transaktion. Die Aktion »0« zeigt alle vorhandenen Restaurants, und die Aktion »1« mit den Argumenten »Name« und »Beschreibung« erzeugt einen neues Restaurant.

Folgende Aufrufe realisieren die gewünschten Funktionen:

```
#Anzeige aller Restaurants
java -classpath /webapp/lib/webapp.jar;/solid/jdbc/SolidDriver.zip -DCFGROOT=/webapp/
etc/ de.webapp.Examples.Persistence.Restaurant.GenericRestaurant 0
```

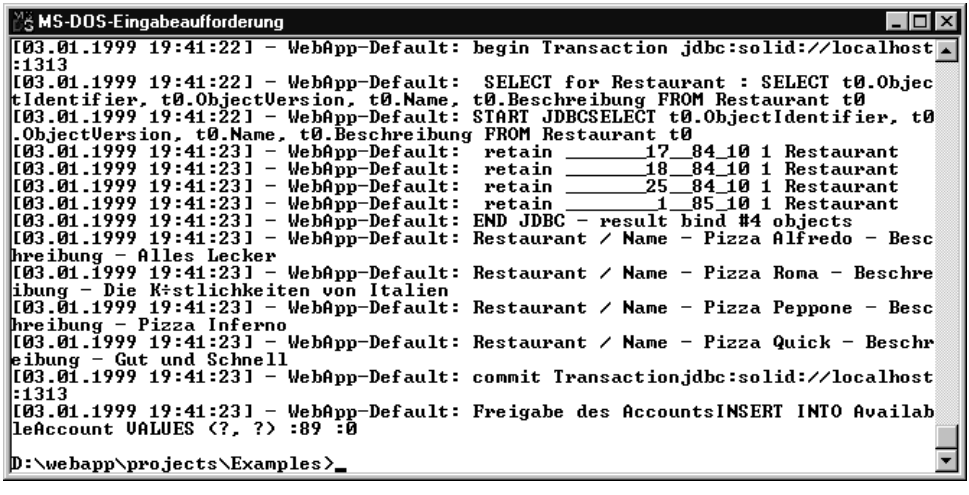


*Abbildung 9.21: Beispielausgabe aller Restaurants*

In Abbildung 9.21 sehen Sie die beispielhafte Ausgabe eines Programmlaufs, in dem alle Restaurants angezeigt werden.

1. Der `Store` wird initialisiert.
2. Die Transaktion beginnt, und die Anfrage nach alle Restaurants wird ausgeführt.
3. In der Datenbank befinden sich die Restaurants `Pizza Peppone` und `Pizza Roma`.
4. Die Transaktion wird erfolgreich abgeschlossen.
5. Das Programm terminiert und gibt das Konto zurück.

```
#Einfügen eines Restaurant
java -classpath /webapp/lib/webapp.jar;/solid/jdbc/SolidDriver.zip -DCFGROOT=/webapp/
etc/ de.webapp.Examples.Persistence.Restaurant.GenericRestaurant 1 'Pizza' 'Gut und
Schnell'
```

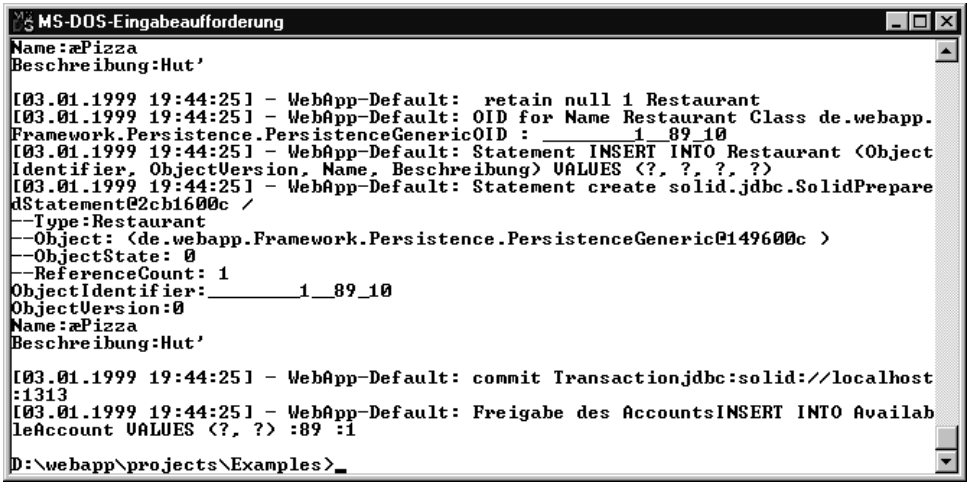


*Abbildung 9.22: Anlage eines neuen Restaurants*

Nachdem der `ObjectIdentifier` bestimmt ist, wird das neue `Restaurant`-Objekt `Pizza` mittels einer `INSERT`-Anweisung in die Datenbank geschrieben. Die Transaktion verläuft erfolgreich. Das nun um einen Zähler höhere Konto wird an die Datenbank zurückgegeben.

## 9.6 Persistente Objektnetze

Im Beispiel des vorherigen Abschnitts haben wir ein bewußt einfaches Objekt gewählt, das nicht mit anderen Objekten assoziiert war. In der Realität ist dies eher selten der Fall, weil in der Regel jedes Geschäftsobjekt (Business-Objekt) aus Attributen und Assoziationen besteht. Eine Assoziation bezüglich eines Geschäftsobjekts ist eine Beziehung oder Aggregation mit einem anderem Geschäftsobjekt. Es ist möglich, hierarchische Strukturen in einer Datenbank zu hinterlegen und sie als Objektnetz in der Anwendung zu nutzen. Die Assoziationen können zwischen unterschiedlichen Typen beziehungsweise Tabellen definiert werden. [Oestereich 1998, S.47ff]. In objektorientierten Programmiersprachen werden solche Beziehungen gewöhnlich über Datenstrukturen wie `Array`, `Vector`, `Hashtable`, `Collection` usw. realisiert oder als einfacher Verweis bei einer 1:1-Assoziation.

### 9.6.1 Persistenz durch Serialisierung

Wir wollen nun unser Restaurant-Beispiel um eine Assoziation erweitern. Dazu assoziieren wir das Restaurant mit allen Gerichten, die es anbietet. Im weiteren werden wir als Gegenbeispiel den Serialisierungsmechanismus des JDKs in der Nutzung dem des Persistence-Frameworks gegenüberstellen.

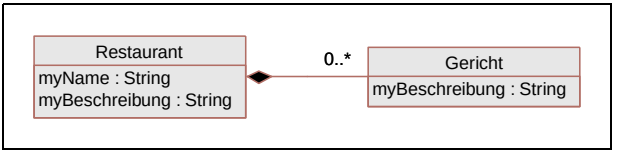


*Abbildung 9.23: Ein Restaurant hat viele schmackhafte Gerichte.*

Natürlich muß das Restaurant über Methoden verfügen, um Auskunft über seine Gerichte zu geben. Zudem ist es sinnvoll, ein Gericht von der Karte nehmen, beziehungsweise auf die Karte setzen zu können. Listing 9.15 zeigt eine nicht persistente Realisierung der beiden Klassen `Gericht` und `Restaurant`:

```
// ##########################################################################
// Datei: Restaurant.java
package de.webapp.Examples.Persistence.Restaurant.Serial ;

import java.util.Vector ;
import java.util.Enumeration ;
import java.io.Serializable  ;

// Restaurant
public class Restaurant implements Serializable
{
   protected String myName ;
   protected String myBeschreibung ;
   protected Vector myGericht ;
   public Restaurant () { myGericht = new Vector () ;}
   public  String getName() { return myName; }
   public void setName(String aName ) { myName = aName ; }
   public  String getBeschreibung() { return myBeschreibung; }
   public void setBeschreibung (String aBeschreibung ) {
       myBeschreibung = aBeschreibung ; }
   public void addToGericht(Gericht aGericht) {
      myGericht.addElement(aGericht) ;
   }
   public void removeFromGericht (Gericht aGericht) {
      myGericht.removeElement(aGericht) ;
   }
   public Enumeration getGericht () {
      return myGericht.elements();
   }
```

```
   public String toString() {
      return  "Name: " + myName + "\n" +
         "Beschreibung: " + myBeschreibung + "\n" +
         "Gerichte:\n" + myGericht + "\n" ;
   }
} // Ende der Klasse

// ##################################################################
// Datei: Gericht.java
package de.webapp.Examples.Persistence.Restaurant.Serial ;

import java.io.Serializable  ;

// Gericht
public class Gericht implements java.io.Serializable {
   protected String myBeschreibung ;
   public Gericht () {}
   public  String getBeschreibung() { return myBeschreibung; }
   public void setBeschreibung (String aBeschreibung ) {
       myBeschreibung = aBeschreibung ; }
   public String toString() {
      return  myBeschreibung + "\n" ;
   }
} // Ende der Klasse
```

*Listing 9.15: Die Klassen `Restaurant` und `Gericht`*

Für jedes `Restaurant`-Objekt ist es möglich, über die Methode `getGericht()` alle `Gerichte` zu erhalten. Ein neues Gericht wird dem Restaurant mit der Methode `addToGericht(Gericht)` hinzugefügt und ein nicht so schmackhaftes Gericht mit der Methode `removeFromGericht(Gericht)` wieder entfernt.

Eine simple Form der Persistenz verwirklicht die Java-Serialisierung. Über sie ist es möglich, Objekte, die die Schnittstelle `java.io.Serializable` implementieren, in einen Zeichenstrom umzuwandeln. Dieser Zeichenstrom läßt sich problemlos in einer Datei `Restaurant.serial` speichern und auch wieder aus ihr laden (Listing 9.16). Voraussetzung für das Speichern von Objektnetzen ist, daß alle beteiligten Objekte `Serializable` implementieren.

```
package de.webapp.Examples.Persistence.Restaurant.Serial ;

import java.util.Vector ;
import java.util.Enumeration ;
import java.io.* ;
public class SerialRest {
public static Gericht newGericht(String aBeschreibung){
   Gericht aGericht = new Gericht() ;
   aGericht.setBeschreibung(aBeschreibung) ;
   return aGericht ;
}
```

```
// Speicher oder Laden eines Restaurant-Objektnetzes
public static void main(String[] argc)
   throws Exception {
   Restaurant aRestaurant ;
   if(argc.length > 0 && argc[0].equals("0") ) {
      aRestaurant = new Restaurant() ;
      aRestaurant.setName("Pizza Peppone") ;
      aRestaurant.setBeschreibung("Schnell und Lecker") ;
      aRestaurant.addToGericht(newGericht("Pizza Vegetaria")) ;
      aRestaurant.addToGericht(newGericht("Pizza Inferno")) ;
      aRestaurant.addToGericht(newGericht("Pizza Hawaii")) ;

      FileOutputStream f = new FileOutputStream("Restaurant.serial");
      ObjectOutputStream s = new ObjectOutputStream(f);
      s.writeObject(aRestaurant);
      s.flush();
      s.close();
   } else {
      FileInputStream f = new FileInputStream("Restaurant.serial");
      ObjectInputStream s = new ObjectInputStream(f);
      aRestaurant = (Restaurant)s.readObject();
      s.close();
      System.out.println(aRestaurant);
   }
}
} // Ende der Klasse
```

*Listing 9.16: Serialisierung eines `Restaurants` samt seiner `Gerichte`*

Zuerst wird in der Methode `main(String[])` entschieden, ob ein neues Restaurant erzeugt oder ein bestehendes Restaurant angezeigt werden soll.

Beim Erzeugen eines neuen `Restaurants` werden Name und Beschreibung gesetzt. Jedes `Gericht` wird mit der Methode `newGericht(String)` erzeugt und dem neuen `Restaurant` hinzugefügt. Zu guter Letzt wird das nun entstandene Objektnetz eines `Restaurants` mit seinen `Gerichten` mittels eines `ObjectStreams` in die Datei `Restaurant.serial` geschrieben. Beim nächsten Aufruf ohne Parameter wird dieses Objektnetz komplett wieder gelesen und ausgeben.

Der sehr einfachen Handhabung von serialisierten Objekten steht ein nicht zu unterschätzender Nachteil gegenüber: Es nicht möglich, Teile von Objektnetzen erst zu laden, wenn sie wirklich benötigt werden.


*Abbildung 9.24: Ergebnis der Restaurantserialisierung*

### 9.6.2 Modellierung von Assoziationen durch Stellvertreter

Natürlich lassen sich auch mit dem Persistence-Framework Objektnetze in einer Datenbank abbilden. `Restaurant` und `Gericht` müssen dazu in zwei verschiedene Tabellen geschrieben werden. Was fehlt, ist die Verknüpfung. Die Lösung dieses Problems sind Referenz-Klassen. Sie dienen als Stellvertreter für Assoziationen und sorgen dafür, daß bei Bedarf die jeweiligen `Persistence`-Objekte aus der Datenbank geladen werden. Bei Assoziationen, die eine Aggregation bilden, werden sogar Änderungen eines abhängigen `Persistence`-Objekts (z.B. `Gericht`) einer Assoziation, bei der Speicherung des ursprünglichen `Persistence`-Objekts (z.B. `Restaurant`) mitgespeichert (Abbildung 9.27).

Auch der Zugriff ist elegant gelöst: Die 1:1-Referenz-Klasse `ToOneReference` greift nur auf die Datenbank zu, wenn sich das benötigte `Persistence` nicht im Cache des `PersistencePeers` befindet. Von der 1:N-Referenz-Klasse `ToManyReference` wird ein Zugriff nur durchgeführt, wenn noch kein `Persistence`-Objekt geladen wurden. Der Ladevorgang erfolgt automatisch bei der ersten Anfrage auf die Menge. Dies geschieht auch durch Hinzufügen oder Entfernen eines `Persistence` zu beziehungsweise aus der Assoziation. Wenn also ein bestehendes `Restaurant` ein neues `Gericht` auf die Karte setzen will, aber vorher noch keine `Gerichte` für dieses `Restaurant` genutzt wurden, werden die vorhanden `Gerichte` nachgeladen.

Da Objekte auf sehr unterschiedliche Weise miteinander in Beziehung stehen können, müssen wir eine Reihe von Schnittstellen definieren (Tabelle 9.6). Die Verflechtungen der Schnittstellen und ihrer Realisierungen zeigt Abbildung 9.25.

| Schnittstelle | Erklärung |
|---|---|
| `I_PersistenceReference` | Basisschnittstelle aller Assoziationen<br>Besorgen des Schlüsselwertes<br>Zugriff auf die Beschreibung der Referenz<br>Anfertigen einer Kopie |
| `I_PersistenceToOneReference` | 1:1-Assoziation |
| `I_PersistenceToManyReference` | 1:N-Assoziation |
| `I_PersistenceToManyModifiedReference` | 1:N-Assoziation, deren Änderung bei der Änderung des Persistence direkt mitberücksichtigt wird. |
| `I_PersistenceModify` | Markierung der Änderungsmethoden, `create()`, `update()` und `delete()` |

*Tabelle 9.6: Schnittstellen der `Reference`-Klassen des Persistence-Frameworks*

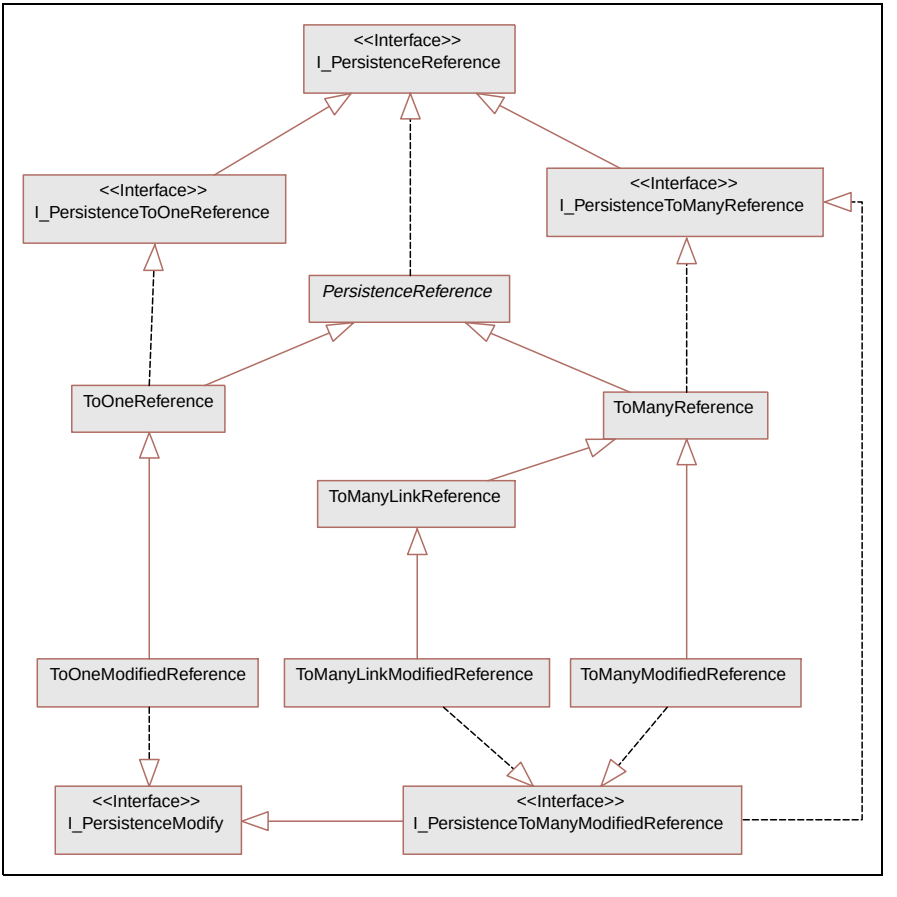


*Abbildung 9.25: Referenz-Klassen und Schnittstellen des Persistence-Frameworks*

Besonders zu beachten ist die Modellierung von 1:N-Assoziation einer N:M-Relation (`ToManyLinkReference`). Eine solche Beziehung herrscht beispielsweise zwischen Lieferanten und Artikeln. Ein Artikel kann häufig von mehreren Lieferanten geliefert werden. Genauso liefert ein Lieferant viele Artikel. Betrachtet man die Daten, interessieren jedoch meist nur die Artikel eines Lieferanten oder die Lieferanten eines Artikels. In einer relationalen Datenbank werden solche Assoziationen mit Hilfe von Zwischentabellen abgebildet.

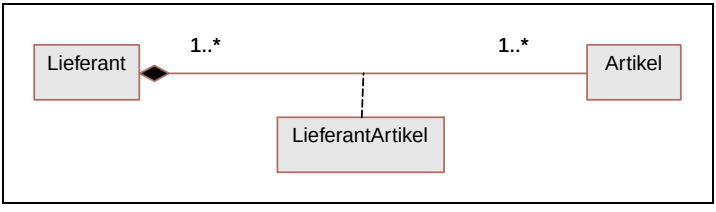


*Abbildung 9.26: Link-Klasse zwischen Lieferant und Artikel*

Im Persistence-Framework vermittelt die Klasse `ToManyLinkReference` die korrekte Sicht auf Artikel und Lieferanten (Abbildung 9.26) und versteckt die notwendigen, aufwendigen Qualifizierung vor dem Nutzer. Bei der Aufnahme eines neuen Artikels in die Assoziation wird vollautomatisch ein neuer Datensatz in der `LieferantArtikel`-Zwischentabelle mit den Schlüsseln zu `Lieferant` und `Artikel` angelegt.

Der einzige Unterschied zur `ToManyReference` ist die Qualifizierung der Datensätze. Generell ist über Referenz-Klassen das Modellieren von beliebig qualifizierten Assoziationen möglich. Für den Standardfall gibt es die Möglichkeit, eine statische Qualifizierungserweiterung anzugeben. Eine Sortierung `'Order By tO.Beschreibung'` oder eine statische Qualifikation `'AND Beschreibung like '%Pizza%'` kann mit der Methode `setQualifierExtension(String)` in der Schnittstelle `I_PersistenceToManyReference` gesetzt werden. Für die beiden Beispiele würde das bedeuten, daß die Gerichte eines Restaurants standardmäßig alphabetisch nach ihrem Namen geordnet wären, beziehungsweise eine Assoziation von Restaurants zu Gerichten, deren Name Pizza enthält, bestünde.

Für parametrisierte Qualifikation (Verallgemeinerung von `ToManyLinkReference`) bietet die Basisklasse `ToManyReference` die Methode `refValueQualifier()`. Die Methode muß dann bei speziellen qualifizierten Assoziationen entsprechend überschrieben werden. In der Konfiguration ist die Angabe einer speziellen Referenz-Klasse vorgesehen (Anhang B.1). Somit sind dem Einsatz verschiedenartiger Referenzen fast keine Grenzen gesetzt.

Mit Hilfe der Referenz-Klassen ist es möglich, problemlos durch ein Objektnetz zu navigieren. Die Objekte einer Assoziation werden erst erstellt und gebunden, wenn sie wirklich in der Anwendung benötigt werden. Der Speicherverbrauch einer Anwen-

dung verläuft dementsprechend kontrolliert und ressourcenschonend. Das Persistence-Framework erfüllt also die gängigen Anforderungen für den Einsatz innerhalb eines Servers.

### 9.6.3 Modellierung eines persistenten Objektnetzes

Als Beispiel wollen wir die Assoziation zwischen `Restaurant` und `Gericht` umsetzen. Im Vergleich zum Beispiel aus Abschnitt 9.5 kommen die Schritte 1. und 5. hinzu:

1. Analyse der Assoziationen und Auswahl der entsprechenden Referenz-Klassen
2. Anlegen der Tabellen
3. Einfügen der Objekttypen in die Tabelle `TypeNumber`
4. Erstellen einer `Store`-Konfigurationsdatei
5. Kodieren der `Persistence`-Klassen

Zunächst die Analyse: Die Attribute der Klassen sind der Einfachheit halber alle vom Typ `java.lang.String`. Jedes `Restaurant` besitzt seine `Gerichte` in Form einer `ToManyModifiedReference` (Abbildung 9.27). Wie oben bereits erwähnt, bewirkt dies, daß jede Änderung eines `Gerichts` auch beim Speichern eines `Restaurants` gespeichert wird. Jedes `Gericht` hat eine eindeutige Assoziation zu seinem `Restaurant` durch ein `ToOneReference`-Objekt.

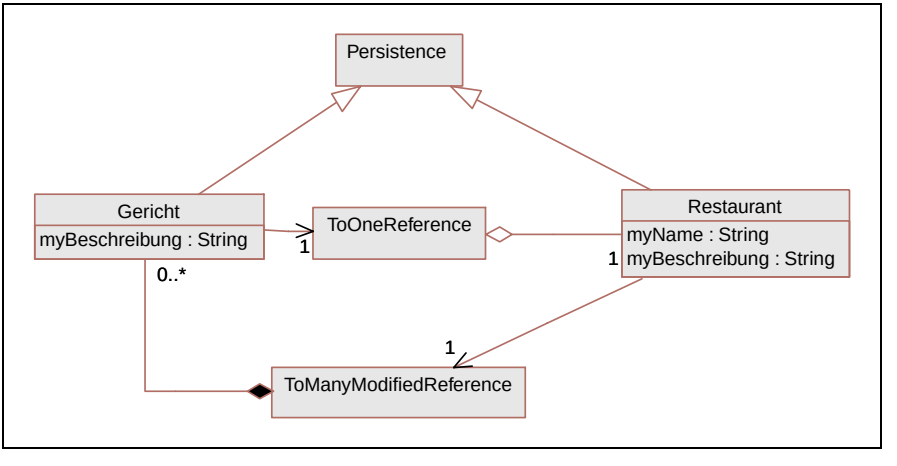


*Abbildung 9.27: Modellierung der Assoziationen mit Referenz-Klassen des Persistence-Frameworks*

Da die Tabelle `Restaurant` bereits in Abschnitt 9.5 angelegt wurde, müssen wir dies hier nicht erneut tun – bleibt das Anlegen der `Gerichte`-Tabelle mit folgender SQL-Anweisung:

```
CREATE TABLE Gericht (
   ObjectIdentifier CHAR(16) NOT NULL,
   ObjectVersion INTEGER,
   RestaurantID CHAR(16) NOT NULL,
   Beschreibung VARCHAR,
   PRIMARY KEY (ObjectIdentifier),
   FOREIGN KEY (RestaurantID) REFERENCES Restauarant (ObjectIdentifier)
);
```

Anschließend muß der Typ `Gericht` in die Tabelle `TypeNumber` eingefügt werden:

```
INSERT INTO TypeNumber ('Gericht',11) ;
```

Zur Definition der Eigenschaften von unseren `Persistence`-Objekten muß nun eine Konfiguration (Listing 9.17) des `Stores` erstellt werden. Deren Aufbau gliedert sich in drei Teile: Im ersten Teil der Konfigurationsdatei wird lediglich der Name des Stores festgelegt. Teil zwei definiert eine Datenbankverbindung – in diesem Fall für eine Solid-Datenbank. Danach werden in Teil drei die Typen `Restaurant` und `Gericht` definiert.

```
{
// Teil I
  Name = "Restaurant" ;
// Teil II
  JDBCConnection = {
   DriverClass = "solid.jdbc.SolidDriver" ;
   URLConnect = "jdbc: solid: //localhost: 1313" ;
   Properties = {
     user = "web" ;
     password = "app" ;
   }
   } ;
// Teil III
  Types = {
  Restaurant = {
    Type = {
     Name = "Restaurant" ;
     Class = "de.webapp.Examples.Persistence.Restaurant.Restaurant";
     Entity = "Restaurant";
     EntityAlias = "t0" ;
     Attributes = (
       "ObjectIdentifier",
       "ObjectVersion",
       "Name";
       "Beschreibung";
     ) ;
     Associations = (
       { Name = "Gericht" ; Type = "ToMany" ; Modified = "YES" ;
        ResultType = "Gericht" ; Key = "RestaurantID" ;
       }
     ) ;
```

```
    } ;
  } ;
  Gericht = {
    Type = {
     Name = "Gericht" ;
     Class = "de.webapp.Examples.Persistence.Restaurant.Gericht";
     Entity = "Gericht";
     EntityAlias = "t0" ;
     Attributes = (
       "ObjectIdentifier",
       "ObjectVersion",
       "RestaurantID";
       "Beschreibung";
     );
     Associations = (
       { Name = "Restaurant" ; Type = "ToOne" ;
        ResultType = "Restaurant" ; }
     ) ;
    }
  } ;
  }
} // Ende der Konfiguration
```

*Listing 9.17: Konfiguration des `Stores` für `Restaurant` und `Gericht`*

Im Gegensatz zu dem einfachen Beispiel aus Abschnitt 9.5 werden diesmal statt der generischen Klassen Ableitungen der Klasse `de.webapp.Framework.Persistence.Persistence` verwendet. Die Kodierung der erforderlichen Klassen ist etwas aufwendiger, kann aber prinzipiell auch von einem Werkzeug automatisch erledigt werden. Generell sind echte Ableitungen von `Persistence` den generischen Klassen aus Geschwindigkeitsgründen vorzuziehen, da generische Klassen ihre Attribute in Hashtabellen hinterlegen müssen, während echte Ableitungen direkt auf Instanzvariablen zugreifen können.

```
package de.webapp.Examples.Persistence.Restaurant ;

import de.webapp.Framework.Persistence.*;
import de.webapp.Framework.Utilities.ReferenceVector;

import java.sql.* ;

public class Restaurant extends Persistence {
   protected String myName ;
   protected String myBeschreibung ;
   protected I_PersistenceToManyModifiedReference myGerichtToManyRef ;

protected Restaurant() {}
// Konstruktion eines neuen Restaurants
public Restaurant(I_PersistencePeer aPeer)
   throws PersistenceException {
   super(aPeer) ;
}
```

```
// Konstruktion des Restaurants aus der Datenbank
public Restaurant(I_PersistencePeer aPeer,ResultSet rs)
   throws PersistenceException {
   super( aPeer, rs ) ;
}
// Binden der Attribute aus einer Ergebnismenge
protected int bind(ResultSet rs , int offset)
   throws PersistenceException,
   SQLException {
   offset = super.bind(rs,offset) ;
   myName          = (String) rs.getObject(++offset) ;
   myBeschreibung   = (String) rs.getObject(++offset) ;
   if(myGerichtToManyRef != null)
      myGerichtToManyRef.setValue(getObjectIdentifier()) ;
   return offset;
}
// Bei der Änderung der Identität muß dies
   mit den Assozationen abgeglichen werden.
public void setObjectIdentifier(String identifier) {
   super.setObjectIdentifier(identifier) ;
   if( myObjectState == STATE_UNDEFINED ||
       myObjectState == STATE_TRANSIENT ||
       myObjectState == STATE_UNREFERENCED ) {
      if(myGerichtToManyRef != null)
      myGerichtToManyRef.setValue(getObjectIdentifier()) ;
   }
}
// Zugriff auf die Attribute
public String getName() { return myName ; }
public void setName(String aValue) {
   myName = aValue ;
   setModified() ;
}
public String getBeschreibung() { return myBeschreibung ; }
public void setBeschreibung(String aValue) {
   myBeschreibung = aValue ;
   setModified() ;
}
// Zugriff auf die Assoziationen
public void addToGericht(I_Persistence aObject)
   throws PersistenceException {
   getGerichtReference().addToReference(aObject) ;
   setReferenceModified() ;
}

public void removeFromGericht(I_Persistence aObject)
   throws PersistenceException {
   getGerichtReference().removeFromReference(aObject) ;
   setReferenceModified() ;
}
```

```
public ReferenceVector getGericht()
   throws PersistenceException {
   return getGerichtReference().getObjects() ;
}

public I_PersistenceToManyModifiedReference getGerichtReference()
   throws PersistenceException {
   if(myGerichtToManyRef == null) {
      myGerichtToManyRef = (I_PersistenceToManyModifiedReference)
         getPeer().getType().associationWithName(
         "Gericht").newReference() ;
      myGerichtToManyRef.setValue(getObjectIdentifier());
   }
   return myGerichtToManyRef ;
}
// Synchronisation der Assoziationen
protected void afterCreate ()
   throws PersistenceException {
   super.afterCreate() ;
   if(myGerichtToManyRef != null)
      myGerichtToManyRef.create() ;
}

protected void afterUpdate ()
   throws PersistenceException {
   super.afterUpdate() ;
   if(myGerichtToManyRef != null)
      myGerichtToManyRef.update() ;
}
// Lösche alle Assoziationen, die MODIFIED sind.
protected void beforeDelete ()
   throws PersistenceException {
   getGericht();
      myGerichtToManyRef.delete() ;
   myGerichtToManyRef = null ;
   super.beforeDelete() ;
}
// Kopie des Objekts
public Object clone()
   throws CloneNotSupportedException {
   Restaurant theRestaurant = (Restaurant) super.clone() ;
   if(myGerichtToManyRef != null)
      theRestaurant.myGerichtToManyRef =
         (I_PersistenceToManyModifiedReference)
         myGerichtToManyRef.clone() ;
   return theRestaurant ;
}
} // Ende der Klasse
```

*Listing 9.18: Die Klasse `Restaurant` als Ableitung von `Persistence`*

Für jede Ableitung von Persistence müssen drei Konstruktoren realisiert werden. Obwohl die Konstruktoren öffentlich (public) sind, sollten sie niemals direkt außerhalb einer Framework-Ergänzung aufgerufen, sondern immer die Methode newPersistence(String) des Stores genutzt werden. Der Default-Konstruktor wird als protected erklärt, damit nicht aus Versehen eine Instanz eines Persistence ohne PersistencePeer entstehen kann. Der Konstruktor Restaurant(I_PersistencePeer) wird zum Erzeugen eines gänzlich neuen Persistence-Objekts benötigt. Im Gegensatz dazu dient der Konstruktor Restaurant(I_PersistencePeer, ResultSet) zur Erzeugung eines Persistence aus der Datenbank. Über die Methode bind(ResultSet, int) werden dabei die Werte aus dem ResultSet den entsprechenden Attributen zugewiesen.

An dieser Stelle gibt es zwei empfindliche Abhängigkeiten. Die erste ist, daß die Reihenfolge der Konfiguration mit der Kodierung exakt übereinstimmen muß, da das ResultSet positionsabhängig ausgelesen wird. Die zweite besteht darin, daß verschiedene JDBC-Treiber nicht immer die selben Abbildungen von Datenbankdatentypen zu Objektdatentypen haben. Bei einem Wechsel der Datenbank kann es daher in der Methode bind(ResultSet, int) zu schwerwiegenden Problemen kommen. Oft treten diese im Zusammenhang mit Geldbeträgen, ganzzahligen Werten und Zeiten auf. Ein anderes, vergleichsweise geringes Problem ist, daß unterschiedliche Treiber nicht die gleiche Anzahl an Nachkommastellen liefern (z.B. Solid zwei und MS-Access vier Nachkommastellen).

Die Methode setObjectIdentifier(String) muß überschrieben werden, damit bei einem neuem Objektnetz die automatisch bestimmte Objektidentität neuer Referenz-Objekte entsprechend gesetzt wird. Der Fremdschlüssel wird also selbständig propagiert. Für den Zugriff auf die Attribute sind die get- und set-Methoden gedacht. Ein direkter Attributzugriff ist durch die Deklaration ausgeschlossen (protected).

Jede Referenz mit der Schnittstelle I_ToManyReference macht die Realisierung der vier Methoden aus Tabelle 9.7 notwendig. Hier ist ein Manipulation der Werte möglich, da direkt auf die internen Datenstrukturen zugriffen wird.

| Methode | Erklärung |
|---|---|
| addTo<Name> | Hinzufügen eines Persistence-Objekts zur Referenz |
| removeFrom<Name> | Entfernen eines Persistence-Objekts aus der Referenz |
| get<Name> | Rückgabe aller Persistence-Objekte der Referenz |
| get<Name>Reference | Rückgabe der Referenz für spezielle Manipulationen |

*Tabelle 9.7: Methoden einer 1: N-Assoziation eines Persistence*

Im Fall des Restaurants ist dies für die Assoziation zu seinen Gerichten realisiert worden. Da es sich bei dieser Assoziation um eine Aggregation handelt, muß in den before<XX>-Methoden (Listing 9.19, syncModifiedReference()) der entsprechende Ab-

gleich mit der Assoziation vorgenommen werden. Wenn ein `Restaurant` gelöscht wird, verschwinden auch seine `Gerichte`. Je nach Anwendungslogik kann dies positiv oder negativ sein. Bei jeder Modifikation der Assoziationszugehörigkeit wird der Zustand des `Persistence`-Objekts mittels der Methode `setReferenceModified()` beeinflußt.

Zu guter Letzt, kann jedes `Persistence`-Objekt eine Kopie von sich mit der Methode `clone()` generieren. Die Referenzen des Frameworks generieren ebenfalls ein echte Kopie (siehe `ToManyReference.clone()`), allerdings ohne daß die `Persistence`-Objekte der Referenz mitkopiert werden. So werden z.B. die Gerichte eines Restaurants bei seiner Kopie nicht mitkopiert. Nur das Basisobjekt wird dupliziert.

```
package de.webapp.Examples.Persistence.Restaurant ;

import de.webapp.Framework.Persistence.*;
import java.sql.* ;

public class Gericht extends Persistence {
   protected String myBeschreibung ;
   protected String myRestaurantID ;
   protected ToOneReference myRestaurantToOneRef ;

protected Gericht() {}
// Anlage eines neuen Gerichts
public Gericht(I_PersistencePeer aPeer)
   throws PersistenceException {
   super(aPeer) ;
}
// Anlage eines Gerichts, das schon in der Datenbank vorhanden ist.
public Gericht(I_PersistencePeer aPeer,ResultSet rs)
   throws PersistenceException {
   super(aPeer,rs) ;
}
// Binden der Attribute aus der Ergebnismenge
public int bind(ResultSet rs , int offset)
   throws PersistenceException,
   SQLException {
   offset = super.bind(rs,offset) ;
   myRestaurantID = (String)  rs.getObject(++offset) ;
   myBeschreibung  = (String)  rs.getObject(++offset) ;

   if(myRestaurantToOneRef != null)
      myRestaurantToOneRef.setValue(myRestaurantID) ;
   return offset;
}
// Zugriff auf die Attribute
public String getBeschreibung() { return myBeschreibung ; }
public void setBeschreibung(String aValue) {
   myBeschreibung = aValue ;
   setModified() ;
}
```

```
public String getRestaurantID() { return myRestaurantID ; }
public void setRestaurantID(String aValue) {
   myRestaurantID = aValue ;
   setModified() ;
}
// Zugriff auf die Referenz
public I_Persistence getRestaurant()
   throws PersistenceException {
   return getRestaurantReference().getObject() ;
}

public void setRestaurant(I_Persistence aRestaurant)
   throws PersistenceException {
   getRestaurantReference().setObject(aRestaurant) ;
   myRestaurantID = (String) myRestaurantToOneRef.getValue() ;
   setModified() ;
}

public I_PersistenceToOneReference getRestaurantReference()
   throws PersistenceException {
   if(myRestaurantToOneRef == null) {
      myRestaurantToOneRef = (ToOneReference)
         getPeer().getType().associtionWithName(
            "Restaurant").newReference() ;
      myRestaurantToOneRef.setValue(myRestaurantID);
   }
   return myRestaurantToOneRef ;
}
// Bei einer Speicherung erfolgt ein Schlüsselabgleich für neue Objektnetze.
protected void syncModifiedReference()
   throws PersistenceException {
   if(myRestaurantID == null) {
      if(myRestaurantToOneRef != null) {
         I_Persistence aObject = myRestaurantToOneRef.getObject() ;
         if(aObject != null)
            myRestaurantID = (String)aObject.getObjectIdentifier() ;
      }
   }
}
// Synchronisation bei Speicherung
protected void beforeCreate ()
   throws PersistenceException {
   super.beforeCreate() ;
   syncModifiedReference() ;
}

protected void beforeUpdate ()
   throws PersistenceException {
   super.beforeUpdate() ;
   syncModifiedReference() ;
}
```

```
// Erstelle eine Kopie dieses Objekts
public Object clone()
   throws CloneNotSupportedException {
      Gericht theGericht = (Gericht) super.clone() ;
   if(myRestaurantToOneRef != null)
      theGericht.myRestaurantToOneRef =
         (ToOneReference) myRestaurantToOneRef.clone() ;
   return theGericht ;
}
} // Ende der Klasse
```

*Listing 9.19: Die Klasse `Gericht` als Ableitung von `Persistence`*

Die Klasse `Gericht` besitzt eine 1:1-Assoziation zu der Klasse `Restaurant`. Die Konstruktion, der Zugriff auf die Attribute und das Anfertigen einer Kopie geschieht analog zur `Restaurant`-Ableitung. Die Kodierung der Assoziation ist allerdings unterschiedlich.

| Methode | Erklärung |
|---|---|
| get<Name> | Rückgabe des `Persistence` |
| set<Name> | Setzen des `Persistence` |
| get<Name>Reference | Rückgabe der Referenz für spezielle Manipulationen |

*Tabelle 9.8: Methoden einer 1: 1-Assoziation*

Da nur ein einziges `Persistence-Objekt` von der `Reference` behandelt werden muß, werden direkte `set`- und `get`-Methoden verwendet. In dem `Persistence`-Objekt geht mit der Änderung der Referenz gleichzeitig eine Änderung des Fremdschlüssels (hier `RestaurantID`) einher. Daher muß auch der Zustand des `Persistence`-Objekts auf `MODIFIED` gesetzt werden. Die Zustandsänderung wird durch die Methode `setModified()` durchgeführt. Der Aufruf der Methode `syncModifiedReference()` in der `beforeCreate()`- und `beforeUpdate()`-Methode bewirkt eine Synchronisation des Fremdschlüssels mit dem Wert der Referenz. Die ist bedeutsam, wenn ein gänzlich neues Objektnetz gespeichert wird und die Objektidentität des referenzierten Objekts erst im Speichervorgang des Netzes bestimmt wurde.

Nach der etwas langwierigen Definition der benötigten `Persistence`-Klassen folgt nun die Anwendung.

```
package de.webapp.Examples.Persistence.Restaurant;

import java.util.*;
import de.webapp.Framework.ConfigManager.*;

import de.webapp.Framework.Persistence.* ;
import de.webapp.Framework.StoreFactory.StoreFactory ;
```

```
import de.webapp.Framework.Utilities.Log;
import de.webapp.Framework.Utilities.ReferenceVector;
import de.webapp.Framework.Utilities.I_WebAppException ;

public class RestaurantTest
{
   protected I_Store myStore ;
// Hole Store
public RestaurantTest(String aStore)
   throws Exception {
   ConfigManager.getConfigManager();
   Configuration theStore =
      ConfigManager.getConfigManager().getConfiguration(aStore);
   // Store Konfiguration
   myStore = StoreFactory.getInstance(theStore) ;
   if(myStore == null) {
      Log.log("Store konnte nicht angelegt werden") ;
      throw new Exception("Store ist nicht zu konfigurieren" ) ;
   }
}
// Erzeuge neues Gericht
protected Gericht newGericht(String aBeschreibung)
   throws PersistenceException {
   Gericht newGericht = (Gericht)myStore.newPersistence("Gericht") ;
   newGericht.setBeschreibung(aBeschreibung) ;
   return newGericht ;
}
// Erzeugen eines Peppone-Restaurants
protected void newRestaurantTest()
   throws PersistenceException {
   Restaurant newRestaurant =
      (Restaurant)myStore.newPersistence("Restaurant") ;
   newRestaurant.setName("Pizza Peppone") ;
   newRestaurant.setBeschreibung("Schnell und Lecker") ;
   newRestaurant.addToGericht(newGericht("Pizza Vegetaria")) ;
   newRestaurant.addToGericht(newGericht("Pizza Inferno")) ;
   newRestaurant.addToGericht(newGericht("Pizza Hawaii")) ;
   newRestaurant.create() ;
}
// Suche das Peppone-Restaurant
protected void searchAllRestaurantTest()
   throws PersistenceException {
  ReferenceVector aVector = myStore.getRetriever("Restaurant").retrieve(
     " WHERE t0.Name = 'Pizza Peppone'") ;
  for(Enumeration e=aVector.elements();e.hasMoreElements() ; ) {
     Restaurant aObject=(Restaurant)e.nextElement() ;
     Log.log("Restaurant / Name - " + aObject.getValue("Name")
        + " - Beschreibung - " + aObject.getValue("Beschreibung") );
     for(Enumeration
        e1=((ReferenceVector)aObject.getGericht()).elements();
```

```
        e1.hasMoreElements() ; ) {
        Gericht aGericht=(Gericht)e1.nextElement() ;
        Log.log((String)aGericht.getBeschreibung() );
     }
   }
}
// Kommandoverzweigung
public void action(int aCommand, String[] args) {
   if(myStore != null) {
      synchronized( myStore) {
         try {
            try {
               myStore.begin() ;
               switch(aCommand) {
               case 0 :
                  searchAllRestaurantTest() ;
                  break ;
               case 1 :
                  newRestaurantTest() ;
                  break ;
               default :
                  Log.log("Kommando " + aCommand + args
                     + " existiert nicht " ) ;
               }
               myStore.commit() ;
            } catch (Exception e) {
               if(myStore != null)
                  myStore.rollback() ;
               throw e ;
            } finally {
               myStore.close() ;
         }
      } catch (Throwable e ) {
         Throwable aThrowable = e ;
         while (aThrowable != null) {
            aThrowable.printStackTrace();
            if(aThrowable instanceof I_WebAppException) {
               aThrowable  =
                    ((I_WebAppException)aThrowable).getThrowable() ;
            } else
               aThrowable = null ;
         }
      }
    }
  }
}
// Start der Aufgabe
public static void main(String args[])
   throws Exception {
```

```
   RestaurantTest theRestaurant = new RestaurantTest("RestaurantStore");
   Integer aCommand  = new Integer(args[0]) ;
   theRestaurant.action(aCommand.intValue(),args);
}
} // Ende der Klasse
```

*Listing 9.20: Zugriff auf ein `Restaurant` mittels des Persistence-Frameworks*

Der Start der kleinen Testumgebung `RestaurantTest` (Listing 9.20) erzeugt das Restaurant Peppone mit den entsprechenden `Gerichten`, beziehungsweise zeigt es an. Im Konstruktor wird der `Store` konfiguriert. Die Methode `action(int, String[])` vermittelt die jeweils gewünschte Funktion. In ihr wird der `Store` synchronisiert und der notwendige Transaktionsrahmen geschaffen. Die Methode `searchAllRestaurantTest()` sucht das Restaurant Peppone mit der Unterstützung eines `PersistenceRetrievers` aus der Datenbank und schreibt es ins Standard-Log. Die Methode `newRestaurant(String, String)` erzeugt ein neues Peppone-Restaurant und fügt mittels `newGericht(String)` einige `Gerichte` hinzu. Die Speicherung des Restaurants erfolgt über eine einzige `create()`-Anweisung an das neue `Restaurant`-Objekt.

## 9.7 Persistence im Einsatz

Ein Persistence-Framework anstelle von JDBC einzusetzen, bringt in der Regel eine ganze Reihe von Vorteilen mit sich. So ist es möglich, die Datenbank vollständig objektorientiert zu kapseln. Lediglich ein Spezialist im Entwicklerteam muß noch über sehr gute SQL-Kenntnisse verfügen. Ist ein Persistence-Framework sinnvoll konfiguriert, stehen die Chancen gut, weitgehend ohne SQL auszukommen. Das wiederum erhöht die Portierbarkeit, Erweiterbarkeit und Wartbarkeit von Anwendungen erheblich.

Im folgenden sind die wesentlichen Eigenschaften des Persistence-Frameworks nochmals kurz aufgelistet:

- dienstorientierte Architektur
- Alle Dienste sind konfigurierbar und durch Schnittstellen definiert.
- Kapselung von JDBC für den Zugriff auf Tabellen
- Konfigurierbarkeit der Objekt-relationalen Abbildung
- Die Nutzung von SQL beschränkt sich auf die Formulierung von Qualifizierungen für spezielle Anfragen.
- Erzeugen von Objektnetzen
- Navigation über Assoziationen ohne SQL-Kenntnisse
- generische Objekte für schnelles Prototyping

- `null` darf als Wert verwendet werden.
- Unifikation – Eindeutigkeit des `Persistence` in einem `Store`
- Geschwindigkeitsvorteil durch Einsatz von Caches
- mehrere unabhängige Stores in einer Java-VM
- Verfügbarkeit einer Laufzeit-Typbeschreibung
- Transaktionsbehandlung

# 10 Generierung von dynamischen HTML-Seiten mit Servlets

Nachdem wir nun die Grundlagen für webbasierte Anwendungen mit SMI (Kapitel 8) und dem Persistence-Framework (Kapitel 9) gelegt haben, bleibt noch zu klären, wie die Aufbereitung von dynamischen Inhalten auf dem Server erreicht werden kann. In den letzten zwei Jahren hat es eine Flut von unterschiedlichen Ansätzen für die dynamische Generierung von HTML-Seiten via Servlets gegeben. Angestachelt durch Microsofts Active-Server-Pages (ASP) steht der Java-Fraktion mit den JavaServer-Pages (JSP) nun das portable Gegenstück zur Verfügung [JSP 99]. Die JSP können in jedem Web-Server, der eine Servlet-Engine besitzt, eingesetzt werden. Das Bereitstellen bestimmter Informationen auf unterschiedlichen Betriebssystemen und Server-Umgebungen ist also sehr leicht geworden. Die JSP sind außerdem eine elegante Lösung zur Formatierung der Inhalte. Zur Zeit wird Java als Skript-Sprache benutzt, Java-Beans-Komponenten dienen dabei zur Funktionskapselung.

Dieses Kapitel wird am Ende auf die Programmierung mit JSP eingehen, zuvor sollen jedoch die vielen weiteren Ansätze beschrieben und eingeordnet werden.

## 10.1 Serverseitige Generierung von HTML-Seiten

Kurz nach der Veröffentlichung des Servlet-API 1.0 (1997) wurde HTML von Hewlett-Packard (Anders Kristensen) in Form von Java-Klassen zur Verfügung gestellt. Für Programmierer ist diese Art der HTML-Kodierung eine wahre Freude. Sie müssen sich endlich nicht mehr mit den Parametern einiger HTML-Befehle wie `FORM` oder `TABLE` auseinandersetzen, sondern können wohlgeformte Objekte nutzen. Zum schnellen Bereitstellen bestimmter, festgelegter Darstellungen ist diese Vorgehensweise unübertroffen. Die Präsentation von Inhalten ist jedoch selten die Aufgabe der Programmierer, sondern die der Designer. Diese fordern von den Programmierern Unterstützung und möchten nur begrenzt in die Tiefen einer mächtigen Programmiersprache wie Java abtauchen.

Aus dieser Erkenntnis resultieren mehrere sehr verschiedene Ansätze, die Aufgabe mit speziell zugeschnittenen Skript-Sprachen zu lösen. Einige ermöglichen den Zugriff auf

Java-Objekte, andere bieten einen abgeschlossenen, eigenständigen Sprachumfang, und wieder andere beschränken sich auf das simple Ersetzen von Markierungen durch Inhalte. Welche Lösung in einer konkreten Situation die beste ist, hängt sehr stark vom Projektinhalt und der Zusammensetzung des Entwickler-Teams ab.

Im folgenden werden die verschiedenen Ansätze kurz vorgestellt. Für die Beispiele im dritten Teil des Buches haben wir uns für JSP entschieden, da dieser Weg von Sun propagiert und sich wahrscheinlich durchsetzen wird.

### 10.1.1 HTML-Programmierung mit Java

In jedem Projekt gibt es die Notwendigkeit, Informationen aus einer Datenquelle zu formatieren. Manchmal möchte man zudem einfache Eingabedialoge automatisch generieren. Dafür drängt sich die Kapselung einer HTML-Seite in einer konfigurierbaren Klasse geradezu auf. Die einzelnen HTML-Elemente werden dazu ebenfalls in Klassen gekapselt und zum Seiten-Objekt einfach hinzugefügt.

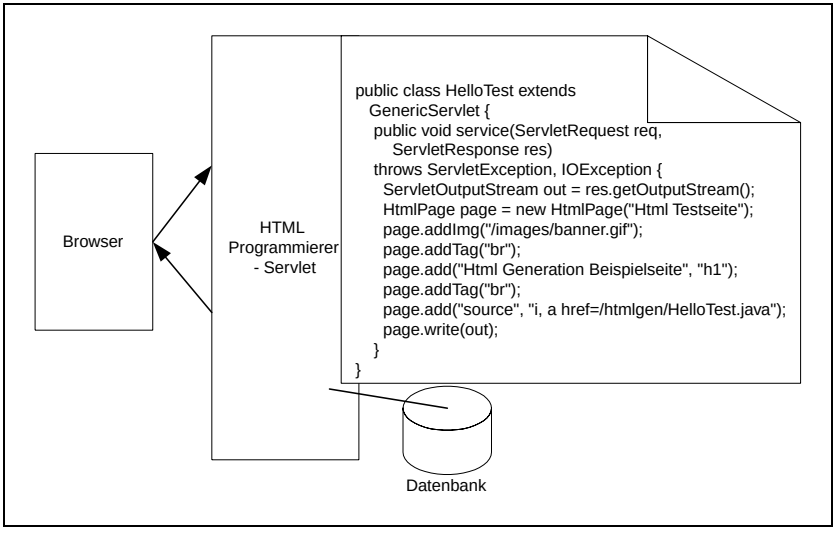


*Abbildung 10.1: Programmierung von HTML in Java*

Jede so erstellte dynamische HTML-Seite muß von einem Servlet ausgegeben werden. In Abbildung 10.1 ist eine entsprechende Realisierung zu sehen. Das Servlet muß dabei vorher übersetzt und bei der Servlet-Engine registriert werden. Das Beispiel entstammt dem Nexus-Paket (Tabelle 10.1).

Ein sehr gelungenes HTML-Paket liefert auch der frei verfügbare Jetty-Web-Server. Die Kapselung aller HTML-Elemente macht das Programmieren leicht und übersichtlich. Die Klasse `Page` ermöglicht eine hohe Abstraktion einer HTML-Seite. Kopf und Rumpf

einer HTML-Seite sind leicht erweiterbar. Das entsprechende Methodenangebot `writeBodyTag(Writer)`, `writeElements(Writer)`, `writeHtmlEnd(Writer)` und `writeHtmlHead(Writer)` bietet einen ausreichenden Spielraum. Die vorhandene `PageFactory` läßt zudem den Zugriff auf mehrere vorkonfigurierte HTML-Seiten zu. Somit lassen sich Module von Seiten entwickeln und ein entsprechend generischer Zugang über ein `HTMLDisplayServlet` schaffen.

| Paket | Verweis | Bemerkung |
|---|---|---|
| HTMLKona | `http://www.weblogic.com/docs/examples/htmlkona/htmlexamples.html` | Gut eingebunden in das Web-Server-Gesamtkonzept von WebLogic. Integration weiterer Produkte z. B. JDBCKona zum Zugriff auf Datenbanken. |
| Jetty | `http://www.mortbay.com/` | Die Servlet-Engine enthält ein Paket mit Java-Klassen, um HTML in Java programmieren zu können. Der Quellcode ist sehr ansehnlich. |
| Nexus | `http://www-uk.hpl.hp.com/people/ak/java/` | Vom Nexus-Web-Server unabhängige Verwendung möglich. Sehr einfache Realisierung. |
| SDSU | `http://www.eli.sdsu.edu/java-SDSU/` | Sehr einfache Kapselung. GNU-Lizenz. |
| ECS | `http://java.apache.org/ecs/` | Das Element Construction Set kann alle HTML-4.0-Tags verwenden und bietet eine Unterstützung von XML-Tags. Die Flexibilität und Geschwindigkeit der Objekte ist beeindruckend. |

*Tabelle 10.1: Pakete für die Programmierung von HTML*

Die weiteren Pakete sind sehr unterschiedlich. HTMLKona ist das einzige echte Produkt. Es ist mit dem BEA-WebLogic-Server in vielen Projekten im Einsatz. Bei den anderen Ansätzen, Nexus, ECS, SDSU und Jetty, ist der Quellcode erhältlich. Somit besteht die Möglichkeit, eigene Anpassungen vorzunehmen (Tabelle 10.1). Das ECS Element Construction Set ist sehr flexibel und bietet dem Programmierer ein weitreichendes und konzeptuell sehr gut durchdachtes Paket. Dem kommerziellen Einsatz steht nichts im Wege.

### 10.1.2 HTML-Generierung mit mächtigen Skript-Sprachen

Einen ganz anderen Weg beschreiten die Produkte zur Anbindung eigener Skript-Sprachen an Servlets (Abbildung 10.2). Hier werden in der Regel vollständige und eigenständige Sprachen genutzt. Über ein Servlet wird dabei die Anfrage an den Interpreter der Sprache vermittelt. Dieser liest dann die entsprechende Skript-Datei und bereitet die Antwort auf.

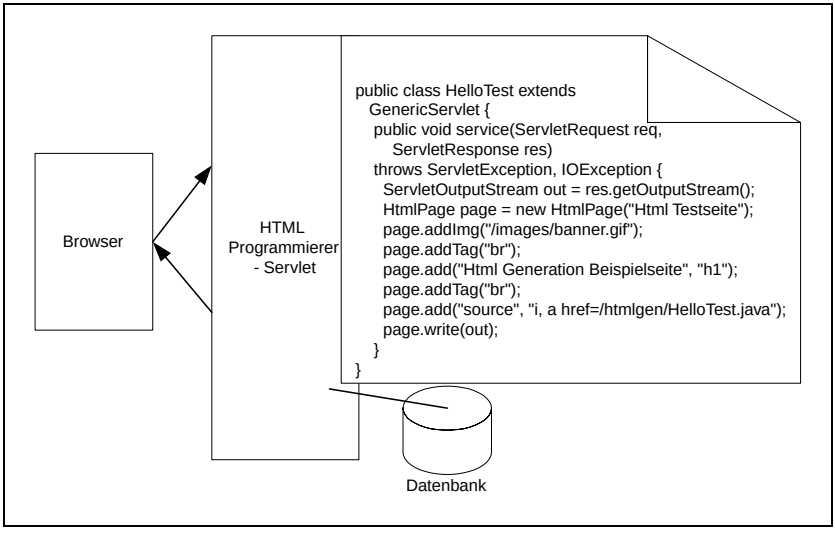


*Abbildung 10.2: Programmierung von HTML in Java*

Zwei unterschiedliche Ansätze dieser Kategorie sind Aspy mit JPython und EcmaScript Page (ESP) (Tabelle 10.2). Beiden gemeinsam ist die Verwendung einer vollständigen Programmiersprache neben HTML und Java.

| Paket | Verweis | Bemerkung |
|---|---|---|
| Aspy | `http://www.dstc.edu.au/aspy/src/` | Verwendet JPython |
| EcmaScript Pages (ESP) | `http://www.mindspring.com/~rrocha` | Basiert auf Fesi, einem Java-basierten EcmaScript-Interpreter |

*Tabelle 10.2: Pakete mit eigenen Skriptsprachen*

Sicherlich kommt die ESP-Lösung JavaScript-Programmierern entgegen. Sie bietet jedoch keinen Zugriff auf Java-Objekte. Das sinnvolle Verwenden von in Java kodierter Geschäftslogik ist somit unmöglich. Diese Skript-Erweiterungen unterstützten die Benutzung einer klaren Architektur, wie wir sie in diesem Buch vorschlagen, nur wenig. Schon interessanter sind die im folgenden Abschnitt behandelten Skript-Sprachen, die Java-Objekte integrieren.

### 10.1.3 HTML-Generierung mit einfachen Skript-Sprachen

Anstatt eine vollständig eigene Sprache zu definieren, ermöglichen einige Ansätze den direkten Zugriff auf Java-Objekte. Dies hat den Vorteil, daß beliebig komplexe Vorgänge von Programmierern in Objekte gekapselt werden können, während Web-Designer nur auf die einfachen Methoden dieser Objekte zugreifen. Die Zielgruppe

sind dementsprechend Web-Designer mit JavaScript-Kenntnissen. Die Sprachen bieten neben der Definition von lokalen Variablen Befehle zur Fallunterscheidung (`IF`) und Iteration (`FOR`).

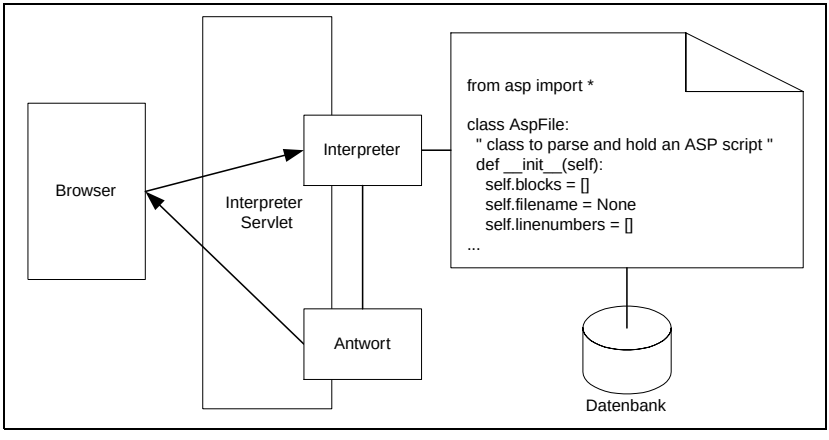


*Abbildung 10.3: Generierung der HTML Seiten durch einen Skript-Interpreter*

Die Argumente der Anfrage werden entweder mit festen Markierungen (`<%REQUEST%>` oder `<%RESPONSE%>`) oder globalen Variablen (`$REQUEST` oder `$RESPONSE`) zur Verfügung gestellt. Da der direkte Zugriff auf Java-Objekte realisiert ist, können beliebige Ergebnisse angefordert und anschließend formatiert werden. Es läßt sich eine Trennung zwischen Ermittlung von Daten und ihrer Aufbereitung erreichen (siehe `$Result` in Abbildung 10.3). Auch funktionale Objekte wie ein Mail-API oder eine Protokollausgabe lassen sich direkt ansprechen. Die Aufgabenteilung in einem Team läßt sich elegant vollziehen, da die Sprachen ein paralleles Arbeiten fördern. Die Programmierer können ihrem Anspruch der Kapselung von Funktionalitäten gerecht werden, und die Designer behalten die Freiheit der Gestaltung.

| Paket | Verweis | Bemerkung |
|---|---|---|
| objectHTML | `http://www.factum-gmbh.de/` | Zugriff auf Java-Objekte (Attribute, Konstanten und Methoden-Zugriff). Erweiterung des Reflect-API, um wirklich auf alle Methoden zugreifen zu können. Optimierter Interpreter mit Parsetree-Cache. Konzept der Formatierer zur programmgesteuerten Ausgabe je nach Objekttyp. Unabhängig vom Servlet-API [Roßbach/ Schreiber 98c]. |

*Tabelle 10.3: Pakete für Java-basierte Skriptsprachen*

| Paket | Verweis | Bemerkung |
|---|---|---|
| Otembo | http://www.meangene.com/otembo/ | Skriptsprache basierend auf einem universellen Tagging-Konzept. |
| WebMacro | http://www.webmacro.org/ | Makro-Sprache, die mit der Hilfe eines speziellen Servlets interpretiert wird. |
| FreeMaker | http://freemarker.org/ | Eine sehr gelungene Makro-Sprache, die unter GPL steht. Die Realisierung nutzt die Java-Collection-Klassen effektiv. |
| InstantOnlineBasic | http://www.gefionsoftware.com/ | Nutzt Server-Side-Includes mit speziellen Servlets |

*Tabelle 10.3: Pakete für Java-basierte Skriptsprachen (Fortsetzung)*

Die Mächtigkeit der Lösungen ist sehr unterschiedlich (Tabelle 10.3). Alle Produkte beherrschen den Zugriff auf Java-Objekte. Die vollständige Integration von Java ist durch Mängel des `java.lang.reflect`-API nicht einfach möglich. Der Zugriff auf bestimmte Signaturen funktioniert nur, wenn der Typ eines Arguments exakt vorliegt. Wenn nur eine Unterklasse angegeben ist, fällt es schwer, die Methode zu bestimmen. Gerade hier zeigt sich eine Stärke von objectHTML und die Schwäche vieler anderer Ansätze. Ein gelungener Ansatz von objectHTML sind außerdem die Formatierer. Für jeden Typ, der in einer Ausgabeanweisung (`<%print ... %>`) vorkommt, kann eine Funktion mit Argumenten zur Formatierung angeben werden. Wenn keine spezielle Formatangabe gemacht wurde, wird die Methode `toString()` angewandt und die resultierende Zeichenkette ausgegeben.

### 10.1.4 HTML-Erzeugung durch Generieren von Java-Klassen

Die Krönung zur Generierung von dynamischen HTML-Seiten ist die Verwendung von Java als Skriptsprache. Begonnen hat diese Entwicklung mit der Page Compilation des Java Webservers 1.1 Ende 1997. Die nächste Generation sind die JavaServer-Pages. Als besondere Ergänzung ist für die JSP die Unterstützung von JavaBean-Komponenten und erweiterbaren Tags ermöglicht worden. (`http://java.sun.com/products/jsp`).

Die Kodierung einer JSP erfolgt mit einer Mischung aus HTML und Java. Einige zusätzliche Markierungen reichen aus, um HTML und Java-Code voneinander zu trennen. Auf die genaue Syntax gehen wir im Abschnitt 10.2.1 ein. Generell gilt, daß aus jeder JSP ein Servlet generiert wird (Abbildung 10.4).

Auf die Parameter des Servlets, die Anfrage (`request`) und die Antwort (`response`), können wir direkt im Skript zugreifen. Nach einmaligem Übersetzen des Skripts steht die Klasse für alle folgenden Zugriffe zur Verfügung. Es ist klar, daß sich hieraus ein deutlicher Geschwindigkeitsvorteil gegenüber den rein interpretativen Skript-Lösungen aus Abschnitt 10.1.3 ergibt. Tabelle 10.4 zeigt die Vielfalt der entstandenen Lösungen.

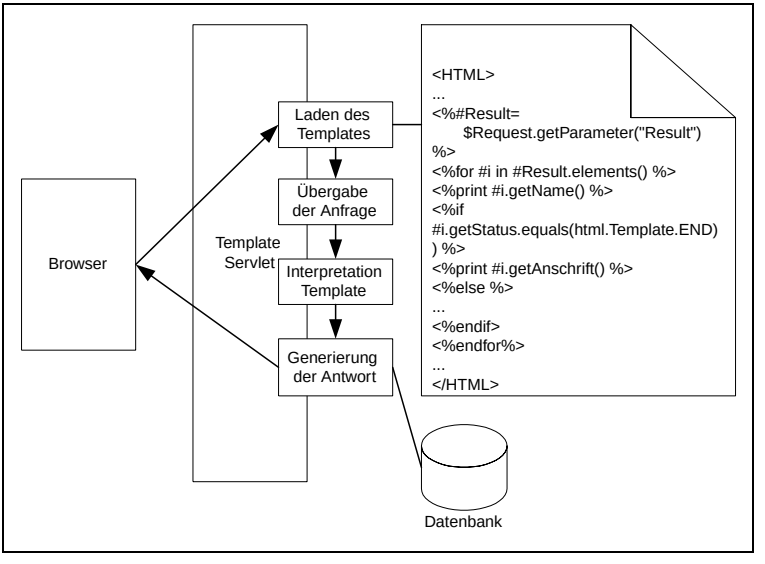


*Abbildung 10.4: Interpretation eines Templates mit Zugriff auf Java-Objekte*

| Paket | Verweis | Bemerkung |
|---|---|---|
| JSP – Java Server Pages | http://java.sun.com/products/jsp/ | Referenz-Implementierung der JSP1.1<br>Java als Skript-Sprache.<br>Wird kompiliert und in ein Servlet verwandelt. |
| Tomcat | http://jakarta.apache.com/ | Vereinigt die Sun-Referenz-Implementierung des Servlet-API 2.2, JSP1.1 und Apache JServ 1.0 zu einer einheitlichen Version. |
| JRun | http://www.allaire.com/ | Realisiert ab JRun 3.0 auch Servlet API 2.2 und JSP 1.1 |
| ServletExec | http://www.unify.com | Realisiert ab der Version 3.0 auch JSP 1.1 und Servlet API 2.2 |
| Bea | http://www.beasys.com | Vollständige JSP-1.1- und Servlet-API-2.2-Realisierung mit verteiltem Servlet-Container |
| WebSphere | http://www.software.ibm.com/webservers/ | Realisiert die JSP1.0 |

*Tabelle 10.4: Pakete zur Generierung von Java-Servlets*

| Paket | Verweis | Bemerkung |
|---|---|---|
| GNUJSP 0.9.10 | http://www.klomp.org/gnujsp/09x/ | Realisierung der JSP nach Spezifikation 0.91. Ist die wohl am meisten genutzte Realisierung in der Welt der freien Software. Auf der Site werden außerdem alle Informationen und Patches rund um GNUJSP veröffentlicht. |
| GNUJSP 1.0 XML | http://www.euronet.nl/~pauls/java/gnujsp/ | Realisierung der JSP 1.0 über ein XML/DOM-Modell (Document Object Modell) |
| GNUJSP 1.0 | http://www.klomp.org/gnujsp/ | Eine funktionierende JSP-1.0-Realisierung mit vielen Bug-Fixes in den Snapshot-Releases. |
| RocketJSP | http://www.cyberway.com.sg/~munwai/ | Teilweise konform mit der JSP-1.0-Spezifikation |
| zJSP | http://www.zachary.com/creemer/zjsp.html | Realisiert die 0.91-Spezifikation |
| PolyJsp | http://www.plentix.org/ | Realisiert die 0.92-Spezifikation |
| SJSP | http://web.telecom.cz/sator/jsp/index.html | Realisiert die 0.92-Spezifikation |
| JST – JRun Scripting Toolkit | http://www.livesoftware.com/products/jst/index.html | Erweiterung der JSP. Gebunden an JRun-Engine |
| ServletFactory (ServletBuilder) | http://www.earlymorning.com | Einfacher Servlet-Generator (ServletBuilder) auf Basis der JSP-Syntax, allerdings ohne Bean-Tag. Integriert im freien Java Web Server ServletFactory. |
| GSP | http://www.bitmechanic.com/projects/gsp/ | Basiert auf dem Vorgänger der JSP, der Page-Compilation. Eigene Erweiterungen. |
| Servc | http://www.geocities.com/SiliconValley/Pines/3185 | Generierung von Servlets durch Makros. |
| WeAF Java Web Application Framework | http://www.weaf.com/ | Komplette Einbindung von Java. Spezielle Tags für JDBC und SQL. |

*Tabelle 10.4: Pakete zur Generierung von Java-Servlets (Fortsetzung)*

Natürlich gibt es neben der Referenzlösung von Sun andere Produkte. Herausragend ist sicherlich die freie Realisierung GNUJSP von Vincent Partington, die uns während der Entwicklung des WebApp-Frameworks begleitet hat. Auch die Firma Allaire (ehemals Live Software) hat mit dem JRun Scripting Toolkit (JST) eine elegante Erweiterung erstellt. Leider ist JST an die JRun-Servlet-Engine gebunden. Alle JSP-Realisierungen sollen in dem Jakarta-Projekt unter der Führung der Apache-Gruppe und mit Hilfe

von Sun (James Duncan Davidson) zu einer schnellen und leistungsfähigen Referenz-Implementierung vereinigt werden. Der Vorteil dieser Initiative besteht in der kompletten Verfügbarkeit der Quellcodes und der Dynamik der freien Software-Entwicklung. Einige wichtige, sich immer wiederholende Fragen zu JSP sind unter der Adresse *http://www.esperanto.org.nz/jsp/jspfaq.htm* beantwortet.

Alle anderen in der Tabelle 10.4 aufgelisteten Ansätze sind eher von geringer Bedeutung. Um mit den GNU-Server-Pages (GSP) arbeiten zu können, ist man gezwungen, im alten Page-Compilation-Dialekt zu formulieren. Interessant ist lediglich der eingeführte Applikationsbegriff. Auch ServletFactory generiert Servletklassen, die jedoch vom Nutzer installiert werden müssen. Die Integration von Beans fehlt leider vollkommen. Servc ist ein kleiner Servlet-Generator, und das WeAF setzt auf eine Skriptsprache zur einfacheren JDBC-Integration in HTML-Seiten.

## 10.2 JavaServer-Pages

Die JavaServer-Pages sind eine einfache, aber mächtige Möglichkeit, HTML-Seiten zu generieren. Die schon erwähnte Einbindung von Java-Beans ist eine wertvolle Ergänzung. Mit ihrer Hilfe lassen sich vorgefertigte Java-Komponenten in einer JSP verwenden. Der Anschluß an eine Datenbank, einen RMI- oder Mail-Server wird zum Kinderspiel.

Alle JavaServer-Pages werden über ein `JspServlet` angesprochen. Dafür ist es notwendig, der Servlet-Engine mitzuteilen, daß alle Anfragen, die mit `.jsp` enden, auf dieses Servlet abgebildet werden. In der Regel geschieht dies über einen Eintrag in einer der Dateien `mappings.properties`, `rules.properties` oder `servlet.properties`.

```
# GNUJSP-Servlet 1.0
servlet.gnujsp.code=org.gjt.jsp10.JSP10Servlet
servlet.gnujsp.initArgs=scratchdir=jsprepository
servlet.gnujsp.aliases=*.jsp
servlet.gnujsp.preload=true
```

*Listing 10.1: Beispiel-Eintrag für GNUJSP in der Datei `servlet.properties`* von jo!

Durch diesen Eintrag werden alle Anfragen an die Servlet-Engine mit der `.jsp`-Dateiendung an das `JspServlet` umgelenkt. Bei jeder solchen Anfrage schaut dieses Servlet nach, ob die Klasse des angeforderten Servlets schon verfügbar und geladen ist. Falls nicht, wird die JSP-Datei übersetzt, und es entsteht eine neue Servlet-Klasse, die in der Regel von der Klasse `HttpServlet` abgeleitet ist (Kapitel 3). Anschließend wird diese Klasse instantiiert und initialisiert. Nun werden die ursprüngliche Anfrage (`HttpServletRequest`) und die Antwort (`HttpServletResponse`) an das `JspServlet` übermittelt.

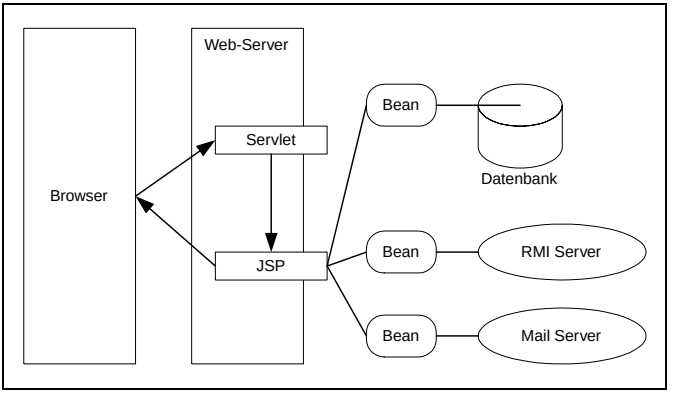


*Abbildung 10.5: Anfrage eines Browsers an JavaServer-Pages*

Generell hat eine JSP Zugriff auf das komplette Angebot des Java-API. Programmierern stehen also alle Türen offen. Was zunächst als Vorteil erscheint, wird zum Handicap, wenn HTML-Designer mit JSP arbeiten sollen. Java ist einfach zu mächtig. Damit es die JSP-Designer etwas einfacher haben, hat Sun die Möglichkeit geschaffen, JavaBeans einfach in JSP integrieren zu können. Java-Programmierer sollen ihren Designerkollegen mit vorgefertigten Komponenten den Einstieg erleichtern. Von einer direkten Realisierung einer RMI-Anbindung oder der Anwendung des Mail-API innerhalb von JSP ist unbedingt abzuraten. Hierfür sollen Standard-Beans geschaffen werden, die lästige `import`-Befehle und mächtige Vorgänge einfach verstecken. Wartbarkeit und Transparenz wachsen so enorm.

Eine wichtige Eigenschaft von JSP ist, daß bei jeder Änderung einer JSP diese automatisch neu übersetzt wird. In der Entwicklungszeit ist dies ein Segen – zumindest solange man die eigenen Klassen, die innerhalb von JSP verwendet werden, nicht ändert. Die einzige sichere Lösung bei einer Änderung der Basisklassen eines Servers besteht momentan darin, alle generierten JSP-Klassen zu löschen. Natürlich bedeutet dies, daß alle JSP bei den nächsten Anfragen erneut übersetzt werden müssen. Eine solche Bootstrap-Phase kann zu langen Wartezeiten und hoher Serverbelastung führen, wenn mehrere JSP in einer HTML-Seite benötigt werden, wie es bei HTML-Frames mit dynamischen Inhalten durchaus vorkommen kann. Zur Zeit wird versucht, ein Verfahren zu finden, mit dem die JSP-Seiten vorkompiliert ausgeliefert werden können. Für die `GNUJSP` haben wir selbst den `JSPExecutor` entwickelt (`http://www.webapp.de/`).

In den folgenden zwei Abschnitten wollen wir auf die wichtigsten Elemente der JSP-Syntax eingehen. Dies ist eine sehr kompakte Übersicht. Für eine genauere Beschreibung sei auf die JSP-Spezifikation und die entsprechenden Spezialbücher zum Thema JSP verwiesen [JSP 99], [Turau 00], [Fields/Kolb 00].

### 10.2.1 Syntax von JavaServer-Pages

Die Syntax der JavaServer-Pages hat sich von Version zu Version immer wieder stark gewandelt. Als wir das Manuskript dieses Buches abgaben, war leider die endgültige Spezifikation 1.1 nicht fertig und die Referenzimplementierung JSP 1.0 gerade verfügbar. Trotzdem wagen wir den Versuch, einen ersten Gesamtüberblick über die JSP-Syntax zu geben.

In der aktuellen Spezifikation der JavaServer-Pages 1.0 ist die JSP-Syntax als XML-Dokument noch in Arbeit (Tabelle 10.6). Natürlich ist darauf geachtet worden, daß die bisherigen übersichtlichen und kurzen `<%...%>`-Tags verfügbar bleiben (Tabelle 10.5).

Für die Definition von Methoden und Attributen ist das HTML-Script-Tag `<SCRIPT runat="server"> </SCRIPT>` durch ein kurzes `<%! %>`-Tag ersetzt worden. Die HTML-Serverside-Include-Syntax [NSCA 96] ist durch das praktische `<%@ include file="" %>` -Tag ersetzt worden.

| Tag | Beschreibung | JSP-Syntax |
|---|---|---|
| Kommentar in der HTML-Ausgabe | Erzeugen eines Kommentars, der in der HTML-Ausgabedatei erhalten bleibt. Die Ausdrücke werden ausgewertet und in die Ausgabe geschrieben. | `<!--Kommentar -->`<br>`<!--Kommentar <%= Ausdruck %> -->` |
| Kommentar in der JSP-Datei | Der Kommentar ist nur in der Orginal-JSP-Datei enthalten. | `<%-- Kommentar --%>` |
| Definition | Definition von Attributen oder Methoden in der gewählten Skriptsprache in der JSP-Seite. | `<%! Definition %>` |
| Ausdruck | Enthält einen Ausdruck in der Skriptsprache der JSP-Seite. | `<%= Ausdruck %>` |
| Scriptlet | Enthält ein Code-Fragment in der Skriptsprache der JSP-Seite. | `<% Scriptlet %>` |
| Include-Anweisung | Integration einer Datei mit Text oder Code zum Übersetzungszeitpunkt der JSP-Seite. | `<%@ include file="Pfad zur Datei" %>` |

*Tabelle 10.5: Die Standard-JSP-Tags*

| Tag | Beschreibung | JSP-Syntax |
|---|---|---|
| Seiten-Anweisung | Definition von Attributen, die für die gesamte JSP-Seite gültig sind. | <%@ page<br>[language="java"]<br>[ extends="Package-Pfad der Oberklasse" ]<br>[ import= {"Package-Pfad der Klasse" \|<br>"Package-Pfad.*" } , ... ]<br>[ contentType= { "mimeType [ ; char-<br>set=characterSet] " \| "text/html ;<br>charset=ISO-8859-1" ]<br>[ session= { "true" \| "false" } ]<br>[ buffer= {"none"\| "8kb" \| Größe in kb } ]<br>[ autoFlush= {"true" \| "false" } ]<br>[ isThreadSafe= { "true" \| "false" }]<br>[ errorPage="Pfad zur Fehlerseite" ]<br>[ isErrorPage={ "true" \| "false" } ]<br>[ info="text" ]<br>%> |
| TagLib-Anweisung | Definition einer Tag- Bibliothek | <%@ taglib uri="URI zur DTD der Tags" prefix="Prefix der Tags" %> |

*Tabelle 10.5: Die Standard-JSP-Tags (Fortsetzung)*

Große Unterschiede in JSP 1.1 zu allen Vorgängern sind durch das `page`- und `Taglib`-Tag gegeben. Mit dem `page`-Tag werden alle für die Seite notwendigen Initialisierungsparameter und übergreifenden Laufzeitparameter definiert. Gut gelungen ist hier die Integration einer übergreifenden Fehlerbehandlung, die die lästigen `try-catch`-Blöcke obsolet macht. Damit ist es möglich, jeder JSP eine übergreifende Fehlerseite zuzuordnen. Die Ausgaben lassen sich nun durch Parameter wie Puffergröße und Autoflush besser steuern. Zudem läßt sich angeben, ob die JSP threadsicher programmiert ist. Die Integration der `import`-Anweisung an dieser Stelle ist etwas umständlich. Es muß nun in jeder JSP die entsprechende Liste zusammengestellt und gepflegt werden. Zuvor konnte man einfach eine gemeinsam genutzte Datei mit `import`-Anweisungen mittels `include`-Tag einbinden.

Die wirkliche Neuerung liegt in der Einbindung weiterer spezieller Tags, die heute die rein interpretierten Lösungen (Abschnitt 10.1) attraktiv für den HTML-Designer machen. Durch das `<%@ taglib .. %>`-Tag wird eine DTD eigener Tags angeben. Für jedes Tag muß dabei eine spezielle Klasse für die Behandlung der JSP-Engine definiert sein. Hier ist ein neues Package `javax.servlet.jsp.taglib` hinzugekommen, das die Infrastruktur für eine portable Realisierung solcher Besonderheiten definiert. Für die genauen Details dieser Integration sei auf die JSP-Spezifikation verwiesen [JSP 99], [Turav 00], [Fields/Kolb 00]. Wir sind sehr gespannt auf die Umsetzung in den diversen HTML-Werkzeugen und Java-Entwicklungsumgebungen. Sowohl Sun, IBM, BEA als auch Allaire haben ihre Unterstützung zugesagt.

### 10.2.2 Integration von Java-Beans in JavaServer-Pages

In der Spezifikation JSP 1.0 sind eine Reihe neuer JSP-Tags hinzugekommen, die komplexere Funktionen kapseln. Der schon im API 2.1 vorhandene `RequestDispatcher` wurde nun durch die Standard-Tags `<jsp:include>` und `<jsp:foward>` integriert. Damit wird es möglich, in einer JSP einfach Verzweigungen und Integrationen zu implementieren. Die Vorgangssteuerung ist so im Designer-Entwicklungsteam direkter umzusetzen. Das `<jsp:plugin>`-Tag dient zur einfachen HTML-Integration von Applets und Beans in die Ausgabe (Tabelle 10.6).

| Tag | Beschreibung | JSP-Syntax |
|---|---|---|
| `<jsp:forward>` Weiterleitung an anderen URI zur Laufzeit | Innerhalb eines Verteilers (Dispatcher JSP) wird entschieden, daß eine andere JSP, Ressource oder ein anderes Servlet zur Ausgabe genutzt werden soll. | `<jsp:forward page="Context relativer URI" />` |
| `<jsp:include>` Einfügen von Inhalten zur Laufzeit. | Eine Seite wird durch statische oder dynamische Inhalte ergänzt. Es können nur Ressourcen derselben Applikation genutzt werden. | `<jsp:include page="Context relativer URI" />` |
| `<jsp:plugin>` Erzeugt HTML-Code für Applet oder Bean | Erzeugt HTML-Anweisungen zum Download des Java-Plugins zur Ausführung eines Applets oder Beans auf dem Client-Browser. | `<jsp:plugin`<br>`type={ "bean" \| "applet" }`<br>`[code="Objektklasse" ]`<br>`[codebase="Basispfad für Objektcode" ]`<br>`[name="Name der Instanz auf der HTML-Seite"]`<br>`[archive="Liste der Klassenarchive" ]`<br>`[align={"button"\|"top"\|"middle"\|"left"\|"right"}]`<br>`[height="displayHeight"]`<br>`[width="displayWidth" ]`<br>`[hspace="horizontalGutter"]`<br>`[vspace="verticalGutter"]`<br>`[jreversion={"JREVersionNumber"\|"1.1"}]`<br>`[nspluginurl="URL zum Netscape Navigator Plugin" ]`<br>`[iepluginurl="URL zum Microsoft Internet Explorer Plugin" ]`<br>`>`<br>`[<params> <param name="Name des Parameters"` |

*Tabelle 10.6: Erweiterte JSP-Tags zur Integration von Beans, Applets und anderen Servlet-Ressourcen*

| Tag | Beschreibung | JSP-Syntax |
|---|---|---|
| | | value="Wert des Parameters">+ </params> ]<br>[<fallback> Text einer Fehlermeldung für den Nutzer </fallback> ]<br></jsp:plugin> |
| <jsp:useBean><br>Nutzung eines Java-Beans | Regelt den Zugriff auf ein JavaBean. Falls es nicht im entsprechenden Kontext vorhanden ist, wird es erzeugt. | <jsp:useBean<br>id="Identität des Beans"<br>scope={"page"\|"request"\|"session"\|"application"}<br>{ class="Package-Pfad der Klasse" \|<br>type="Package-Pfad der Klasse" \|<br>class="Package-Pfad der Klasse"<br>type="Package-Pfad zur Schnittstelle oder Klasse" \|<br>beanName="Name des Beans" type="Package-Pfad zur Schnittstelle oder Klasse" }<br>{ /> \|<br>oder andere JSP oder HTML-Tags </jsp:useBean><br>} |
| <jsp:getProperty><br>Lesen eines Bean-Attributs | Zugriff auf ein Attribut eines Beans und die Formatierung des Wertes als Zeichenkette. | <jsp:getProperty<br>name="Identität des Beans"<br>property="Name des Attributs" /> |
| <jsp:setProperty><br>Schreiben eines Bean-Attributs | Setzen eines Attributs eines Beans. Dabei wird aus der Zeichenkette für ausgewählte Datentypen der entsprechende Wert generiert. | <jsp:setProperty<br>name="Identität des Beans"<br>{ property="*" \|<br>property="Name des Attributs" [<br>param="Name des Parameters" ] }<br>value={"Zeichenkette""\| "<%= Ausdruck %>" }<br>/> |

*Tabelle 10.6: Erweiterte JSP-Tags zur Integration von Beans, Applets und anderen Servlet-Ressourcen (Fortsetzung)*

Die Integration von vorgefertigten Komponenten zur Aufbereitung von Informationen ist ein sehr effizienter Ansatz. Natürlich benutzt Sun hierfür Java-Beans. Diese lassen sich mittels der <jsp:useBean>-Markierung in eine Seite einbinden (Listing 10.5).

```
<jsp:useBean id="CALENDAR"
     class=" calendar.JspCalendar "
     scope="request" />
<jsp:setProperty name="CALENDAR" property="TimeZone" value="ECT" />
```

*Listing 10.2: Beispiel einer Definition eines Java-Beans innerhalb einer JavaServer-Page*

Jedes JSP-Bean besitzt einen eindeutigen Namen, eine Klasse und einen Lebenszyklus (Listing 10.5). Der Modus (`scope`) `request` gibt an, daß bei jeder Anfrage ein solches Bean erzeugt wird. Im Modus `session` wird das Bean an die `HttpSession` des Browser-Clients gebunden und dann bei weiteren Anfragen wiederverwendet. Im Modus `page` schließlich werden die Informationen lokal für diese JSP gespeichert. Übergreifend können Werte im `ServletContext` mittels des Modus `application` hinterlegt bzw. erreicht werden. Der Zugriff auf die Attribute wird durch die Markierung `<jsp:getProperty>` ermöglicht. Das Setzen eines Wertes eines Bean-Attributes wird durch das `<jsp:setProperty>`-Tag unterstützt.

Als Beispiel zur JSP-Spezifikation wird das `dates.JspCalendar`-Bean mitgeliefert, das zuverlässig alle Daten des Abfragezeitpunkts bestimmt.

```
<HTML>
<HEAD>
   <TITLE>JSP-WebApp-Zeit</TITLE>
   <jsp:useBean id="CLOCK" class="dates.JspCalendar" scope="request"
type="dates.JspCalendar"/>
</HEAD>
<BODY>
<UL>
<LI>   Tag des Monats:  <jsp:getProperty name="CLOCK"  property="dayOfMonth"/>
<LI>   Jahr:  <jsp:getProperty name="CLOCK"  property="year"/>
<LI>   Monat: <jsp:getProperty name="CLOCK"  property="month"/>
<LI>   Zeit: <jsp:getProperty name="CLOCK"  property="time"/>
<LI>   Datum: <jsp:getProperty name="CLOCK"  property="date"/>
<LI>   Tag: <jsp:getProperty name="CLOCK"  property="day"/>
<LI>   Tag des Jahres: <jsp:getProperty name="CLOCK"  property="dayOfYear"/>
<LI>   Woche des Jahres: <jsp:getProperty name="CLOCK"  property="weekOfYear"/>
</UL>
</BODY>
</HTML>
```

*Listing 10.3: Anwendungsbeispiel für ein Date-Bean*

Bean-Attribute müssen immer durch entsprechende Anführungszeichen eingeschlossen werden. Der Zugriff auf ein Attribut erfolgt durch die Angabe des Bean-Bezeichners (`CLOCK`). Das Beispiel in Listing 10.5 zeigt, daß wir ohne expliziten Gebrauch von Java dynamisch HTML-Seiten formulieren können. Einige werden sich noch an das `DISPLAY`-Tag der 0.92-Spezifikation erinnern und hoffen, daß es bald wieder in einer `Taglib` auftaucht. Besser wäre noch ein Formatter-Konzept, wie es `objectHTML` bietet (Tabelle 10.3).

Zur einfacheren und übersichtlicheren Handhabung werden einige Objekte direkt durch bestimmte Variablen angeboten (Tabelle 10.7).

| Implizite Objekte einer JSP | <jsp:useBean>"scope" | Typ |
|---|---|---|
| application | Application | javax.servlet.ServletContext |
| session | Session | javax.servlet.http.HttpSession |
| config | Page | javax.servlet.ServletConfig |
| pageContext | Page | javax.servlet.jsp.PageContext |
| page | Page | javax.servlet.jsp.JspPage |
| response | Page | Unterklasse von javax.servlet.ServletResponse |
| request | Request | Unterklasse von javax.servlet.ServletRequest |
| out | Page | javax.servlet.jsp.jspWriter |
| exception | Page | java.lang.Throwable (Nur verfügbar, wenn isErrorPage="true") |

*Tabelle 10.7: Standardvariablen, die in einer JavaServer-Page verfügbar sind*

Die Syntax der Beans hat sich grundsätzlich von der Version 0.91 über die Version 0.92 und zur Version 1.0/1.1 der JSP-Spezifikation geändert. Bis zur endgültigen Veröffentlichung der Version 1.0/1.1 wird sich sicherlich noch einiges ändern. Dennoch hoffen wir, daß die grundsätzlichen Syntax-Elemente gleich bleiben werden. Diese werden an zwei Beispielen im nächsten Abschnitt noch einmal verdeutlicht.

### 10.2.3 Einfache Beispiele für JavaServer-Pages

Natürlich können wir auch ohne Bean-Integration mit Java direkt auf das Datum zugreifen (Listing 10.7). Allerdings ist offensichtlich, daß dies lange nicht so elegant ist wie mit dem Einsatz von Beans:

```
<%page import="java.util.Date" %>
<HTML>
<HEAD>
   <TITLE>JSP-WebApp-Datumsseite</TITLE>
</HEAD>
<BODY>
<H1>JSP-WebApp-Datumsseite </H1>
Hallo WebApp'er - Heute ist der <%= new Date() %>.
</BODY>
</HTML>
```

*Listing 10.4: Realisierung einer JSP zur Anzeige eines Datums*

Das uns schon aus Kapitel 3 bekannte `RequestInfoServlet` läßt sich auch als JSP formulieren (Listing 10.8). Zuerst wird die Anfrage ausgegeben.

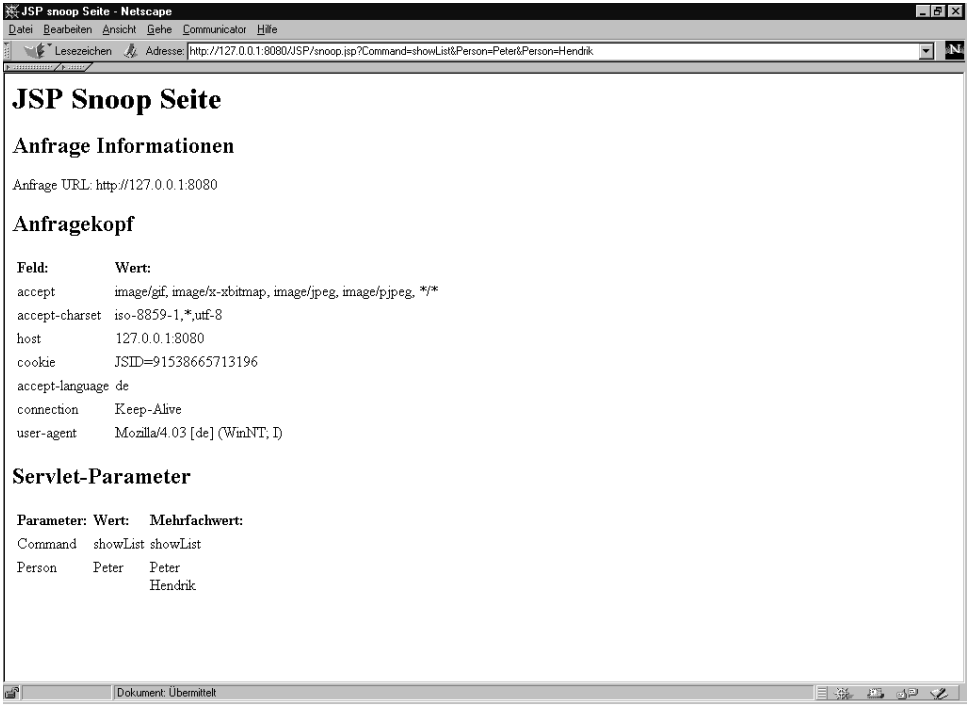


*Abbildung 10.6: Servlet snoop als JavaServer-Page*

Es folgen die eventuell vorhandenen Kopfinformationen, welche eigentlich bei jeder Browser-Anfrage angegeben sind. Abgeschlossen wird die Ausgabe mit der Formatierung der Anfrageparameter.

```
<%page import="javax.servlet.http.HttpUtils, java.util.Enumeration" %>
<HTML>
<HEAD>
  <TITLE>JSP snoop Seite</TITLE>
</HEAD>
<BODY>

<H1>JSP Snoop Seite</H1>
<H2>Anfrage Informationen</H2>
<align=right>Anfrage URL: <%= HttpUtils.getRequestURL(request) %>
<%
  Enumeration e = request.getHeaderNames();
  if(e != null && e.hasMoreElements()) {
%>
<H2>Anfragekopf</H2>
<TABLE>
<TR>
  <TH align=left>Feld:</TH>s
```

```
  <TH align=left>Wert:</TH>
</TR>
<%
  while(e.hasMoreElements()) {
    String k = (String) e.nextElement();
%>
<TR>
  <TD><%= k %></TD>
  <TD><%= request.getHeader(k) %></TD>
</TR>
<%}%>
</TABLE>
<%}%>
<%
  e = request.getParameterNames();
  if(e != null && e.hasMoreElements()) {
%>
<H2>Servlet-Parameter</H2>
<TABLE>
<TR valign=top>
  <TH align=left>Parameter:</TH>
  <TH align=left>Wert:</TH>
  <TH align=left>Mehrfachwert:</TH>
</TR>
<%
  while(e.hasMoreElements()) {
    String k = (String) e.nextElement();
    String val = request.getParameter(k);
    String vals[] = request.getParameterValues(k);
%>
<TR valign=top>
  <TD><%= k %></TD>
  <TD><%= val %></TD>
  <TD><%
    for(int i = 0; i < vals.length; i++) {
      if(i > 0)
        out.print("<BR>");
        out.print(vals[i]);
    }
  %></TD>
</TR>
<%}%>
</TABLE>
<%}%>
</BODY>
</HTML>
```

*Listing 10.5: Snoop-JavaServer-Page*

# Teil III
# Anwendungen

Mit dem WebApp-Framework steht uns nun eine leistungsfähige Grundlage zur Entwicklung von webbasierten Anwendungen zur Verfügung. Trotz dieser vergleichsweise komfortablen Umgebung ist das Design von Web-Anwendungen, insbesondere wegen der Webbrowser als Clients und des zustandslosen HTTPs, nicht einfach. In diesem Buchteil werden wir daher anhand von drei Beispielen zeigen, wie mit dem WebApp-Framework Anwendungen realisiert werden können. In Kapitel 11 stellen wir dazu einen generischen Browser für die in Kapitel 9 beschriebenen Stores vor. Kapitel 12 beschreibt einen OnlineShop und Kapitel 13 zeigt die Entwicklung einer browserbasierten Chat-Anwendung.

Generell unterteilen wir dabei den Entwicklungsprozeß in vier Phasen:

1. Festlegen der Abläufe, Informationsobjekte und Dialoge
2. Kodierung der Klassen
3. Erstellen der Anzeige-Templates
4. Erstellen der Konfigurationsdateien

Es ist wichtig, sich darüber im klaren zu sein, daß bereits in Phase Eins alle wichtigen Entscheidungen getroffen werden. Phase Zwei, Drei und Vier sind lediglich Produktionsphasen, in denen die Entscheidungen aus Phase Eins umgesetzt werden.

Dementsprechend liegt der formale Schwerpunkt in der Phase Eins. Vor der Kodierung legen wir ein Anwendungsfall- oder Use-Case-Diagramm an [Oestereich 98, S.207ff], um einen Überblick über die möglichen Aktionen zu erlangen. Anschließend bilden wir dies in einem Ablaufdiagramm ab, für das wir die Notation der Zustandsdiagramme verwenden [Oestereich 98, S. 310ff]. Als Ereignisse dienen uns dabei die SMI-Kommandos aus Kapitel 8. Auf diese Weise erhalten wir eine Liste der Kommandos und eine genaue Übersicht über die Dialogfolge.

Um die Kommandos und damit das Verhalten der Anwendung genauer zu definieren, erstellen wir dann eine Kommandotabelle. Sie enthält eine Auflistung der Kommandos, benötigte Parameter sowie eine Funktionsbeschreibung jedes Kommandos.

So gewappnet können wir mit dem Kodieren der Klassen und Dialoge beginnen. Dabei legen wir Wert darauf, Anwendungslogik und reine Anzeige so weit wie möglich zu trennen. Erfahrungsgemäß ist es sinnvoll, zunächst die abstrakteren Teile einer Anwendung zu kodieren. Da die Anwendungslogik allgemeinerer Natur ist als die HTML-Seiten oder JavaServer-Pages (Kapitel 10), die ihr zur Anzeige dienen, beginnen wir mit dem Kodieren der Klassen (Phase Zwei). Erst danach – in Phase Drei – kodieren wir die Anzeigeelemente. In Phase Vier fügen wir schließlich alles mittels der SMI-Konfiguration zusammen.

Natürlich ist es nicht immer möglich, sich strikt an die von uns vorgeschlagene Reihenfolge zu halten. Sie hat uns jedoch als Leitfaden wertvolle Dienste geleistet und uns geholfen, schneller bessere Ergebnisse zu produzieren.

Der vorgeschlagene Entwicklungsprozeß ist zudem übertragbar auf andere Architekturen zum Bau von webbasierten Anwendungen.

Entsprechend diesem übergeordneten Ziel haben wir in den vorgestellten Beispielen Wert auf den Entwicklungsvorgang gelegt. Fehlerbehandlung und optische Umsetzung wurden dabei ein wenig vernachlässigt, um die Grenzen dieses Buches nicht zu sprengen.

# 11 StoreBrowser

Das erste Beispiel für das WebApp-Framework ist ein generischer Browser für Stores, wie sie in Kapitel 9 beschrieben sind. Mit ihm soll es möglich sein, die Struktur eines Stores einzusehen, ihn mit Initialdaten zu füllen und Korrekturen an einzelnen Objekten vorzunehmen. Dabei soll immer eine objektorientierte Sicht auf die Daten gewahrt bleiben. Abbildung 11.1 zeigt die Anwendungsfälle des StoreBrowsers.

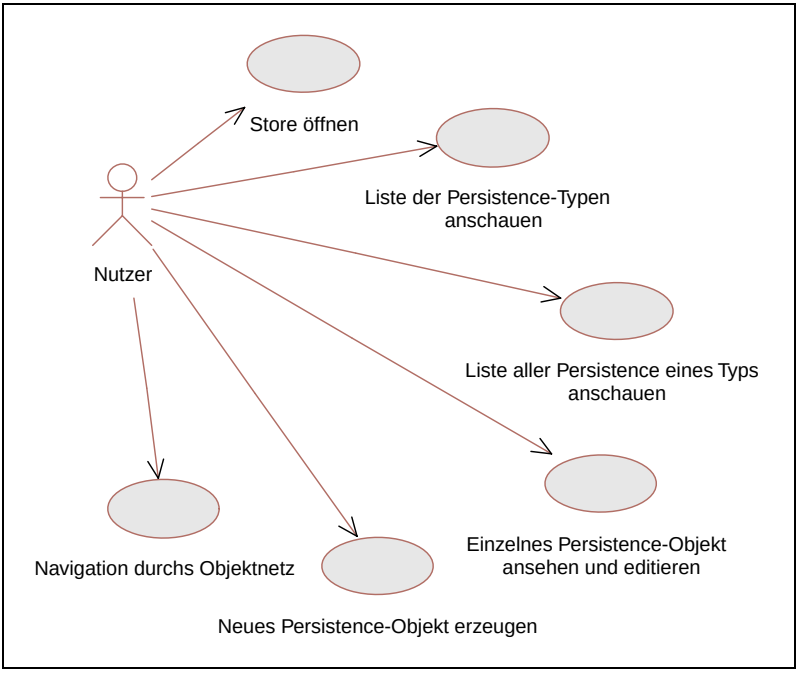


*Abbildung 11.1: Anwendungsfälle des StoreBrowsers*

Es ergeben sich folgende Anforderungen:

- Zugang zu einem beliebigen Store
- Überblick über die `Persistence`-Typen eines Stores

- Auflistung aller `Persistence`-Objekte eines Typs
- Detailsicht auf ein `Persistence`-Objekt
- Objekte erzeugen, ändern und löschen
- Navigation über Assoziationen
- Nutzerführung über ein Menü

Um die Anforderungen umzusetzen, wollen wir zunächst etwas mehr Klarheit über die abzubildenden Abläufe gewinnen.

## 11.1 Abläufe

Nach dem Starten des StoreBrowsers soll der Nutzer zur Eingabe eines Storenamens aufgefordert werden (Abbildung 11.2). Hat er dies getan, sollen auf der nächsten Seite alle Objekttypen dieses Stores angezeigt werden. Nun kann der Nutzer einen der Typen wählen und sich eine Listenansicht aller Objekte dieses Typs anzeigen lassen. Diese soll nicht nur die Attribute der Objekte anzeigen, sondern auch Verweise auf assoziierte Objekte. Insgesamt sollen drei Navigations-Möglichkeiten geboten werden:

1. Es soll sich ein `Persistence`-Objekt auswählen lassen, das dann im Detail angezeigt wird und bearbeitet werden kann.
2. Ein Verweis soll direkt zu einem Eingabedialog für ein `Persistence`-Objekt führen.
3. Jedes `Persistence`-Objekt, das mit einem oder mehreren anderen `Persistence`-Objekten assoziiert ist, muß die Möglichkeit der Navigation dorthin bieten.

Möglichkeit Eins und Zwei führen im Grunde zum selben Dialog, nämlich einer Detailsicht auf ein Objekt des gewählten Typs. Der Unterschied liegt lediglich darin, daß Möglichkeit Eins die Manipulation eines bestehenden `Persistence`-Objekt mittels der Funktionen Ändern und Löschen ermöglicht, während Möglichkeit Zwei nur das Erzeugen eines `Persistence`-Objekts erlaubt. Um das Beispiel nicht unnötig kompliziert zu machen, fassen wir beide Möglichkeiten in einem Dialog zusammen.

Möglichkeit Drei kann zu zweierlei Dialogen führen. Bei einer `ToOne`-Assoziation muß ähnlich wie in Möglichkeit Eins und Zwei eine Detailansicht des assoziierten Objekts angezeigt werden. Bei einer `ToMany`-Assoziation hingegen muß eine Listenansicht des assoziierten Objekttyps gezeigt werden.

Das Ablaufdiagramm in Abbildung 11.2 zeigt die erforderlichen Dialoge mit den möglichen Ereignissen. Der besseren Übersicht wegen ist die Möglichkeit weggelassen, über ein Menü zu allen hierarchisch untergeordneten Dialogen zu gelangen. Abbildung 11.3 zeigt das vollständige Diagramm. Da Web-Anwendungen nicht über ein klar definiertes Ende im üblichen Sinne verfügen, enthalten beide Diagramme keinen End-Zustand.

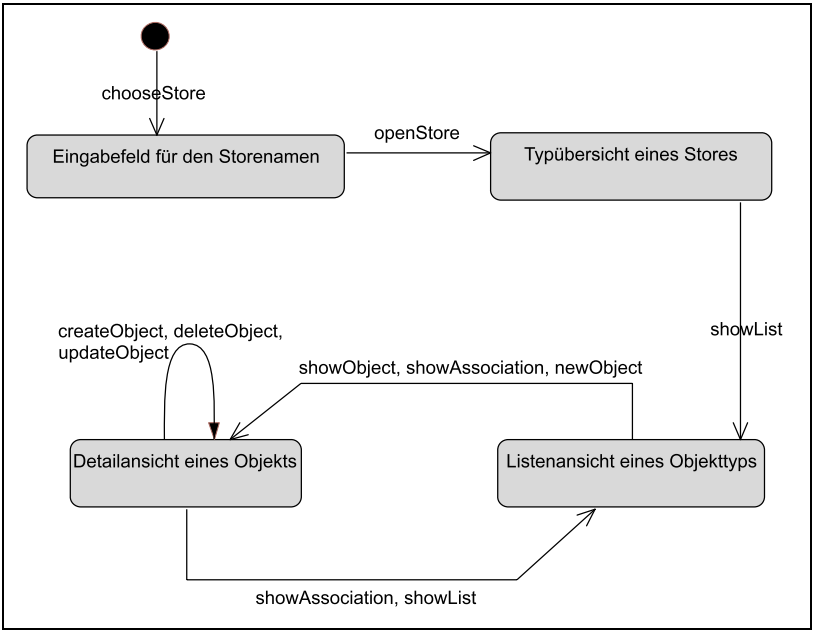


*Abbildung 11.2: Vereinfachtes Ablaufdiagramm des StoreBrowsers*

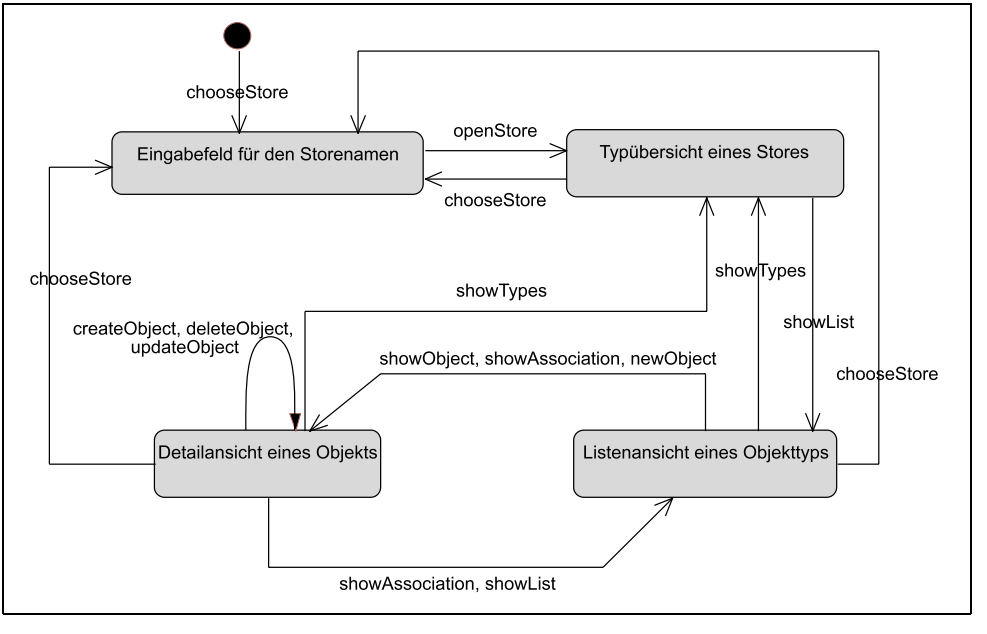


*Abbildung 11.3: Vollständiges Ablaufdiagramm des StoreBrowsers*

Aus dem Diagramm ergeben sich zwangsläufig die Ereignisse beziehungsweise Kommandos, die zu einem Dialogwechsel führen. Zum besseren Verständnis zeigt Tabelle 11.1 eine Auflistung der möglichen Kommandos und ihrer Bedeutung.

| Kommando | Parameter | Bedeutung |
|---|---|---|
| `chooseStore` | - | Schließt einen vorhandenen Store und zeigt ein Eingabefeld an, in das der Nutzer den Namen eines neuen Stores eingeben kann. Löst das Kommando `openStore` aus. |
| `openStore` | Name des Stores | Öffnet einen Store und zeigt dessen Typen an. Die Auswahl eines Typs löst das Kommando `showList` aus. |
| `showTypes` | - | Zeigt eine Übersicht aller Objekttypen des gewählten Stores an. Die Auswahl eines Objekttyps löst das Kommando `showList` aus. |
| `showList` | Objekttyp | Zeigt eine Übersicht aller Objekte eines Typs. Von diesem Dialog aus können die drei Kommandos `showObject`, `newObject` und `showAssociation` ausgelöst werden. |
| `showObject` | Objektschlüssel und Objekttyp | Zeigt eine editierbare Detailsicht eines Objekts an. Von diesem Dialog aus können die vier Kommandos `createObject`, `updateObject`, `deleteObject` und `showAssociation` ausgelöst werden. |
| `newObject` | Objekttyp | Zeigt den Eingabedialog für ein Objekt an. Es kann nur das Kommando `createObject` ausgelöst werden. |
| `showAssociation` | Assoziationsname, `ObjectIdentifier` des Ausgangsobjekts | Zeigt ein einzelnes assoziiertes Objekt bzw. eine Liste assoziierter Objekte an. Siehe `showList` und `showObject`. |
| `createObject` | Objekttyp | Legt ein neues Objekt an. Anschließend bieten sich die gleichen Möglichkeiten wie unter `showObject` beschrieben. |
| `updateObject` | `ObjectIdentifier`, Attribute | Ändert ein bestehendes Objekt. Anschließend bieten sich die gleichen Möglichkeiten wie unter `showObject` beschrieben. |
| `deleteObject` | `ObjectIdentifier` | Löscht ein bestehendes Objekt. Anschließend bieten sich die gleichen Möglichkeiten wie unter `newObject` beschrieben. |

*Tabelle 11.1: Beschreibung der Kommandos des StoreBrowsers. Bei den möglichen Folgekommandos wurde auf die Ereignisse verzichtet, die aus Menüeinträgen resultieren.*

## 11.2 Design der Klassen

Nachdem wir definiert haben, welche Abläufe der StoreBrowser umsetzen muß, kommen wir nun zum Design der Klassen. Bevor wir überhaupt einen Store benutzen können, müssen wir ihn öffnen. Nach Gebrauch sollte er wieder geschlossen werden. Nun stellt sich die Frage, in welcher Datenstruktur der Store während der Benutzung hinterlegt werden soll. Da verschiedene Nutzer des StoreBrowsers gleichzeitig unterschiedliche Stores betrachten können sollten und sich dabei nicht in die Quere kommen dürfen, fällt unsere Entscheidung zugunsten der nutzerspezifischen `HttpSession` (Kapitel 3).

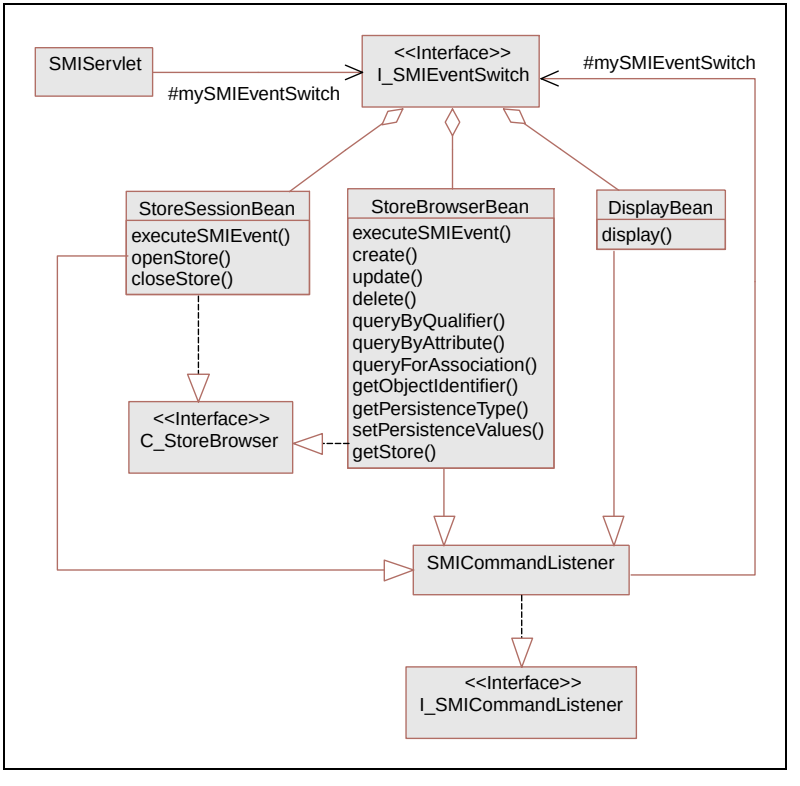


*Abbildung 11.4: Klassendiagramm des StoreBrowsers*

Weiterhin entscheiden wir uns dafür, die Verwaltungsfunktionen des Stores (Öffnen und Schließen) von seinen Manipulationsfunktionen zu trennen und sie in einem eigenen Bean (Kapitel 8) zu kapseln. Wir nennen das resultierende Verwaltungs-Bean `StoreSessionBean` (Listing 11.1) und das Manipulations-Bean `StoreBrowserBean`. Beide

erben von SMICommandListener und implementieren die Schnittstelle C_StoreBrowser, in der alle notwendigen Konstanten definiert sind (Abbildung 11.4).

```
package de.webapp.StoreBrowser;

import de.webapp.Framework.SMI.*;
import de.webapp.Framework.SMICommandListener.*;
import de.webapp.Framework.ConfigManager.*;
import de.webapp.Framework.StoreFactory.*;
import de.webapp.Framework.Persistence.*;

/** Kapselt die Verwaltung eines Stores in einer HttpSession */
public class StoreSessionBean extends SMICommandListener
   implements C_StoreBrowser, C_SMIStore {

/** Leitet ein SMIEvent weiter und ruft ein Folgekommando auf */
public void executeSMIEvent(SMIEvent aSMIEvent) {
   // Kommando ausführen
   super.executeSMIEvent(aSMIEvent);
   SMICommand aSMICommand = (SMICommand)aSMIEvent;
   // Folgekommando erlangen und ausführen
   String followUpCommand =
      (String)aSMICommand.getValue(C_FollowUpCommand);
   if (followUpCommand != null) {
      // Folgekommando setzen ...
      aSMICommand.setCommand(followUpCommand);
      // ... und ausführen
      getSMIEventSwitch().executeSMIEvent(aSMICommand);
   }
}
/** Instantiiert einen Store und hinterlegt diesen in der Session */
public void openStore(SMICommand aSMICommand)
      throws PersistenceException {
   // Name des Stores in Erfahrung bringen
   String theStoreName = (String)aSMICommand.getValue(C_StoreName);
   // Konfiguration des Stores erlangen
   Configuration theConfiguration =
      ConfigManager.getConfigManager().getConfiguration(
         theStoreName,
         theStoreName + C_StoreSuffix, true);
   // Store instantiieren ...
   I_Store theStore = (I_Store)StoreFactory.getInstance(theConfiguration);
   // ... und in der HttpSession hinterlegen
   aSMICommand.putSessionValue(C_Store, theStore);
   aSMICommand.putSessionValue(C_StoreName, theStoreName);
}
/** Schließt einen Store und entfernt ihn aus der Session */
public void closeStore(SMICommand aSMICommand) throws PersistenceException {
   I_Store theStore = (I_Store)aSMICommand.getSessionValue(C_Store);
   if (theStore != null) {
      theStore.close();
```

```
      theStore = null;
      // Store aus der HttpSession entfernen
      aSMICommand.removeSessionValue(C_Store);
   }
}
} // Ende der Klasse
```

*Listing 11.1: Die Klasse `StoreSessionBean`*

Die Klasse `StoreSessionBean` besteht aus nur drei Methoden: `openStore(SMICommand)`, `closeStore(SMICommand)` und `executeSMIEvent(SMIEvent)`. Die Methode `openStore()` geht davon aus, daß sich im `SMICommand` ein Parameter mit dem Schlüssel `C_StoreName` befindet. Um die Konfiguration eines Stores lesen zu können, nehmen wir an, daß sich die Konfigurationsdatei eines Stores zur Anwendung »OnlineShop« im Konfigurationsverzeichnis des Registry-Eintrags `OnlineShop` befindet und den Namen `OnlineShop.store` hat (Kapitel 12). Treffen diese Annahmen zu, ist es möglich, mittels des Storenamens aus dem `SMICommand` die Konfiguration des Stores zu lesen und an eine `StoreFactory` zu übergeben, die den Store instantiiert und initialisiert. Anschließend werden Store und Storename unter den Schlüsseln `C_Store` und `C_StoreName` in der Session hinterlegt. Beide Konstanten sind in der Schnittstelle `C_StoreBrowser` (Listing 11.2) definiert.

```
package de.webapp.StoreBrowser;

/** Konstanten des StoreBrowsers */
public interface C_StoreBrowser {

public static final String C_PersistenceType = "_PersistenceType";
public static final String C_PersistenceAttributePrefix = "_PA";
public static final String C_PersistenceOID = "_ObjectIdentifier";
public static final String C_StoreName = "_Storename";
public static final String C_Store = "_Store";
public static final String C_StoreSuffix = ".store";
public static final String C_PersistenceSQLQualifier = "_SQLQualifier";
public static final String C_PersistenceSearchValue = "_SearchValue";
public static final String C_PersistenceSearchAttribute =
   "_SearchAttribute";
public static final String C_PersistenceAssociationName =
   "_AssociationName";
public  static final String C_PersistenceResult = "_PersistenceResult";
} // Ende der Schnittstelle
```

*Listing 11.2: Die Konstanten des StoreBrowsers sind in der Schnittstelle `C_StoreBrowser` definiert.*

Die Methode `closeStore()` schließt den mit `openStore()` geöffneten Store wieder. Dazu wird der Store aus der Session ermittelt, per `close()` geschlossen und dann aus der Session entfernt. Sollte es sich um einen Store handeln, der die Schnittstelle `HttpSessionBindingListener` implementiert, kann es sein, daß beim Entfernen aus der Session erneut versucht wird, den Store zu schließen. Da sich Stores beliebig oft schließen lassen, ohne daß dies zu einem Fehler führt, können wir diesen Vorgang getrost ignorie-

ren. Den Storenamen belassen wir in der Session, um später auf einen Defaultwert zurückgreifen zu können.

Bleibt die Methode `executeSMIEvent()` zu erklären: Sie ist überschrieben worden, um automatisch ein Folgekommando ausführen zu können. Falls im Kommando ein Parameter mit dem Schlüssel `C_FollowUpCommand` (aus `de.webapp.Framework.C_SMI`) definiert ist, wird dieser als Folgekommando interpretiert und über den `SMIEventSwitch` ausgeführt. Dies geschieht, nachdem über `super.executeSMIEvent()` das ursprüngliche Kommando ausgeführt wurde.

Da Methoden in `SMICommandListenern` selten über Ausgabefunktionen verfügen, kommt es sehr häufig vor, daß ein Folgekommando zur Darstellung eines Ergebnisses aufgerufen wird. Genauso ist es auch im `StoreSessionBean`. Das Öffnen und Schließen des Stores führt zu keinerlei Ausgabe, da wir als Designentscheidung Anzeige und Anwendungslogik trennen. Zunächst wollen wir uns jedoch nur um die Anwendungslogik kümmern – zur Anzeige kommen wir später (Abschnitt 11.3).

Nachdem wir nun über die Möglichkeit verfügen, einen Store in einer Session zu hinterlegen und zu verwalten, müssen wir Methoden zur Benutzung dieses Stores erstellen. Diese sind im `StoreBrowserBean` (Listing 11.3) zusammengefaßt.

```
package de.webapp.StoreBrowser;

import java.util.*;

import de.webapp.Framework.SMI.*;
import de.webapp.Framework.SMICommandListener.*;
import de.webapp.Framework.Persistence.*;
import de.webapp.Framework.Utilities.ReferenceVector;

/** Bean zum Zugriff auf einen sessiongebundenen Store */
public class StoreBrowserBean extends SMICommandListener
   implements C_StoreBrowser, C_SMIStore {

/** Legt um jede Aktion einen Transaktionsrahmen und führt ein Folgekommando aus */
public void executeSMIEvent(SMIEvent aSMIEvent) {
   SMICommand aSMICommand = (SMICommand)aSMIEvent;
   // Store aus der HttpSession erlangen
   I_Store theStore = getStore(aSMICommand);
   // alleinigen Zugriff sichern
   synchronized (theStore) {
      try {
         // Transaktion beginnen
         theStore.begin();
         // Kommando ausführen
         super.executeSMIEvent(aSMIEvent);
         // Transaktion beenden
         theStore.commit();
         // Folgekommando ausführen
```

```
            String followUpCommand =
                (String)aSMICommand.getValue(C_FollowUpCommand);
            if (followUpCommand != null) {
                aSMICommand.setCommand(followUpCommand);
                getSMIEventSwitch().executeSMIEvent(aSMICommand);
            }
        }
        catch (Throwable t) {
            try {
                // im Fehlerfall Rollback versuchen
                theStore.rollback();
            }
            catch (PersistenceException pe) {
                pe.setThrowable(t);
                throw new SMIEventListenerException(
                   "Fehler beim Rollback.", pe);
            }
            throw new SMIEventListenerException("Persistence-Fehler.", t);
        }
    }
}
/** Erzeugt ein persistentes Objekt. */
public void create (SMICommand aSMICommand)
       throws PersistenceException {
    // Store aus der HttpSession erlangen
    I_Store theStore = getStore(aSMICommand);
    // Neues Persistence eines bestimmten Typs erzeugen
    I_Persistence thePersistence =
        theStore.newPersistence(getPersistenceType(aSMICommand));
    // Attribute aus der Anfrage setzen
    setPersistenceValues(thePersistence, aSMICommand);
    // Persistence-Objekt in der Datenbank speichern
    thePersistence.create();
    // Persistence-Objekt im Kommando hinterlegen
    aSMICommand.putValue(C_PersistenceResult, thePersistence);
}
/** Ändert persistentes Objekt dauerhaft. */
public void update (SMICommand aSMICommand)
       throws PersistenceException {
    // Store aus der HttpSession erlangen
    I_Store theStore = getStore(aSMICommand);
    // Objekt anhand eines ObjectIdentifiers aus dem Store holen
    I_Persistence thePersistence =
       theStore.persistenceWithKey(getObjectIdentifier(aSMICommand));
    // Attribute aus der Anfrage setzen
    setPersistenceValues(thePersistence, aSMICommand);
    // Änderung in der Datenbank speichern
    thePersistence.update();
    // Geändertes Objekt im Kommando hinterlegen
    aSMICommand.putValue(C_PersistenceResult, thePersistence);
}
```

```
/** Löscht persistentes Objekt aus der Datenbank */
public void delete (SMICommand aSMICommand)
      throws PersistenceException {
   // Store aus der HttpSession erlangen
   I_Store theStore = getStore(aSMICommand);
   // Objekt anhand eines ObjectIdentifiers aus dem
   // Store holen und löschen
   theStore.persistenceWithKey(getObjectIdentifier(aSMICommand)).delete();
}
/** Suche per SQL-Qualifizierung. */
public void queryByQualifier (SMICommand aSMICommand)
      throws PersistenceException {
   // Store aus der HttpSession erlangen
   I_Store theStore = getStore(aSMICommand);
   // Retriever des Objekttyps erlangen
   I_PersistenceRetriever theRetriever =
      theStore.getRetriever(getPersistenceType(aSMICommand));
   // Retriever den SQL-Ausdruck aus dem
   // Kommando als Anfrage übergeben
   Object theResult =
      theRetriever.retrieve((String)aSMICommand.getValue(
         C_PersistenceSQLQualifier));
   // Ergebnis der Anfrage im Kommando hinterlegen
   aSMICommand.putValue(C_PersistenceResult, theResult);
}
/** Anfrage nach einem Objekt eines bestimmten Typs
mit einem bestimmten Attribut. */
public void queryByAttribute (SMICommand aSMICommand)
      throws PersistenceException {
   // Store aus der HttpSession erlangen
   I_Store theStore = getStore(aSMICommand);
   // I_PersistencePeer des verlangten Typs erlangen
   I_PersistencePeer thePeer =
      theStore.getPeer(getPersistenceType(aSMICommand));
   // Retriever des Typs erlangen
   I_PersistenceRetriever theRetriever = thePeer.getRetriever();
   // Peer über seine I_PersistenceType nach dem
   // benötigten PersistenceAttribute fragen
   PersistenceAttribute theSearchAttribute =
   thePeer.getType().attributeWithName(
      (String)aSMICommand.getValue(C_PersistenceSearchAttribute));
   // Wert des Attributs in echtes Objekt umwandeln
   Object theSearchValue =
      theSearchAttribute.getObjectForString(
         (String)aSMICommand.getValue(C_PersistenceSearchValue)) ;
   // Anfrage mit Wert des Attributs und Attribut starten
   Object theResult =
      theRetriever.retrieve(theSearchAttribute, theSearchValue);
   // Ergebnis der Anfrage im Kommando hinterlegen
   aSMICommand.putValue(C_PersistenceResult, theResult);
}
```

```
/** Anfrage nach assoziierten Objekten eines gegebenen Persistence-Objekts. */
public void queryForAssociation (SMICommand aSMICommand)
      throws PersistenceException {
   // Store aus der HttpSession erlangen
   I_Store theStore = getStore(aSMICommand);
   // Persistence anhand des ObjectIdentifiers aus dem Store holen
   I_Persistence theObject =
      theStore.persistenceWithKey(getObjectIdentifier(aSMICommand));
   // Name der Assoziation  aus dem Kommando erlangen
   String theAssociationName =
      (String)aSMICommand.getValue(C_PersistenceAssociationName);
   // Beschreibendes Assoziationsobjekt erlangen
   PersistenceAssociation theAssociation =
      theObject.getPeer().getType().getAssociationWithName(
         theAssociationName);
   // Assoziierte Objekte erfragen
   I_PersistenceReference theReference =
      theAssociation.getReferenceObject(theObject);
   Object theSearchResult;
   if (theAssociation.getAssociationType() ==
      C_PersistenceAssociation.TO_ONE) {
      theSearchResult =
         ((I_PersistenceToOneReference)theReference).getObject();
   }
   else {
      theSearchResult =
         ((I_PersistenceToManyReference)theReference).getObjects();
   }
   // Typ der assoziierten Objekte im Kommando hinterlegen
   aSMICommand.putValue(C_PersistenceType,
      theAssociation.getResultType());
   // Assoziierte Objekte im Kommando hinterlegen
   aSMICommand.putValue(C_PersistenceResult, theSearchResult);
}
/** Liest den ObjectIdentifier aus dem Kommando */
protected String getObjectIdentifier(SMICommand aSMICommand) {
   return (String)aSMICommand.getValue(C_PersistenceOID);
}
/** Liest den PersistenceType aus dem Kommando */
protected String getPersistenceType(SMICommand aSMICommand) {
   return (String)aSMICommand.getValue(C_PersistenceType);
}
/** Setzt die Werte eines persistenten Objektes. */
protected static void setPersistenceValues(
   I_Persistence aPersistence, SMICommand aSMICommand)
      throws PersistenceException {
   Enumeration e = aSMICommand.getKeys();
   String aValue;
   String aKey;
   while (e.hasMoreElements()) {
      aKey = (String)e.nextElement();
```

```
      // Falls ein Schlüssel mit einem bestimmten Präfix beginnt...
      if (aKey.startsWith(C_PersistenceAttributePrefix)
            && aKey.length() > C_PersistenceAttributePrefix.length()) {
         // ... wird sein Wert gelesen ...
         aValue = (String)aSMICommand.getValue(aKey);
         // ... das Präfix aus dem Schlüssel entfernt ...
         aKey = aKey.substring(C_PersistenceAttributePrefix.length());
         // ... und der Wert dem Attribut mit dem
         // Schlüsselnamen zugewiesen
         aPersistence.getPeer().getType().getAttributeWithName(
            aKey).setStringValue(aPersistence, aValue);
      }
   }
}


/** Gibt einen Store aus der Session zurück. */
protected I_SMIStore getStore(SMICommand aSMICommand) {
   return (I_SMIStore)aSMICommand.getSessionValue(C_Store);
}
} // Ende der Klasse
```

*Listing 11.3: Die Klasse* `StoreBrowserBean`

Sämtliche Methoden im `StoreBrowserBean` arbeiten mit dem Store. Um dies sicher tun zu können, müssen die Aufrufe synchronisiert und innerhalb einer Transaktion erfolgen. Daher überschreiben wir auch im `StoreBrowserBean` die `executeSMIEvent()`-Methode und legen um den Aufruf von `super.executeSMIEvent()` eine Transaktion. Außerdem stellen wir mit einer `synchronized`-Anweisung sicher, daß mehrere gleichzeitige Zugriffe auf den Store ausgeschlossen sind.

Im folgenden werden wir die benötigten Funktionen einzeln durchgehen. Da das Kommando `openStore` bereits durch das `StoreSessionBean` abgedeckt ist, müssen wir uns nicht weiter darum kümmern. Das Kommando `chooseStore` impliziert das Schließen des Stores – auch dieses Kommando ist also bereits abgedeckt. Als nächstes steht `showList` auf der Liste.

### 11.2.1 Kommando showList

Das Kommando soll eine Liste aller `Persistence`-Objekte eines Typs anzeigen. Dazu muß der verlangte Typ durch das `SMICommand` an die entsprechende Methode übergeben werden. Wir definieren, daß der Typ immer über den Schlüssel `C_PersistenceType` referenziert wird und kapseln den entsprechenden Zugriff auf das `SMICommand` in der Hilfsmethode `getPersistenceType(SMICommand)`.

Um alle `Persistence`-Objekte eines Typs aus dem Store zu erhalten, muß der `I_PersistenceRetriever` für den verlangten Objekttyp mit einer leeren SQL-Qualifizierung aufgerufen werden. In der Methode `queryByQualifier()` geschieht genau dies – zumindest solange im `SMICommand` keine Qualifizierung unter dem Schlüssel

C_PersistenceSQLQualifier hinterlegt wurde. Das Ergebnis der Suchanfrage wird unter dem Schlüssel C_PersistenceResult im SMICommand gespeichert, um in Folgekommandos darauf zugreifen zu können. So werden wir auch mit allen anderen Ergebnissen von Suchanfragen verfahren.

### 11.2.2 Kommando showObject

Das nächste Kommando – showObject – verlangt, ein einzelnes Objekt zu erlangen. Üblicherweise benötigt man dazu wieder den Typ C_PersistenceType sowie einen Schlüsselnamen C_PersistenceSearchAttribute und einen Schlüsselwert C_PersistenceSearchValue. Natürlich würde hier auch der eindeutige ObjectIdentifier reichen, da in ihm ja auch der Typ kodiert ist. Wir wollen jedoch der größeren Mächtigkeit wegen den allgemeineren Ansatz wählen. Die Funktionalität wird durch die Methode queryByAttribute(SMICommand) abgedeckt. Im einzelnen geschieht folgendes: Mit dem Objekttyp können wir über den Store einen Retriever und den PersistencePeer erlangen. Durch den PersistencePeer erhalten wir das PersistenceType-Objekt dieses Typs. Wie wir uns erinnern, beschreibt PersistenceType sämtliche Attribute und Assoziationen eines Persistence-Objekts (Kapitel 9.4.2.1).

Wir können also den PersistenceType nach einem PersistenceAttribute-Objekt nach dem Wert von C_PersistenceSearchAttribute fragen. Das erlangte PersistenceAttribute-Objekt wiederum läßt sich zusammen mit dem C_PersistenceSearchValue für eine Datenbankanfrage an den Retriever benutzen. Dazu muß jedoch vorher noch das durch C_PersistenceSearchValue referenzierte String-Objekt in ein Objekt des Attributtyps umgewandelt werden. Glücklicherweise ist dazu die Methode getObjectForString(String) des PersistenceAttributes in der Lage. Schließlich haben wir die beiden gewünschten Parameter beisammen und können den Retriever nach dem gesuchten Objekt fragen. Das Ergebnis dieser Anfrage wird wiederum unter dem Schlüssel C_PersistenceResult im SMICommand hinterlegt.

### 11.2.3 Kommando showAssociation

Genauso wie showObject ist showAssociation ein aufwendigeres Kommando. Um assoziierte Objekte eines Persistence zu erlangen, müssen wir uns zunächst das Objekt besorgen, dessen assoziierte Objekte verlangt werden. Daher muß der ObjectIdentifier des Ausgangsobjekts Teil der Anfrage sein. Mit der Hilfsmethode getObjectIdentifier(SMICommand) können wir diesen aus dem SMICommand filtern und damit den Store nach unserem Ausgangsobjekt fragen. Außerdem muß der Name der Assoziation unter dem Schlüssel C_PersistenceAssociationName des SMICommands hinterlegt sein. Mit diesem Namen können wir das PersistenceType-Objekt nach einem PersistenceAssociation-Objekt fragen. Das wiederum kennt ein Referenz-Objekt, das über die Methoden getObject() beziehungsweise getObjects() verfügt. Je nachdem, ob es sich um eine ToOne- oder ToMany-Assoziation handelt, wird eine der beiden Methoden ausgeführt und das Ergebnis unter dem Schlüssel C_PersistenceResult im SMICommand hinterlegt.

### 11.2.4 Kommando createObject

Als nächstes kümmern wir uns um das Kommando `createObject`. Um ein Objekt in den Store einzufügen, muß zunächst ein neues, leeres `Persistence`-Objekt erzeugt werden. Dies erledigt die Methode `newPersistence(String)` des Stores für uns. Als Parameter muß dabei der Typname des Objekts übergeben werden. Anschließend werden die Attribute gesetzt und die `create()`-Methode `des Persistence`-Objekts aufgerufen. Was sich einfach anhört, ist jedoch leider nicht so. Zunächst müssen wir eine Vereinbarung treffen, die festlegt, mit welchen Schlüsseln die Werte der einzelnen Attribute bezeichnet sind. Wir definieren, daß Attributschlüssel für ein `Persistence` immer mit dem Präfix `C_PersistenceAttributePrefix` beginnen, an das der Attributname angehängt wird. Das ermöglicht es uns, die gewünschten Attribute aus allen Parametern eines `SMICommands` herauszufiltern. Genau dies erledigt die Hilfsmethode `setPersistenceValues (I_Persistence, SMICommand)`. Darüber hinaus setzt sie das Attribut auch, nachdem sie es identifiziert hat. Nach dem Aufruf von `setPersistenceValues()` steht dem Aufruf von `Persistence.create()` nichts mehr im Wege. Das neu geschaffene Objekt hinterlegen wir wieder in `C_PersistenceResult`. Der gesamte Vorgang ist in `StoreBrowserBean.create(SMICommand)` kodiert.

### 11.2.5 Kommando updateObject

Das Kommando `updateObject` ist sehr ähnlich zu `createObject`. Der Unterschied besteht darin, daß nicht ein neues Objekt vom Store erzeugt werden muß, sondern lediglich ein bereits bestehendes Objekt modifiziert werden soll. Voraussetzung ist, daß der `ObjectIdentifier` des zu modifizierenden Objekts im `SMICommand` übergeben wird. Um auf diesen einfach zugreifen zu können, bedienen wir uns einmal mehr der Hilfsmethode `getObjectIdentifier(SMICommand)`. Mittels des erhaltenen `ObjectIdentifiers` wird der Store nach dem zugehörigen `Persistence` gefragt, mit `setPersistenceValues()` werden die geänderten Attribute gesetzt, und anschließend wird die `update()`-Methode des `Persistence` aufgerufen. Der beschriebene Vorgang wird von der Methode `StoreBrowserBean.update(SMICommand)` realisiert.

### 11.2.6 Kommando deleteObject

Das letzte der Modifikationskommandos ist `deleteObject`. Es ist in der Methode `delete(SMICommand)` kodiert und umfaßt lediglich zwei Zeilen. Über den `ObjectIdentifier` wird eine Referenz auf das zu löschende `Persistence` erlangt. Anschließend wird dessen `delete()`-Methode aufgerufen. Es erübrigt sich in diesem Fall, ein Ergebnis zu hinterlegen.

### 11.2.7 Sonstige Kommandos

Übrig bleiben die Kommandos `newObject` und `showTypes`. Bei genauem Hinsehen wird deutlich, daß – diese Kommandos – durch die Methode `display(SMICommand)` des `DisplayBeans` realisiert werden können (Kapitel 8.5).

## 11.3 Anzeige

Aufgrund der freien Verfügbarkeit entscheiden wir uns dafür, zur Anzeige GNUJSP von Vincent Partington (Kapitel 10.2) einzusetzen. Zum Aufruf der JavaServer-Pages, setzen wir ebenfalls das `DisplayBean` aus Kapitel 8.5 ein. Doch zunächst planen wir den Entwurf der Anzeige.

### 11.3.1 Code-Fragmente importieren

Um über einen Ansatzpunkt für Veränderungen der gesamten Anwendung zu verfügen, bedienen wir uns des Server-Side-Include-Mechanismus (SSI) der GNUJSP. Über ihn können wir häufig benötigte Code-Fragmente durch eine Anweisung in unsere JavaServer-Pages einfügen.

Als erstes definieren wir Kopf und Fuß aller Seiten. Die beiden entsprechenden Code-Fragemente `header.html` und `footer.html` sind in Listing 11.4 und Listing 11.5 abgedruckt.

```
<HTML>
<HEAD>
   <TITLE>StoreBrowser</TITLE>
</HEAD>
<BODY text="#800000" link="#000080" vlink="#000080" alink="#FF0000">
```

*Listing 11.4: Code-Fragment `header.html`.*

```
</BODY>
</HTML>
```

*Listing 11.5: Code-Fragment `footer.html`*

Genauso verfahren wir mit den `import`-Anweisungen. Um nicht in jeder Datei eine Liste von immer gleichen Anweisungen aufführen zu müssen, hinterlegen wir sie in der Datei `import.import` (Listing 11.6).

```
<%@ page import = "java.util.*" %>
<%@ page import = "javax.servlet.*" %>
<%@ page import = "de.webapp.Framework.Utilities.ReferenceVector" %>
<%@ page import = "de.webapp.Framework.Persistence.*" %>
<%@ page import = "de.webapp.Framework.SMI.SMICommand" %>
<%@ page import = "de.webapp.StoreBrowser.C_StoreBrowser" %>
```

*Listing 11.6: Die Datei `import.import`*

Jetzt können wir mit dem ersten Dialog beginnen.

### 11.3.2 Dialog zur Storeauswahl

Als Komfortmerkmal beim Storeauswahl-Dialog erwarten wir, daß der Name des zuletzt geöffneten Stores automatisch im Eingabefeld erscheint. Dazu müssen wir zunächst über das `SMICommand` die Session nach dem Wert für `C_StoreName` fragen. Standardmäßig hinterlegt das `DisplayBean` das `SMICommand` unter dem Namen `SMIEvent` als Attribut des `HttpServletRequests`. Dies nutzen wir, um das Kommando zu erhalten. Anschließend fragen wir nach dem Storenamen und hinterlegen ihn in der lokalen Variablen `theStorename`, die wir später als Wert des Eingabefeldes ausgeben. Alternativ hätten wir auch das Anfrage-Objekt nach der Session und diese nach dem Namen fragen können, da wir aber im weiteren hauptsächlich mit dem `SMICommand` arbeiten, wollen wir die Konsistenz wahren und hier nicht von dem sonstigen Kodierstil abweichen.

Weiterhin zu bemerken ist, daß das Kommando `openStore` als verstecktes Feld fest im Formular kodiert ist (Listing 11.7 und Abbildung 11.5).

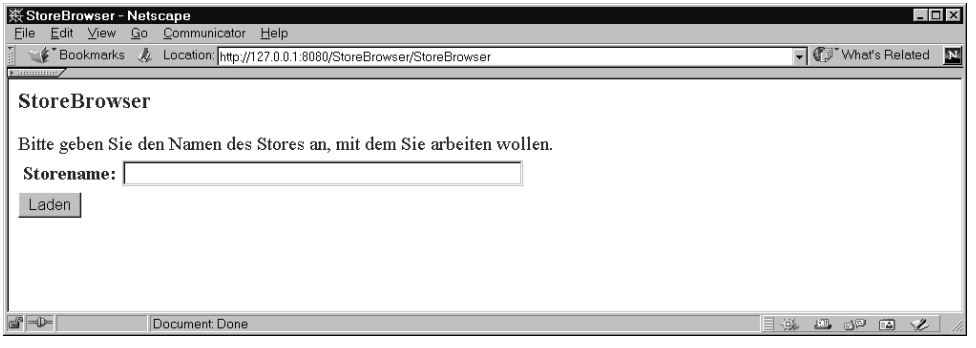


*Abbildung 11.5: Dialog zur Eingabe eines Storenamens*

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>
<%
// Kommando aus der Anfrage erlangen
SMICommand aSMICommand = (SMICommand)request.getAttribute("SMIEvent");
// Storenamen aus der Session besorgen
String theStorename =
   (String)aSMICommand.getSessionValue(C_StoreBrowser.C_StoreName);
%>
<h3>StoreBrowser</h3>
<form action="StoreBrowser" method="POST">
<input type="Hidden" name="Command" value="openStore">
Bitte geben Sie den Namen des Stores an, mit dem Sie arbeiten wollen.<br>
<table>
<tr>
<td><b>Storename:</b></td>
<td>
```

```
<input type="Text" name=" <%=C_StoreBrowser.C_StoreName%>" size="40"
value="<%=theStorename%>">
</td>
</tr>
</table>
<input type="Submit" value="Laden">
</form>
<%@ include file="footer.html" %>
```

*Listing 11.7: Dialog zur Eingabe eines Storenamen: `chooseStore.jsp`*

### 11.3.3 Typübersicht

Nach der Öffnung des Store, ist er in der Session hinterlegt. Im nächsten Dialog zeigen wir nun die Namen der Typen dieses Stores an (Listing 11.8 und Abbildung 11.6). Eine simple Iteration über die `PersistencePeers` und deren `PersistenceType`-Objekte erfüllt die Anforderung. Um den Link zur Listenansicht korrekt zu kodieren, müssen wir als Argument jeweils den `Persistence`-Typ übergeben. Als Schlüssel verwenden wir dazu wie vereinbart die Konstante `C_PersistenceType`.

Weiterhin verfügt der Dialog über einen Verweis zur Startseite.


*Abbildung 11.6: Dialog zur Typübersicht*

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>

<%
// Kommando aus der Anfrage erlangen
SMICommand aSMICommand = (SMICommand)request.getAttribute("SMIEvent");
%>
<!-Menü mit Link zum Storewahl-Dialog anzeigen -->
<h3><a href="StoreBrowser?Command=chooseStore">StoreBrowser</
a>:<%=aSMICommand.getSessionValue(C_StoreBrowser.C_StoreName)%></h3>
```

```
<ul>
<%
// Store aus der Session erlangen
I_Store theStore =
   (I_Store)aSMICommand.getSessionValue(C_StoreBrowser.C_Store);
// Iteration über die PersistencePeers
for(Enumeration e=theStore.getPeers();e.hasMoreElements();) {
   I_PersistencePeer thePeer = (I_PersistencePeer)e.nextElement();
   // Anzeige der Typnamen als Link
   out.println("<li><a href='StoreBrowser?Command=showList&"
      + C_StoreBrowser.C_PersistenceType + "="
      + thePeer.getType().getName()
      + "'>"
      + thePeer.getType().getName()
      + "</a><br>");
}
%>
</ul>
<%@ include file="footer.html" %>
```

*Listing 11.8: Dialog zur Typübersicht:* `types.jsp`

### 11.3.4 Listenansicht eines Objekttyps

Die Ausgabe der Listenansicht gliedert sich in drei Schritte (Listing 11.9 und Abbildung 11.7). Zunächst muß der Tabellenkopftext mit den Namen der Attribute und Assoziationen ausgegeben werden. Hierzu wird das Typbeschreibungsmodell des Stores benutzt. Anschließend wird über die Liste des `C_PersistenceResults` iteriert. Es werden alle Attribute ausgegeben, die Schlüssel-Attribute als Verweise zur Detailansicht des jeweiligen Objekts. Angehängt werden dann die Assoziationen, ebenfalls als Verweise. Schließlich muß nur noch ein Verweis zum Erzeugen eines Objekts ausgegeben werden.

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>
<%@ import = "java.net.URLEncoder" %>
<%
// Kommando aus der Anfrage erlangen
SMICommand aSMICommand = (SMICommand)request.getAttribute("SMIEvent");
// PersistenceType aus der Anfrage erlangen
String thePersistenceType =
   (String)aSMICommand.getValue(C_StoreBrowser.C_PersistenceType);
// Store aus der Session erlangen
I_Store theStore =
   (I_Store)aSMICommand.getSessionValue(C_StoreBrowser.C_Store);
// Resultat der Anfrage nach allen Objekten dieses Typs besorgen
ReferenceVector theResult =
   (ReferenceVector)aSMICommand.getValue(
       C_StoreBrowser.C_PersistenceResult);
%>
```

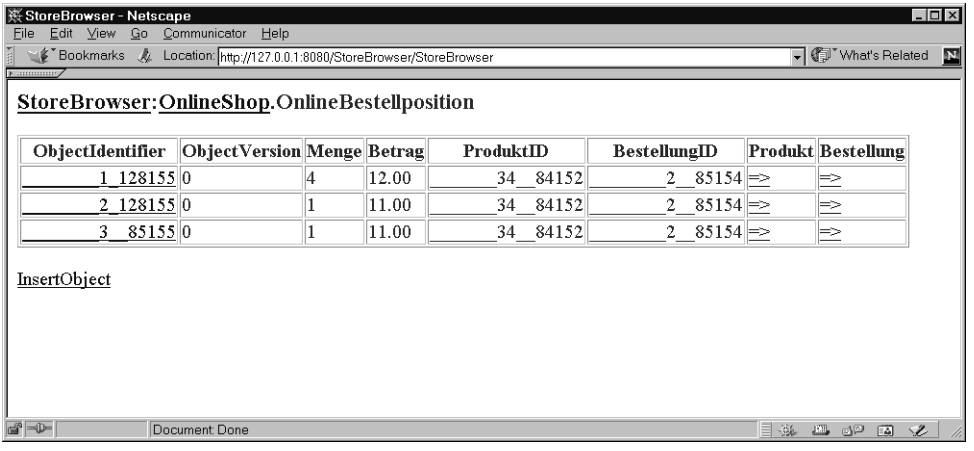


*Abbildung 11.7: Dialog zur Listenansicht eines Objekttyps:* `list.jsp`

```
<!--Menü mit Links zum Storewahl-Dialog und zur Typübersicht anzeigen -->
<h3><a href="StoreBrowser?Command=chooseStore">StoreBrowser</a>:<a
href="StoreBrowser?Command=showTypes"><%=aSMICommand.getSessionValue(C_StoreBrowser.C
_StoreName)%></a>.<%=thePersistenceType%></h3>
<table border=1>
<tr>
<%
/* normale Attribute ... */
Vector theAttributes =
   theStore.getPeer(thePersistenceType).getType().getAttributes();
for(Enumeration e = theAttributes.elements();e.hasMoreElements();){
   PersistenceAttribute theAttribute =
      (PersistenceAttribute) e.nextElement() ;
   out.println("<th>" + theAttribute.getName() + "</th>");
}
/* ...und die Assoziationen */
Vector theAssociations =
   theStore.getPeer(thePersistenceType).getType().getAssociation();
for(Enumeration e = theAssociations.elements();e.hasMoreElements();) {
   PersistenceAssociation theAssociation =
      (PersistenceAssociation) e.nextElement();
   out.println("<th>" + theAssociation.getName() + "</th>");
}
out.println("</tr>");
// Iteration über die Ergebnismenge
for(Enumeration e = theResult.elements(); e.hasMoreElements();) {
   I_Persistence theObject = (I_Persistence)e.nextElement();
   out.println("<tr>");
   for(Enumeration e1 = theAttributes.elements();e1.hasMoreElements();) {
      PersistenceAttribute theAttribute =
         (PersistenceAttribute)e1.nextElement();
      Object theSearchValue theObject.getValue(theAttribute);
```

```
      out.print("<td>");
      // Schlüsselattribute als Link darstellen
      if (theAttribute.isKeyAttribute()) {
         // Da der Schlüssel Leer- und Sonderzeichen enthalten kann,
         // muß der URL kodiert werden
         String theURL =
         URLEncoder.encode("StoreBrowser?Command=showObject"
         + "&_TemplateName=object.jsp&"
         + C_StoreBrowser.C_PersistenceSearchAttribute + "="
         + theAttribute.getName() + "&"
         + C_StoreBrowser.C_PersistenceSearchValue + "="
         + theSearchValue + "&"
         + C_StoreBrowser.C_PersistenceType + "="
         + thePersistenceType);
         out.print("<a href='" + theURL + "'>" + theSearchValue + "</a>");
      }
      else {
         out.print(theSearchValue);
      }
      out.println("</td>");
   }
   // Assoziationen als Link ausgeben
   for(Enumeration e1 = theAssociations.elements();e1.hasMoreElements();) {
      PersistenceAssociation theAssociation =
         (PersistenceAssociation) e1.nextElement() ;
      out.println("<td><a href='StoreBrowser?Command=showAssociation&"
         + C_StoreBrowser.C_PersistenceOID + "="
         + theObject.getObjectIdentifier() + "&"
         + C_StoreBrowser.C_PersistenceAssociationName + "="
         + theAssociation.getName() + "'>=></a></td>");
   }
   out.println("</tr>");
}
%>
</table>
<br>
<%
   // Verweis zum Erzeugen eines Objekts dieses Typs
   out.println("<a href='StoreBrowser?Command=newObject&"
      + C_StoreBrowser.C_PersistenceType + "="
      + thePersistenceType + "'>Objket erzeugen</a>") ;
%>
<%@ include file="footer.html" %>
```

*Listing 11.9: Dialog zur Listenansicht eines Objekttyps*

### 11.3.5 Detailansicht eines Objekttyps

Die Detailansicht eines Objekts zeigt seine Attribute und Werte sowie Typ und Store des Objekts (Listing 11.10 und Abbildung 11.8). Dazu wird über die Attribute des anzuzeigenden Typs iteriert und für den Fall, daß ein `C_PersistenceResult` existiert, der

jeweilige Attributwert angezeigt. Dabei spielen die Attribute `ObjectIdentifier` und `ObjectVersion` eine Sonderrolle. Im Gegensatz zu allen anderen Attributen dürfen sie nicht in editierbaren Eingabefeldern angezeigt werden. Ebenso dürfen ihre Schlüssel nicht mit dem für änderbare Attribute reservierten Präfix `C_PersistenceAttributePrefix` beginnen. Nach der Ausgabe der Attribute werden genau wie in der Listenansicht noch Verweise zu assoziierten Objekten angezeigt. Schließlich geben wir noch die Schalter zum Einfügen, Ändern und Löschen aus, die beiden letzteren jedoch nur, wenn tatsächlich ein Objekt angezeigt wurde.

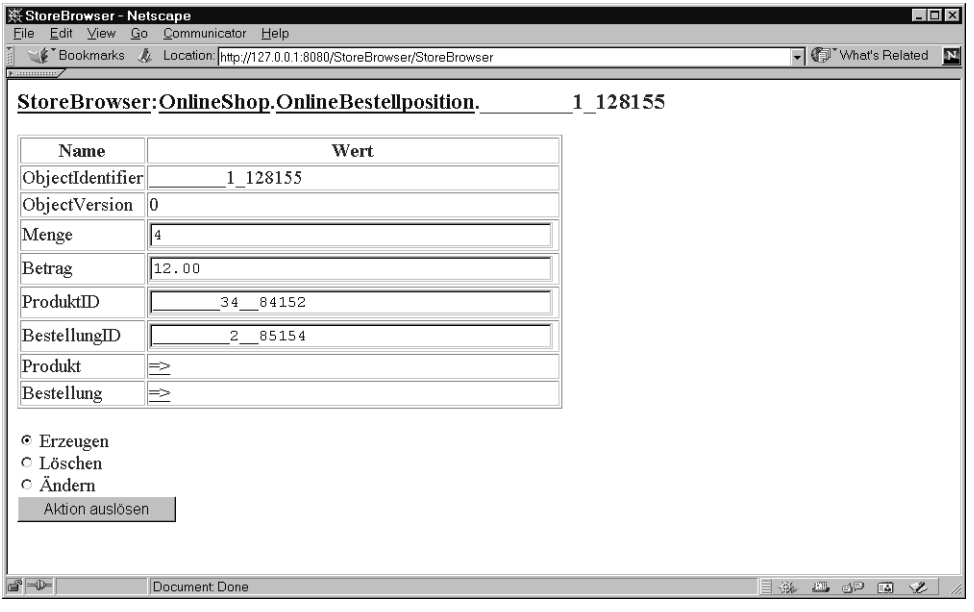


*Abbildung 11.8: Dialog zur Detailansicht eines Objekts*

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>

<%
// Kommando aus der Anfrage erlangen
SMICommand aSMICommand = (SMICommand)request.getAttribute("SMIEvent") ;
// Result aus dem Kommando holen
I_Persistence theResult =
   (I_Persistence)aSMICommand.getValue(C_StoreBrowser.C_PersistenceResult) ;
// Typ besorgen
String thePersistenceType =
   (String)aSMICommand.getValue(C_StoreBrowser.C_PersistenceType) ;
// Store aus der Session holen
I_Store theStore =
   (I_Store)aSMICommand.getSessionValue(C_StoreBrowser.C_Store) ;
```

```
// Wert für die Menüzeile festlegen
String theOID = null;
if (theResult != null)
   theOID = theResult.getObjectIdentifier();
else
   theOID= "undefined";
%>
<!--Menü mit Links zum Storewahl-Dialog, zur Typübersicht und Listenansicht anzeigen
-->
<h3><a href="StoreBrowser?Command=chooseStore">StoreBrowser</a>:<a
href="StoreBrowser?Command=showTypes"><%=aSMICommand.getSessionValue(C_StoreBrowser.C
_StoreName)%></a>.<a
href="StoreBrowser?Command=showList&_PersistenceType=<%=thePersistenceType%>"><%=theP
ersistenceType%></a>.<%=theOID%></h3>
<form action="StoreBrowser" method="post">
<table border=1>
<tr><th>Name</th><th>Wert</th></tr>
// Typ des Objekts für createObject hinterlegen
<input type="Hidden" name="<%=C_StoreBrowser.C_PersistenceType%>"
value="<%=thePersistenceType%>">
<%
// Ausgabe der Attribute
Vector theAttributes =
   theStore.getPeer(thePersistenceType).getType().attributes();
for(Enumeration e = theAttributes.elements();e.hasMoreElements();){
   PersistenceAttribute theAttribute =
     (PersistenceAttribute) e.nextElement();
   String theName;
   Object theValue = (Object) new String("");
   if (theResult != null) {
      theValue = theResult.getValue(theAttribute);
   }
   theName = theAttribute.getName();
   // Attributname
   out.println("<tr>\n<td>" + theName + "</td>");
   out.println("<td>");
   // Falls es sich um OID oder ObjectVersion handelt,
   // müssen diese Werte als versteckte Felder übergeben
   // werden und dürfen nicht editierbar sein.
   // Außerdem dürfen ihre Schlüssel nicht mit
   // C_PersistenceAttributePrefix beginnen, da diese Werte nicht
   // gesetzt werden dürfen.
   if (theName.equals("ObjectIdentifier") ||
      theName.equals("ObjectVersion")) {
      %>
      <input type="Hidden" name="_<%=theName%>" value="<%=theValue%>"><%=theValue%>
      <%
   } else {
      %>
      // Eingabefeld für Attribut anzeigen
      <input type="Text"
```

```
name="<%=C_StoreBrowser.C_PersistenceAttributePrefix%><%=theName%>"
value="<%=theValue%>" size=40>
      <%
   }
   %>
   </td>
   </tr>
<%
}
// Ausgabe der Assoziationen
Vector theAssociations =
   theStore.getPeer(thePersistenceType).getType().getAssociation();
for(Enumeration e = theAssociations.elements();e.hasMoreElements();) {
   PersistenceAssociation theAssociation =
      (PersistenceAssociation)e.nextElement();
   out.println("<tr>\n<td>" + theAssociation.getName() + "</td>");
   if (theResult != null) {
      out.println("<td><a href='StoreBrowser?Command=showAssociation&"
         + C_StoreBrowser.C_PersistenceOID + "="
         + theResult.getObjectIdentifier() + "&"
         + C_StoreBrowser.C_PersistenceAssociationName + "="
         + theAssociation.getName()+ "'>=></a></td>");
      out.println("</tr>");
   } else {
      out.println("<td>=></td>");
   }
}
%>
</table>
<br>
<input type="Radio" name="Command" value="createObject" checked>Erzeugen<br>
<%
// Schalter zum Löschen und Ändern nur anzeigen, wenn
// auch tatsächlich ein Objekt angezeigt wurde.
if (theResult != null) {
%>
<input type="Radio" name="Command" value="deleteObject">Löschen<br>
<input type="Radio" name="Command" value="updateObject">Ändern<br>
<%
}
%>
<input type="Submit" value="Aktion auslösen">
</form>
<%@ include file="footer.html" %>
```

*Listing 11.10: Dialog zur Detailansicht eines Objekts: `object.jsp`*

### 11.3.6 Anzeige assoziierter Objekte

Wie bereits erwähnt, referenzieren Assoziationen entweder eines oder viele Objekte. Dementsprechend bietet es sich an, je nach Art der Assoziation die Listenansicht oder die Detailansicht wiederzuverwenden. In der JavaServer-Page `association.jsp` (Listing 11.11) wird also nur festgestellt, welchen Typ das `C_PersistenceResult` hat, danach wird der entsprechende `RequestDispatcher` besorgt. Über `include()` wird schließlich das `SMI-Command` samt `C_PersistenceResult` an die darstellende JavaServer-Page vermittelt. Die JSP `association.jsp` hat also ausschließlich vermittelnde Funktion – die Arbeit führt `object.jsp` und `list.jsp` aus.

```
<%@ include file="import.import" %>

<%
SMICommand aSMICommand = (SMICommand)request.getAttribute("SMIEvent") ;
Object theSearchResult =
   aSMICommand.getValue(C_StoreBrowser.C_PersistenceResult);
if (theSearchResult instance of ReferenceVector) {
   %>
   <jsp:include page = "list.jsp" flush = "true" />
   <%
3 else {
   %>
   <%jsp:include page = "object.jsp" flush = "true" />
   <%
%>
```

*Listing 11.11: JavaServer-Page zur Anzeige von Assoziationen: `association.jsp`*

## 11.4 SMI-Definition und Konfiguration

Nachdem wir die benötigten Beans kodiert und die JavaServer-Pages entworfen haben, fehlt nur noch das Element, das beide zusammenführt. Diese Rolle erfüllt die SMI-Definitionsdatei (Listing 11.12). In ihr wird bestimmt, auf welches Kommando SMI mit welchem Methodenaufruf und welchen Parametern reagiert. Auch Folgekommandos sind hier festgelegt. In der Regel ist dies das Kommando `display`, das auf das `DisplayBean` abgebildet wird.

```
{
   // Definition der Standardwerte
   DefaultCommand = "display";
   Values = {
      _FollowUpCommand = "display";
      _TemplateName = "chooseStore.jsp";
   }
   // Definition der Listener
   Listener = {
      StoreBrowserBean = {
```

```
      Class = "de.webapp.StoreBrowser.StoreBrowserBean" ;
   },
   StoreSessionBean = {
      Class = "de.webapp.StoreBrowser.StoreSessionBean";
   },
   DisplayBean = {
      Class = "de.webapp.Framework.SMICommandListener.DisplayBean";
   }
};
// Definition der Kommandos
Commands = {
   // Anzeige
   display = {
      MethodName = "display";
      Listener = "DisplayBean" ;
   }
   // Anzeige der Persistence-Typen
   showTypes = {
      MethodName = "display";
      Listener = "DisplayBean";
      Values = {
         _TemplateName = "types.jsp"
      }
   }
   // Zeige Liste von Persistence-Objekten
   showList = {
      MethodName = "queryByQualifier";
      Listener = "StoreBrowserBean";
      Values = {
         _TemplateName = "list.jsp"
      }
   }
   // Zeige ein Persistence-Objekt
   showObject = {
      MethodName = "queryByAttribute";
      Listener = "StoreBrowserBean";
      Values = {
         _TemplateName = "object.jsp"
      }
   }
  // Anzeige einer Assoziation
   showAssociation = {
      MethodName = "queryForAssociation";
      Listener = "StoreBrowserBean";
      Values = {
         _TemplateName = "association.jsp"
      }
   }
   // Anzeige der Storeauswahl
   chooseStore = {
      MethodName = "closeStore";
```

```
        Listener = "StoreSessionBean";
      }
      // Öffnen des Stores
      openStore= {
        MethodName = "openStore";
        Listener = "StoreSessionBean" ;
        Values = {
          _TemplateName = "types.jsp"
        }
      }
      // Eingabedialog eines Persistence-Objekt
      newObject= {
        MethodName = "display";
        Listener = "DisplayBean" ;
        Values = {
          _TemplateName = "object.jsp"
        }
      }
     // Erzeugen eines Persistence-Objekt
      createObject = {
        MethodName = "create";
        Listener = "StoreBrowserBean" ;
        Values = {
          _TemplateName = "object.jsp"
        }
      }
      // Löschen eines Persistence-Objekt
      deleteObject = {
        MethodName = "delete";
        Listener = "StoreBrowserBean";
        Values = {
          _TemplateName = "object.jsp"
        }
      }
      // Änderung eines Persistence-Objekt
      updateObject = {
        MethodName = "update";
        Listener = "StoreBrowserBean";
        Values = {
          _TemplateName = "object.jsp"
        }
      }
   }
}
```

*Listing 11.12: Die SMI-Definitionsdatei des StoreBrowsers* `StoreBrowser.smi`

Um den StoreBrowser nutzen zu können, müssen wir ihn jetzt nur noch bei der Servlet-Engine registrieren. Da wir Eigenschaften des Servlet-API 2.1 nutzen, muß es sich dabei um eine entsprechende Engine handeln. Die Konfiguration für jo! sieht aus wie in Listing 11.13 beschrieben.

```
# StoreBrowser
servlet.StoreBrowser.code=de.webapp.Framework.SMI.SMIServlet
servlet.StoreBrowser.initArgs=EventSwitch=StoreBrowser,Context=StoreBrowser
servlet.StoreBrowser.aliases=/StoreBrowser/StoreBrowser
```

*Listing 11.13: StoreBrowser-Eintrag in der Datei `servlet.properties` von jo!*

Mit dem Aliasnamen für den StoreBrowser haben wir gleichzeitig festgelegt, wo sich die JavaServer-Pages befinden müssen – nämlich im Verzeichnis `/StoreBrowser/`.

Nach dem Start der Servlet-Engine steht der Browser unter dem URL `http://<hostname>:<port>/StoreBrowser/StoreBrowser` zur Verfügung!

# 12 OnlineShop

Als zweites Beispiel für das WebApp-Framework werden wir einen OnlineShop entwickeln. Dabei werden wir uns auf die Umsetzung der Kernfunktionalität beschränken und keine Adminstrationskomponenten oder sichere Bezahlung realisieren.

In einem OnlineShop präsentiert der Händler seine Produktpalette seinen Kunden. Der Kunde kann aus dem Angebot Produkte auswählen, sie in seinen persönlichen virtuellen Warenkorb legen und auch wieder herausnehmen. Die Bestellung kann er per Mausklick aufgeben. Dazu muß der Kunde dem Händler seine Zahlungsweise und die Lieferadresse angeben. Damit diese Angaben nicht bei jedem Einkauf erneut gemacht werden müssen, verfügt der Händler über eine Kundendatei. Um auf die Daten in der Datei zurückgreifen zu können, muß sich der Kunde nur noch mit seiner E-Mail-Adresse und einem Geheimwort ausweisen.

Wir beschreiben in diesem Kapitel, die Realisierung eines OnlineShops mit den Mitteln des WebApp-Frameworks.

Die Kernfunktionen unserer OnlineShop-Anwendung sind:

- Anzeige der Geschäftsinformationen
- Anzeige des Produktkatalogs
- Anlegen eines Warenkorbs für jeden Kunden
- Anzeige des Warenkorbs
- Nachträgliche Änderung der Bestellung
- Abschicken und Speichern der Bestellung
- Erfassen eines neuen Kundens
- Anmelden und Abmelden eines Kunden

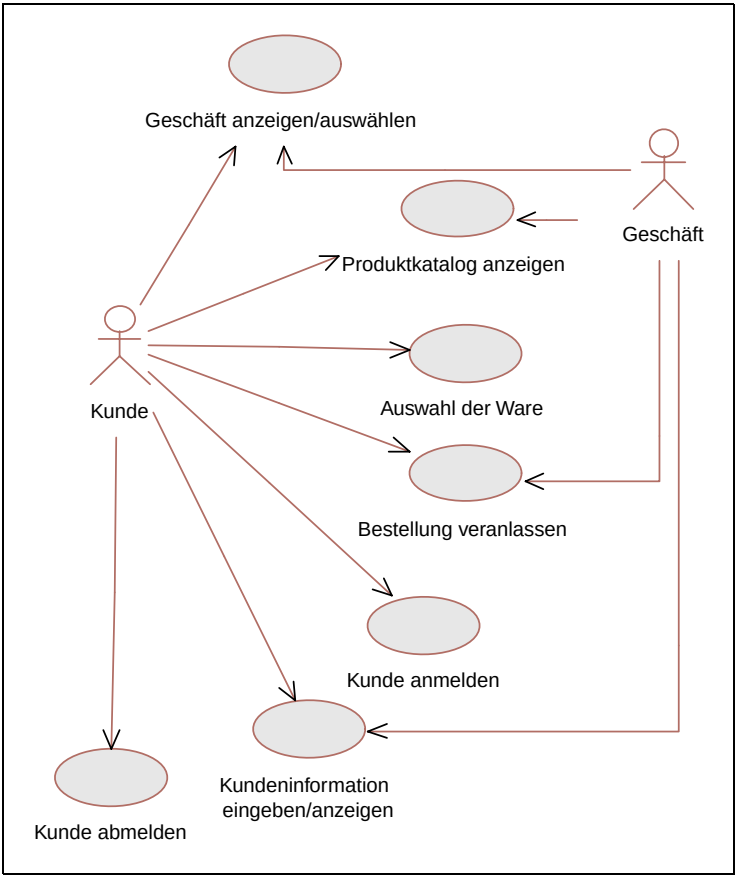


*Abbildung 12.1: Anwendungsfälle des OnlineShop*

## 12.1 Abläufe

Nach dem Aufruf des OnlineShops werden dem Nutzer in der linken Bildschirmhälfte ein Funktionsmenü und in der rechten allgemeine Informationen (Name, Inhaber, Beschreibung etc.) über den Shop präsentiert (Abbildung 12.2). Ist der Kunde bereits angemeldet, erscheint sein Name. Hat er außerdem schon Waren in seinen Korb gelegt, wird ein entsprechender Hinweis angezeigt.

Im Funktionsmenü befinden sich Einträge zum Aufruf des Produktkatalogs, der Anzeige des aktuellen Warenkorbs, der Kundendaten sowie zum An- und Abmelden des Kunden. Mit dem Eintrag Start können wir zudem jederzeit zur Startseite des Shops zurückkehren.

Hat der Kunde den Produktkatalog gewählt, wird ihm eine bestimmte Anzahl an Produkten pro Seite angezeigt. Diese Anzahl soll konfigurierbar sein. Übersteigt die Anzahl der Produkte im Katalog die maximale Anzahl an Produkten pro Seite, kann der Kunde vor- und zurückblättern. Entscheidet er sich dabei für ein Produkt, muß er die gewünschte Menge angeben. Das Produkt wird dann automatisch in seinem virtuellen Warenkorb deponiert. Nach jedem Einfügen eines Produkts in den Warenkorb, kann der Kunde entweder seine Bestellung aufgeben oder zum Katalog zurückkehren.

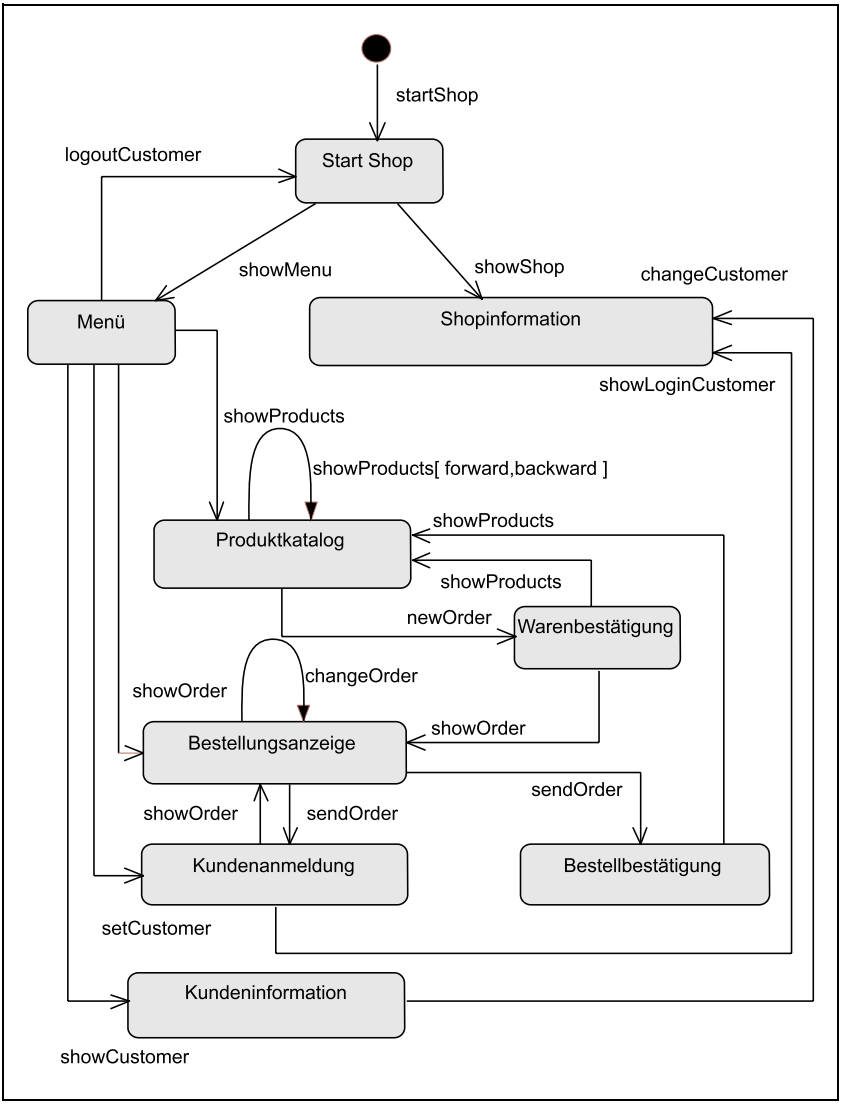


*Abbildung 12.2: Ablaufdiagramm des OnlineShops*

Über das Funktionsmenü kann der Kunde jederzeit seinen Warenkorb einsehen. Die einzelnen Bestellpositionen können dabei leicht verändert werden. So läßt sich die Bestellmenge erhöhen oder verringern. Natürlich kann man ein Produkt auch ganz aus dem Warenkorb entfernen.

Um weitere Produkte in den Korb zu legen, kann der Kunde über das Funktionsmenü jederzeit wieder zum Katalog wechseln.

Die Bestellung kann nur abgeschickt werden, wenn ein Kunde angemeldet ist. Dies kann entweder durch die Funktion Anmelden geschehen oder wird beim Abschicken der Bestellung angefordert. Die Anmeldung erfolgt dadurch, daß der Kunde seine E-Mail-Adresse und ein Geheimwort eingibt. Die einzelnen Liefer- und Zahlungsparameter können in einem getrennten Dialog eingegeben und jederzeit durch den Kunden geändert werden. Der Warenkorb geht bei der Änderung der Kundendaten nicht verloren.

Alle Kommandos des OnlineShops sind in der Tabelle 12.1 genauer beschrieben.

| Kommando | Parameter | Bedeutung |
|---|---|---|
| `startShop` | - | Starte den OnlineShop. Es werden die Anweisungen `showMenu` und `showShop` in zwei getrennten Frames angezeigt. |
| `showMenu` | - | Zeige das Menü mit Verweisen auf die Kommandos `showProducts`, `showOrder`, und `showCustomer`, `loginCustomer`, `logoutCustomer` und `startShop` an. |
| `showShop` | - | Zeige Informationen zum OnlineShop an: Name des Besitzers, E-Mail-Adresse und Verweise auf weitere Informationen. |
| `showProducts` | Offset | Zeige Produktkatalog. Der Parameter `Offset` gibt an, ab welchem Produkt die Anzeige beginnen soll. Die Anzahl der Produkte einer Katalogseite wird in der Servlet-Konfiguration eingetragen. |
| `newOrder` | Objektschlüssel des Produkts, Anzahl der Produkte dieser Bestellung | Erzeuge eine Bestellung mit der neuen Bestellposition. Falls schon eine Bestellung aktiv ist, wird die Ware dieser hinzugefügt. Die Menge und der Preis werden in der Position hinterlegt. |
| `showOrder` | - | Zeige die aktuelle Bestellung, soweit eine im Warenkorb vorhanden ist. Anzeige des Gesamtwerts der Bestellung. |
| `changeOrder` | Menge | Die Menge der Bestellung kann verändert werden. Die Position wird bei Auswahl der Menge 0 gelöscht.. Der Preis wird mit der Menge neu berechnet. |

*Tabelle 12.1: Kommandos des OnlineShops*

| Kommando | Parameter | Bedeutung |
|---|---|---|
| `sendOrder` | Bemerkung | Nehme aktuelle Bestellung aus der Session und erzeuge Bestellung und Position in der Datenbank. Wenn kein Kunde angemeldet ist, verlange die Anmeldung eines Kunden. |
| `showCustomer` | - | Anzeige eines Eingabedialogs für Kundenattribute. Falls ein Kunde schon angemeldet ist, werden die Attribute zur Änderung angeboten. |
| `changeCustomer` | Kundenattribute | Falls der Kunde noch nicht registiert ist, wird ein neuer Kunde angelegt, sonst werden die Attribute des vorhandenen Kunden geändert. Bevor eine Änderung in die Datenbank geschrieben wird, werden die Pflichtfelder geprüft. |
| `showLogin-Customer` | - | Anzeige des Anmeldungsdialogs. |
| `setCustomer` | E-Mail und Geheimwort | Anmelden eines Kunden. Ein aktiver Warenkorb wird übernommen. |
| `logoutCustomer` | - | Abmelden eines Kunden. Ein evtl. aktiver Warenkorb verfällt |

*Tabelle 12.1: Kommandos des OnlineShops (Fortsetzung)*

## 12.2 Design der Klassen

Aus der Definition der Abläufe des OnlineShops ergeben sich zwei Arten von Klassen: die `I_SMICommandListener` zur Realisierung der Abläufe und die Informations-Objekte, die dem OnlineShop zugrundeliegen.

### 12.2.1 Informations-Objekte

Zunächst einmal müssen wir die beteiligten Informations-Objekte bestimmen. Dies sind: der OnlineShop, seine Produkte, die Kunden und ihre Bestellungen mit den jeweiligen Bestellpositionen. Dabei gilt:

- Jeder OnlineShop hat eine Anzahl von Produkten.
- Jeder Kunde ist genau einem Shop zugeordnet.
- Jedem Kunden ist eine Anzahl von Bestellungen zugeordnet.
- Jede Bestellung besitzt eine Liste von Bestellpositionen von Produkten ( Abbildung 12.3).

Als Basis dieser einfachen Geschäftsobjekte verwenden wir die `Persistence`-Objekte aus Kapitel 9. Da bei keinem Objekt spezielle Funktionen realisiert werden müssen, verzichten wir auf die Kodierung spezieller Klassen und greifen auf die Klasse `Persi-`

`stenceGeneric` zurück. Sie bietet den Vorteil, daß wir die Attribute und Assoziationen der Geschäftsobjekte durch eine Konfigurationsdatei definieren können.

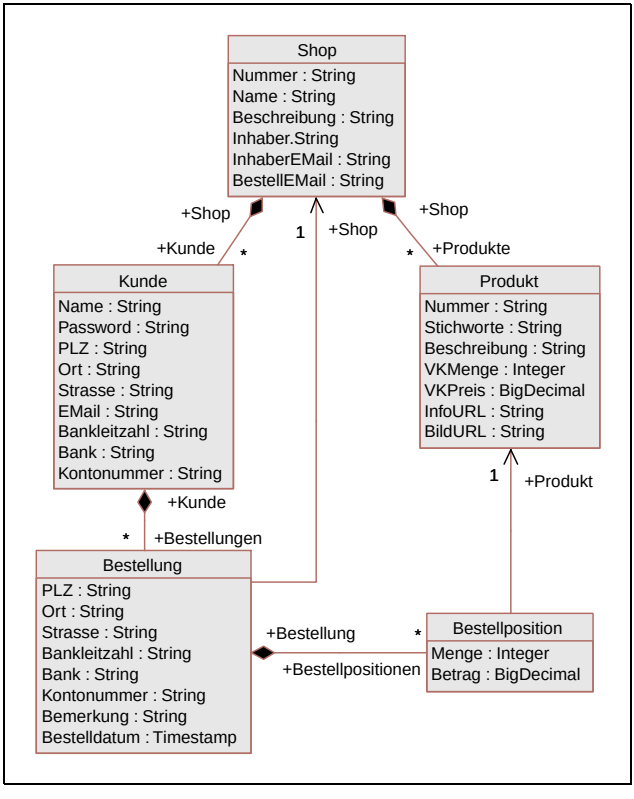


*Abbildung 12.3: Geschäftsobjekte des OnlineShops*

Jeder `OnlineShop` besitzt eine eindeutige Nummer. Dadurch können für jeden Shop eigene unabhängige Komponenten und Datenbankverbindungen geschaffen werden. Obwohl wir uns in diesem Beispiel auf einen Shop beschränken, ist so die Möglichkeit zur einfachen Erweiterung bereits eingebaut.

Beim Anlegen einer Bestellung wird die aktuelle Lieferadresse und Bankverbindung eines Kundens in der Bestellung hinterlegt. Jeder Bestellung kann zudem noch eine Bemerkung des Kunden hinzugefügt werden.

Das Ändern der Kundendaten bewirkt automatisch einen Abgleich mit der aktuellen Bestellung, die noch nicht versandt ist. Jede Bestellposition einer Bestellung wird durch die Menge und den Preis gekennzeichnet. Das Produkt selbst verfügt neben Preis und Beschreibung über ein Bild und einen URL als Verweis auf weitere Informationen.

### 12.2.2 Komponenten des OnlineShops

Beim ersten Start des OnlineShops muß eine Verbindung zur Datenbank hergestellt werden. Wir benutzen dazu natürlich unseren Store (Kapitel 9). Da alle weiteren Beans des OnlineShops Zugriff auf die Informations-Objekte haben sollen, wird der Store im `OnlineShopContext` hinterlegt (Abbildung 12.4). Aus der Ablaufdefinition ist ersichtlich, daß es mehrere Funktionsgruppen pro Kunde, Produkt und Bestellung gibt. Für jede dieser Gruppen definieren wir daher ein eigenes Bean.

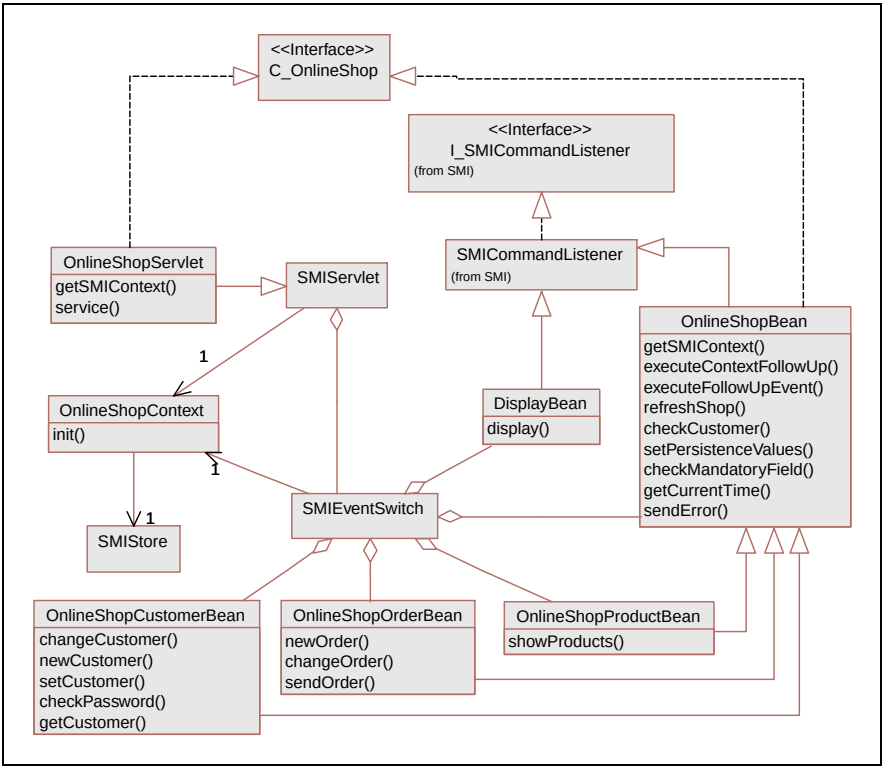


*Abbildung 12.4: Klassendiagramm der Komponenten des OnlineShops*

Als Oberklasse unserer Beans dient das `OnlineShopBean`. Mit ihren Ableitungen `OnlineShopCustomer`, `OnlineShopOrder` und `OnlineShopProduct` und dem Anzeige-Bean `DisplayBean` bildet sie den Kern der OnlineShop-Anwendung. Das `OnlineShopServlet` hat die Aufgabe, alle Zugriffe auf den OnlineShop mit einem Transaktionsrahmen zu versehen und die nötige Synchronisierung vorzunehmen ( Listing 12.1).

Die Realisierung des Warenkorbs erfolgt über das `HttpSession`-Objekt. Für jeden Nutzer des OnlineShops wird in seiner Session ein Warenkorb und sein Kunden-Objekt gehalten. Der Produktkatalog wird im `OnlineShopContext` – und somit für alle Kunden des Shops zugänglich – hinterlegt (Abbildung 12.5).

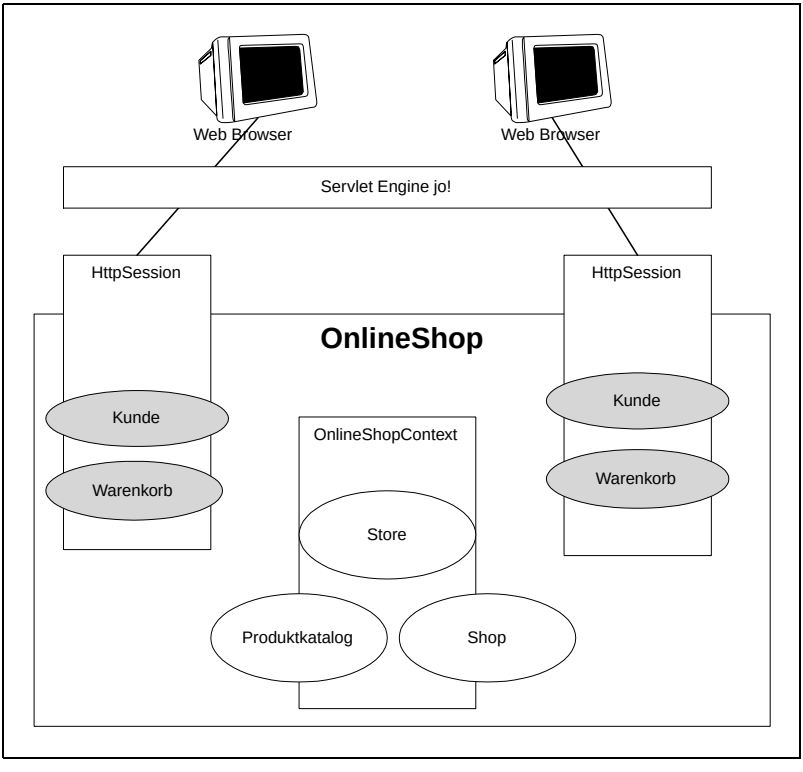


*Abbildung 12.5: Ablage wichtiger Informations-Objekte des OnlineShops*

### 12.2.3 OnlineShopServlet

Das `OnlineShopServlet` ist eine Ableitung des `SMIServlets` (Kapitel 8). Es besteht lediglich aus den beiden Methoden `getSMIContext()` und `service()` (Listing 12.1). Die Methode `getSMIContext()` dient dabei als Hilfsmethode, um den `I_SMIContext` der Anwendung zu erlangen. Dort befindet sich die Instanz eines `SMIStores`, der zentralen Datenbankverbindung des OnlineShops.

Die Methode `service()` umrahmt jede Anfrage eines Kunden mit einem exklusiven Zugriff auf den `Store`. Der Transaktionsrahmen `begin() - commit()/rollback()` sichert alle Operationen ab. Der Aufruf der Kommandos wird in der `service()`-Methode der Oberklasse `SMIServlet` ausgeführt.

```
package de.webapp.OnlineShop ;

import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
import de.webapp.Framework.ConfigManager.* ;
import de.webapp.Framework.SMI.* ;
import de.webapp.Framework.StoreFactory.StoreFactory ;
import de.webapp.Framework.Persistence.* ;
import de.webapp.Framework.Utilities.ReferenceVector;
import de.webapp.Framework.Log.* ;

/** Initialisierung des Stores */
public class OnlineShopServlet extends SMIServlet
   implements C_OnlineShop
{
/** Hole SMIContext der Anwendung über den Switch. */
public I_SMIContext getSMIContext()
   { return getSMIEventSwitch().getSMIContext(); }
/** Bilde Transaktionsrahmen und Sicherung des Store bei jedem Zugriff. */
public void service (ServletRequest req, ServletResponse res)
   throws ServletException, IOException {
   I_Store theStore = (I_Store) getSMIContext().getValue(C_Store) ;
   synchronized(theStore) {
      try {
         theStore.begin();
         super.service(req, res);
         theStore.commit() ;
      } catch (Exception e) {
      try {
         theStore.rollback() ;
      } catch (PersistenceException pe) {
          pe.printStackTrace() ;
      }
      if (e instanceof ServletException) throw (ServletException)e;
      if (e instanceof IOException) throw (IOException)e;
      else throw new ServletException(e.toString()) ;
      }
   }
}
} // Ende der Klasse
```

*Listing 12.1: Klasse* `OnlineShopServlet`

### 12.2.4 Die Schnittstelle C_OnlineShop

Zur Definition aller Konstanten des `OnlineShops` dient die Schnittstelle `C_OnlineShop`. Es werden Definitionen für Kommandos (`C_DisplayCommand`), Parameter der Konfigurationen (`C_ShopNummer`), Übergabeparameter von Kommandos (`C_FollowUpCommand`, `C_CustomerMail`), Ausgabefelder (`C_BestellParameter`) oder Pflichtfelder der Eingabe

(C_CustomerLoginFields) sowie Fehlermeldungen (C_CustomerNotSet) getroffen. Die Schnittstelle und alle Konstanten sind dabei mit dem Präfix C_ gekennzeichnet.

```
package de.webapp.OnlineShop ;

public interface C_OnlineShop {
// #### Anzeigekommandos
/** Kommando für die Anzeige einer Fehlerseite */
public static final String C_ErrorDisplay = "display";
/** Kommando für die Anzeige */
public static final String C_DisplayCommand = "display";
/** Parameter für Folgekommando */
public static final String C_FollowUpCommand = "_FollowUpCommand";
/** Parameter für JSP oder HTML-Seite */
public static final String C_Template = "_TemplateName";
/** Name der Fehlerseite */
public static final String C_ErrorPage = "ShowError.jsp";
/** Name der Login-Seite */
public static final String C_LoginCustomer = "kundeLogin.html";
/** FollowUp-Seite nach dem Login */
public static final String C_LoginCustomerRestartOrder =
   "LoginCustomer.html";

// #### Parameter-Konstanten
/** Schlüssel zur ShopNummer */
public static final String C_ShopNummer = "_Shopnummer" ;
/** Default-Storename */
public static final String C_DefaultStorename = "OnlineShop" ;
/** Schlüssel zum Namen des Stores */
public static final String C_StoreName = "_Storename" ;
/** Schlüssel zum SMIContext */
public static final String C_Context = "_OnlineShopContext" ;
/** Nachricht für Fehlerseite */
public static final String C_Message = "_OnlineShopMessage" ;
/** Parameter für Fehlerseite */
public static final String C_Arguments = "_OnlineShopAgrumentArray" ;

// #### Session Parameter
/** Aktuelle Bestellung einer Session */
public static final String C_Order = "de.webapp.OnlineShop.Order" ;
/** Aktueller Kunde */
public static final String C_Customer = "de.webapp.OnlineShop.Customer" ;
/** Produkt der Bestellung */
public static final String C_ProductID = "_ProductID" ;
/** Menge der Bestellung */
public static final String C_Menge = "_Menge" ;
/** Mailadresse des Kunden */
public static final String C_CustomerMail = "_PAEMail" ;
/** Geheimwort des Kunden */
public static final String C_CustomerPassword = "_PAPassword" ;
/** Nummer der Position der Bestellung, die gelöscht werden soll */
```

```
public static final String C_DeleteBestellPosition = "_DeletePos" ;

/** Seitengröße */
public static final String C_PageSize = "_PageSize" ;
/** Versatz der Produktanzeige */
public static final String C_Offset = "Offset" ;

/** Präfix für Formularfelder, die auf Persistence-Attribute
    abgebildet werden */
public static final String C_PersistenceAttributePrefix = "_PA";

// #### Applikationsschlüssel
/** Schlüssel des Stores in der Applikation OnlineShop */
public static final String C_Store = "de.webapp.OnlineShop.Store" ;
/** Schlüssel des Shops in der Applikation OnlineShop */
public static final String C_Shop = "de.webapp.OnlineShop.Shop" ;
/** Schlüssel der Produkte des Shops in der Applikation OnlineShop */
public static final String C_Products = "de.webapp.OnlineShop.Products" ;

// #### Fehler
/** Kunde nicht gesetzt */
public static final String C_CustomerNotSet =
   "Der Kunde ist nicht gesetzt!" ;
/** Bestellung nicht gesetzt */
public static final String C_OrderNotSet =
   "Es liegt keine Bestellung vor!" ;
/** Produkt der Bestellposition fehlt */
public static final String C_ProductFailed =
   "In Bestellposition fehlt ein Produktzuordnung" ;
/** Bitte Vielfaches der VK Menge setzen */
public static final String C_Vielfaches =
   "Bitte Vielfaches der VK Menge setzen" ;

// #### Felder und Parameter
/** Attribute für den Parameteraustausch zwischen Bestellung und Kunde */
public static final String C_BestellParameter[] = new String[]
   { "Bankleitzahl", "Bank", "Kontonummer", "PLZ", "Ort", "Strasse" } ;
/** Pflichtfelder des Kunden anmelden */
public static final String C_CustomerLoginFields[] = new String[]
   { "EMail", "Password" } ;
/** Pflichtfelder des Kunden */
public static final String C_CustomerFields[] = new String[]
   { "Name", "Password", "EMail", Bankleitzahl", "Bank",
     "Kontonummer", "PLZ", "Ort", "Strasse" } ;
/** Eingabefelder */
public static final String C_CustomerAddressFields[][] = new String[][]
   { {"Bankleitzahl", "Bankleitzahl"},
     {"Bank", "Bank"},
     {"Kontonummer", "Kontonummer"},
     {"PLZ", "PLZ"},
     {"Ort", "Ort"},
```

```
     {"Strasse", "Strasse"}
   } ;
/** Anzeige der Eingabefelder des Kundens */
public static final String C_CustomerInputFields[][] = new String[][]
   { {"Name *", "Name", "50", "Text" },
     {"EMail *", "EMail", "50", "Text" },
     {"Password *", "Password", "8", "Password" },
     {"PLZ *", "PLZ", "5", "Text" },
     {"Ort *", "Ort", "50", "Text" },
     {"Strasse *", "Strasse", "50", "Text" },
     {"Bankleitzahl *", "Bankleitzahl", "10", "Text" },
     {"Bank *", "Bank", "50", "Text" },
     {"Kontonummer *", "Kontonummer", "10", "Text" },
   } ;

} // Ende der Schnittstelle
```

*Listing 12.2: Schnittstelle `C_OnlineShop`*

### 12.2.5 OnlineShopContext

Bei der Initialisierung des `OnlineShopContext` wird die Datenbankverbindung durch die Konfiguration und Initialisierung des Stores hergestellt (Listing 12.3 und Abschnitt 11.2). Dazu wird aus der Konfiguration des Kontexts der Name des Stores, die Seitengröße des Katalogs und die Nummer des Shops gelesen. Das Shop-Objekt wird aus dem Store geladen und mit dem Store im Kontext hinterlegt. Dort sind sie leicht für alle Beans zugänglich.

```
package de.webapp.OnlineShop ;

import java.io.IOException;
import java.util.*;
import de.webapp.Framework.ConfigManager.* ;
import de.webapp.Framework.SMI.* ;
import de.webapp.Framework.StoreFactory.StoreFactory ;
import de.webapp.Framework.Persistence.* ;
import de.webapp.Framework.Log.* ;

public class OnlineShopContext extends SMIContext
   implements C_OnlineShop {

// Initialisiere Store des OnlineShop und wähle Shop aus.
public void init(String aContextName, Hashtable aValues)
      throws SMIException {
   super.init(aContextName, aValues);
   // Hole Store.
   String theStorename = (String)getValue(C_StoreName) ;
   if(theStorename == null)
      theStorename = C_DefaultStorename ;
   I_Store theStore = null ;
   try {
```

```
      ConfigManager.getConfigManager();
      Configuration theStoreConfig =
         ConfigManager.getConfigManager().getConfiguration(
            aContextName,theStorename,true);
     // Initialisierung des Stores
      theStore = StoreFactory.getInstance(theStoreConfig) ;
   } catch (PersistenceException pe) {
      throw new SMIException(pe.toString()) ;
   }
   // Store in den Kontext legen.
   putValue(C_Store,theStore) ;
   // Hole Shop.
   String theShopNummer = (String)getValue(C_ShopNummer) ;
   if(theShopNummer == null)
      throw new SMIException("ShopNummer nicht vorhanden!") ;
   // Anzahl der gleichzeitig sichtbaren Produkte
   String thePageSize = (String) getValue(C_PageSize) ;
   if(thePageSize == null)
      thePageSize = "10" ;
   putValue(C_PageSize,new Integer(thePageSize)) ;
   // Hole den Shop aus der Datenbank.
   synchronized(theStore) {
      try {
         theStore.begin() ;
         // suche Shop
         I_Persistence theShop =
            theStore.persistenceWithKey(
               theShopNummer,"OnlineShop","Nummer") ;
         if(theShop == null)
            throw new SMIException("Shop with Number "
               + theShopNummer + " not found ") ;
         // Shop im Kontext ablegen.
         putValue(C_Shop,theShop) ;
         theStore.commit() ;
      } catch (Throwable e) {
         try {
            theStore.rollback() ;
         } catch (PersistenceException pe ) {
             throw new SMIException(
                 "Shop konnte nicht gefunden werden!", pe) ;
         }
         throw new SMIException(e) ;
      }
   }
}
} // Ende der Klasse
```

*Listing 12.3: Klasse `OnlineShopContext`*

### 12.2.6 OnlineShopBean

Die Oberklasse `OnlineShopBean` hat zwei Funktionen: Zum einen definiert sie für alle Ableitungen einige Hilfsfunktionen. Zum anderen setzt sie einige allgemeine Kommandos um.

So realisiert die Methode `refreshShop(SMICommand)` eine komplette Reinitialisierung des OnlineShops. Mit Hilfe der Methode `sendError(SMICommand)` können alle Beans eine Fehlermeldung absetzen. Die Methode `sendError(SMICommand, Object[])` erlaubt sogar, die Übergabe einer beliebigen Liste von Parametern als Fehlerbeschreibung. Eine schöne Fehlerbehandlung sieht sicherlich etwas anders aus, aber ein Anfang ist gemacht.

Die Methode `executeFollowUpEvent(SMICommand)` führt zur Ausführung eines Nachfolgekommandos, falls dieses mit dem Parameter `C_OnlineShop.C_FollowUpCommand` gesetzt ist (s. Kapitel 11). Die Methode `executeContextFollowUp(SMICommand)` hinterlegt zusätzlich noch den `OnlineShopContext` als Parameter im Kommando.

Die Hilfsfunktionen stehen allen Ableitungen der Klasse `OnlineShopBean` zur Verfügung. So kann mit der Methode `checkMandadoryField()`geprüft werden, ob mit dem `SMICommand` tatsächlich die erwarteten Argumente einer Dialogeingabe übergeben wurden. Die Methode `setPersistenceValues()` setzt die Argumente eines Kommandos als Attribute eines `Persistence`-Objekts. Sowohl `checkMandadoryFields()` als auch `setPersistenceValues()` nutzen jeweils nur Parameter, deren Schlüssel mit dem Präfix `C_PersistenceAttributePrefix` beginnt.

Zu guter Letzt sind noch die Methoden `checkCustomer()` und `getCurrentTime()`. Die erste sorgt dafür, daß ein angemeldeter Kunde in die Bestellung übertragen wird, die zweite gibt die aktuelle Zeit zurück.

```
package de.webapp.OnlineShop;

import java.io.*;
import java.util.*;
import java.text.SimpleDateFormat ;
import de.webapp.Framework.SMI.*;
import de.webapp.Framework.Persistence.*;
import de.webapp.Framework.Utilities.*;

/** Oberklasse der OnlineShopXXXBeans */
public class OnlineShopBean extends SMICommandListener
   implements C_OnlineShop, Serializable {

/** Bestimme Kontext. */
public I_SMIContext getSMIContext() {
   return getSMIEventSwitch().getSMIContext();
}

/** Setze Applikation und Nachfolgekommando. */
```

```
public void executeContextFollowUp(SMICommand aSMICommand)
      throws OnlineShopException, PersistenceException {
   I_SMIContext theApp = getSMIContext() ;
   aSMICommand.putValue(C_Context, theApp ) ;
   executeFollowUpEvent(aSMICommand) ;
}

/** Setze den Wert, der sich hinter _FollowUpCommand verbirgt, im Kontext als
Kommando. */
public void executeFollowUpEvent(SMICommand aSMICommand)
      throws OnlineShopException {
   String followUpCommand =
      (String)aSMICommand.getValue(C_FollowUpCommand);
   if (followUpCommand != null) {
      try {
         // Setze Command.
         aSMICommand.setCommand(followUpCommand);
         // Ausführen des Folgekommandos
         getSMIEventSwitch().executeSMIEvent(aSMICommand);
      } catch( Exception e) {
          throw new OnlineShopException("Aufruf in "
             + getSMIEventSwitch().getName() + " fehlerhaft ",e) ;
      }
   }
}

/** Hole alle Daten des Shops erneut! */
public void refreshShop(SMICommand aSMICommand)
      throws OnlineShopException, PersistenceException {
   I_SMIContext theApp = getSMIContext() ;
   I_Store theStore = (I_Store)theApp.getValue(C_Store) ;
   PersistenceGeneric theShop =
      (PersistenceGeneric)theApp.getValue(C_Shop) ;
   // Hole Produktkatalog.
   I_PersistenceReference aRef =
      theShop.getPersistenceReference("Produkt") ;
   // Lösche Produktkatalog im Speicher.
   aRef.clear() ;
   // Besorge Produkte des Shops erneut.
   ReferenceVector theProducts =
     (ReferenceVector)theShop.getReference("Produkt") ;
   theApp.putValue(C_Products,theProducts) ;
   // Lösche Kunden und Bestellung
   aSMICommand.removeSessionValue(C_Customer);
   aSMICommand.removeSessionValue(C_Order);
   executeFollowUpEvent(aSMICommand)
}

/** Abgleich zwischen Kunde und Bestellung */
protected void checkCustomer(PersistenceGeneric aCustomer, PersistenceGeneric aOrder)
      throws OnlineShopException,PersistenceException {
```

```
   if (aCustomer != null) {
      // Setze Parameter des Kunden für die Bestellung
      if(aOrder != null) {
         aOrder.setReference("Kunde",aCustomer) ;
         for (int i = 0; i < C_BestellParameter.length; i++)
            aOrder.setValue(C_BestellParameter[i],
               aCustomer.getValue(C_BestellParameter[i]));
      }
   }
}

/** Setze die Werte eines persistenten Objektes. */
protected  void setPersistenceValues(I_Persistence aPersistence,
   SMICommand aSMICommand)
      throws PersistenceException {
   Enumeration e = aSMICommand.getKeys();
   String aValue;
   String aKey;
   while (e.hasMoreElements()) {
      aKey = (String)e.nextElement();
      if (aKey.startsWith(C_PersistenceAttributePrefix)
            && aKey.length() > C_PersistenceAttributePrefix.length()) {
         aValue = (String)aSMICommand.getValue(aKey);
         aKey = aKey.substring(C_PersistenceAttributePrefix.length());
         aPersistence.getPeer().getType().getAttributeWithName(
            aKey).setStringValue(aPersistence, aValue);
      }
   }
}

/** Teste, ob die Pflichtfelder aus einer Form gefüllt sind.
Feldnamen werden um C_PersistenceAttributePrefix ergänzt */
protected boolean checkMandadoryField(String aMandadoryFields[],
   SMICommand aSMICommand) {
   String theValue;
   boolean allFilled = true;
   for (int i =0; i < aMandadoryFields.length ;i++) {
      theValue = (String)aSMICommand.getValue(
         C_PersistenceAttributePrefix + aMandadoryFields[i]) ;
      if (theValue == null) {
         allFilled = false ;
         break ;
      } else
          if (theValue.length() == 0) {
             allFilled = false ;
             break;
          }
   }
   return allFilled ;
}
```

```
/** Ermittle aktuelle Zeit in der Zeitzone ECT. */
protected String getCurrentTime() {
   TimeZone tz = TimeZone.getTimeZone("ECT");
   SimpleDateFormat dateformat =
      new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
   dateformat.setTimeZone(tz);
   return dateformat.format(new Date());
}

/** Fehlermeldung mit Nachricht*/
protected void sendError(SMICommand aSMICommand, String aMessage)
      throws OnlineShopException {
   sendError(aSMICommand,aMessage,null) ;
}

/** Fehlermeldung mit Nachricht und Parametern*/
protected void sendError(SMICommand aSMICommand,
   String aMessage, Object[] aArguments)
      throws OnlineShopException {
   I_SMIContext theApp = getSMIContext() ;
   aSMICommand.putValue(C_Context, theApp ) ;
   aSMICommand.putValue(C_Message, aMessage ) ;
   if(aArguments != null)
      aSMICommand.putValue(C_Arguments, aArguments ) ;
   aSMICommand.putValue(C_Template, C_ErrorPage ) ;
   aSMICommand.putValue(C_FollowUpCommand,C_ErrorDisplay);
   executeFollowUpEvent(aSMICommand) ;
}

} // Ende der Klasse
```

*Listing 12.4: Klasse `OnlineShopBean`*

### 12.2.7 OnlineShopProductBean

Die Klasse `OnlineShopProductBean` hat die Aufgabe, den Produktkatalog anzuzeigen (Listing 12.5). Dies geschieht in der einzigen Methode, nämlich `showProducts()`. Bei der ersten Anzeige wird dabei der Katalog über die Assoziation `Produkt` des Shop-Objekts geladen (s. Kapitel 9). Für alle weiteren Anfragen steht dieser dann bereit. Um die Produkte auch anzeigen zu können, werden sie als Wert des Kommandos gesetzt (s. `putValue()`). Des weiteren wird der Parameter `Offset` aus der Anfrage dem Kommando hinzugefügt. Die Anzeige der Seite des Katalogs, die mit der `Offset`-Position beginnt, erfolgt mit Hilfe der Methode `OnlineShopBean.exceuteContextFollowUp()` durch einen Aufruf eines Folgekommandos.

```
package de.webapp.OnlineShop ;

import javax.servlet.http.*;
import de.webapp.Framework.SMI.*;
import de.webapp.Framework.Persistence.*;
```

```
import de.webapp.Framework.Utilities.*;

/** Anzeige der Produkte */
public class OnlineShopProductBean extends OnlineShopBean {

/** Hinterlege die Produkte des Shops. */
public void showProducts(SMICommand aSMICommand)
      throws OnlineShopException, PersistenceException {
   HttpServletRequest request = aSMICommand.getRequest() ;
   I_SMIContext theApp = getSMIContext() ;
   I_Store theStore = (I_Store)theApp.getValue(C_Store) ;
   PersistenceGeneric theShop =
     (PersistenceGeneric) theApp.getValue(C_Shop) ;
   ReferenceVector theProducts = null ;
   // Besorge Produkte des Shops.
   theProducts = (ReferenceVector)theApp.getValue(C_Products) ;
   if(theProducts == null) {
      theProducts = (ReferenceVector)theShop.getReference("Produkt") ;
      theApp.putValue(C_Products,theProducts) ;
   }
   // Vorbereitung zur Ausgabe der Produkte
   if(theProducts != null && theProducts.size() > 0 ) {
      aSMICommand.putValue(C_Context, theApp ) ;
      String theOffset = request.getParameter(C_Offset) ;
      if(theOffset != null)
      aSMICommand.putValue(C_Offset, theOffset ) ;
      executeFollowUpEvent(aSMICommand) ;
   } else
      sendError(aSMICommand,"Der Shop " + theShop.getValue("Name")
         + " hat keine Produkte! "
         + "<br>Bitte wenden Sie sich an die Adminstration "
         + "des Geschäfts !" ) ;
}
} // Ende der Klasse
```

*Listing 12.5: Klasse `OnlineShopProductBean`*

### 12.2.8 OnlineShopOrderBean

In der Klasse `OnlineShopOrderBean` sind alle Änderungskommandos einer Bestellung zusammengefaßt ( Listing 12.6). Die Anzeige erledigt wiederum eine JavaServer-Page (Listing 12.14).

Die Methode `newOrder(SMICommand)` erzeugt eine neue Bestellung, falls noch keine im Warenkorb (also in der `HttpSession`) vorliegt – sonst wird die vorhandene genutzt. Der Bestellung wird eine neue Bestellposition anhand der Anfrageparameter hinzugefügt. Der Wert der Ware wird dabei aus der Menge und dem Preis des Produkts berechnet und im Attribut `Betrag` der Bestellposition vermerkt.

Mit der Methode `changeOrder(SMICommand)` ist es möglich, eine Bestellposition zu ändern. Die Menge der Ware kann verringert oder erhöht werden. Nach der Änderung wird der Wert der Ware neu berechnet. Wenn die Warenmenge Null ist, wird die Position gelöscht.

Durch die Methode `sendOrder(SMICommand)` kann die Bestellung aus der `HttpSession` in die Datenbank übertragen werden. Zuvor wird überprüft, ob der Kunde, auf den die Bestellung gebucht werden soll, angemeldet ist. Ist dies nicht der Fall, wird die Anmeldung erzwungen. Ist er jedoch angemeldet, wird der Bestellung der Shop und der Kunde zugeordnet und das aktuelle Datum als Bestelldatum gesetzt. Nach einer fehlerfreien Speicherung in der Datenbank, folgt die Bestätigung der Bestellung durch die Ausführung eines Nachfolgekommandos.

```
package de.webapp.OnlineShop;

import javax.servlet.http.*;
import java.math.BigDecimal;
import java.sql.Timestamp ;
import de.webapp.Framework.SMI.*;
import de.webapp.Framework.Persistence.*;
import de.webapp.Framework.Utilities.*;

/** Behandlung von Bestellungen */
public class OnlineShopOrderBean extends OnlineShopBean {

/** Lege Bestellung in der Session des Kunden an. */
public void newOrder(SMICommand aSMICommand)
      throws OnlineShopException, PersistenceException {
   // Setup
   I_SMIContext theApp = getSMIContext() ;
   I_Store theStore = (I_Store)theApp.getValue(C_Store) ;
   // Exisitiert Bestellung in der Session?
   PersistenceGeneric theOrder =
      (PersistenceGeneric)aSMICommand.getSessionValue(C_Order);
   // Existiert angemeldeter Kunde?
   PersistenceGeneric theCustomer =
      (PersistenceGeneric) aSMICommand.getSessionValue(C_Customer);
   // Hole Bestellung
   if (theOrder == null) {
      // Neue Bestellung anlegen
      theOrder =
         (PersistenceGeneric)theStore.newPersistence("OnlineBestellung") ;
      // Setze Kundenparameter falls vorhanden.
      if (theCustomer != null)
         checkCustomer(theCustomer,theOrder) ;
      // Nun ist die Bestellung sichtbar
      aSMICommand.putSessionValue(C_Order,theOrder) ;
   }
```

```
   // Erzeuge Bestellposition für ausgewähltes Produkt
   String theProductID = (String)aSMICommand.getValue(C_ProductID) ;
   // Ermittle Bestellmenge.
   int theMenge =
      new Integer((String)aSMICommand.getValue(C_Menge)).intValue() ;
   if (theMenge < 1) {
      sendError(aSMICommand,C_Vielfaches) ;
      return ;
   }
   // Erzeugen neue Bestellposition
   PersistenceGeneric thePosition =
     (PersistenceGeneric)theStore.newPersistence("OnlineBestellposition") ;
   thePosition.setValue("ProduktID",theProductID) ;
   thePosition.setValue("Menge",new Integer(theMenge)) ;
   // Ermittle Betrag, falls Produkt existiert.
   PersistenceGeneric theProdukt =
     (PersistenceGeneric) thePosition.getReference("Produkt") ;
   if (theProdukt == null) {
      sendError(aSMICommand,C_ProductFailed,
         new Object[]{theOrder,thePosition}) ;
      return ;
   }
   BigDecimal theVKBetrag = (BigDecimal)theProdukt.getValue("VKPreis") ;
   Integer theVKMenge = (Integer)theProdukt.getValue("VKMenge") ;
   if(theMenge % theVKMenge.intValue() != 0) {
      sendError(aSMICommand,"Eingebene Menge "
         + theMenge
         + " ist kein Vielfaches der Produktmenge " + theVKMenge ) ;
      return ;
   }
   // OK, Eintragen des Betrags.
   thePosition.setValue("Betrag",
   theVKBetrag.multiply(new BigDecimal(theMenge / theVKMenge.intValue()))) ;
   // Bestellungsposition zur Bestellung hinzufuegen
   theOrder.addToReference("Bestellposition",thePosition) ;
   executeContextFollowUp(aSMICommand) ;
}
/** Änderung der Menge */
// Löschen einer Bestellposition, wenn Menge 0.
public void changeOrder(SMICommand aSMICommand)
      throws OnlineShopException, PersistenceException {
   HttpServletRequest request = aSMICommand.getRequest() ;
   String thePosition =
      (String)request.getParameter(C_DeleteBestellPosition) ;
   if (thePosition == null) {
      sendError(aSMICommand,
         "Keine Bestellposition zu Löschung an geben") ;
      return ;
   }
   int position = new Integer(thePosition).intValue() ;
   // Manipuliere Bestellung.
```

```
   PersistenceGeneric theOrder=
      (PersistenceGeneric)aSMICommand.getSessionValue(C_Order);
   synchronized(theOrder) {
      if(theOrder != null
         && theOrder.getObjectState() == C_Persistence.STATE_UNDEFINED) {
         String theMenge = (String)request.getParameter(C_Menge) ;
         if(theMenge == null) {
            sendError(aSMICommand,"Keine Menge angeben") ;
            return ;
         }
         int menge = new Integer(theMenge).intValue() ;
         ReferenceVector theOrders =
            (ReferenceVector)theOrder.getReference("Bestellposition") ;
         if(menge == 0) {
            theOrders.removeElementAt(position) ;
            if(theOrders.size() == 0)
               aSMICommand.removeSessionValue(C_Order) ;
         } else {
            // Hole Bestellposition und korrigiere Menge und Betrag.
            PersistenceGeneric theOrderPos = (PersistenceGeneric)
               theOrders.elementAt(position) ;
            theOrderPos.setValue("Menge",new Integer(menge));
            PersistenceGeneric theProdukt = (PersistenceGeneric)
            theOrderPos.getReference("Produkt") ;
            // Korrigiere Betrag.
            BigDecimal theVKBetrag =
               (BigDecimal)theProdukt.getValue("VKPreis") ;
            Integer theVKMenge = (Integer)theProdukt.getValue("VKMenge") ;
            theOrderPos.setValue("Betrag",
            theVKBetrag.multiply(
               new BigDecimal(menge / theVKMenge.intValue()))) ;
         }
      }
   }
   // Führe Nachfolgekommando aus.
   executeContextFollowUp(aSMICommand) ;
}

/** Hole alle Daten des Shops neu! */
public void sendOrder(SMICommand aSMICommand)
      throws OnlineShopException, PersistenceException {
   I_SMIContext theApp = getSMIContext() ;
   I_Store theStore = (I_Store)theApp.getValue(C_Store) ;
   PersistenceGeneric theCustomer =
     (PersistenceGeneric)aSMICommand.getSessionValue(C_Customer);
   if(theCustomer == null) {
      // Kein Kunde angemeldet => erzwinge Anmeldung.
      aSMICommand.putValue(C_FollowUpCommand,C_DisplayCommand) ;
      aSMICommand.putValue(C_Template,C_LoginCustomerRestartOrder) ;
      executeContextFollowUp(aSMICommand) ;
      return ;
```

```
    }
    // Kunde angemeldet => besorge aktive Bestellung von Session.
    PersistenceGeneric theOrder =
       (PersistenceGeneric)aSMICommand.getSessionValue(C_Order);
    PersistenceGeneric theShop = (PersistenceGeneric)
       theApp.getValue(C_Shop) ;
    // Sende Bestellung.
    if (theOrder != null
       && theOrder.getObjectState() == C_Persistence.STATE_UNDEFINED) {
       if (theOrder.getValue("ShopID") == null)
          theOrder.setReference("Shop",theShop) ;
       // Setze akuelle Zeit
       theOrder.setValue("Bestelldatum",
          Timestamp.valueOf(getCurrentTime()+".000000000")) ;
       // Setze Werte im Persistence.
       setPersistenceValues(theOrder,aSMICommand) ;
       // Check
       ReferenceVector thePositions =
          (ReferenceVector)theOrder.getReference("Bestellposition") ;
       if (thePositions.size() > 0) {
          // Die Bestellung wird dem aktuellen Kunden zugeordnet
          // und gespeichert.
          theCustomer.addToReference("Bestellung",theOrder) ;
          theCustomer.update() ;
          // Lösche die aktuelle Bestellung aus der Session.
          // Bestellung nun erfolgt.
          aSMICommand.removeSessionValue(C_Order);
       } else {
          sendError(aSMICommand,C_OrderNotSet) ;
          return ;
       }
       executeContextFollowUp(aSMICommand) ;
    } else
       sendError(aSMICommand,C_OrderNotSet) ;
}
} // Ende der Klasse
```

*Listing 12.6: Klasse `OnlineShopOrderBean`*

### 12.2.9 OnlineShopCustomerBean

Die Klasse `OnlineShopCustomerBean` realisiert die Kundenverwaltungskommandos. Um die Daten eines Kunden zu ändern, muß die Methode `changeCustomer()` aufgerufen werden. Sie überprüft zunächst, ob alle Pflichtfelder gesetzt wurden. Ist dies der Fall wird nachgesehen, ob der Kunde schon registriert ist. Als Kriterium dient dabei die E-Mail-Adresse der Eingabe. Danach wird ob das Geheimwort überprüft. Falls die Angaben korrekt sind, werden die Werte aus der Eingabe übernommen und in der Datenbank abgespeichert. Ansonsten wird eine Fehlermeldung angezeigt.

Die Methode `setCustomer()` ist der gerade beschriebenen Methode sehr ähnlich – nur daß ausschließlich eine Anmeldung erfolgt, ohne daß die Daten eines Kunden geändert werden. Nach erfolgreicher Anmeldung werden die Kundendaten in die aktive Bestellung übernommen und ein eventuell vorhandenes Folgekommando, das im Kommando steckt, ausgeführt.

Die Hilfsmethode `getCustomer()` sucht den Kunden aus der Datenbank. Dies geschieht auf Basis des aktuellen Shops und der E-Mail-Adresse.

```
package de.webapp.OnlineShop;

import javax.servlet.http.*;

import de.webapp.Framework.SMI.*;
import de.webapp.Framework.Log.Log;
import de.webapp.Framework.Persistence.*;
import de.webapp.Framework.Utilities.*;

/** Realisiert Kundenkommandos. */
public class OnlineShopCustomerBean extends OnlineShopBean {

/** Änderung eines Kunden */
public void changeCustomer(SMICommand aSMICommand)
      throws OnlineShopException, PersistenceException {
   I_SMIContext theApp = getSMIContext() ;
   I_Store theStore = (I_Store)theApp.getValue(C_Store) ;
   PersistenceGeneric theShop =
      (PersistenceGeneric)theApp.getValue(C_Shop) ;
   PersistenceGeneric theCustomer = null ;
   // Test, ob alle Felder im Kommando enthalten sind.
   if (!checkMandadoryField(C_CustomerFields,aSMICommand)) {
      sendError(aSMICommand,
         "Nicht alle Pflichtfelder gesetzt ", C_CustomerFields ) ;
      return ;
   }
   String theMail = (String)aSMICommand.getValue(C_CustomerMail) ;
   // bestimme Kunde
   theCustomer = (PersistenceGeneric)
      getCustomer(theStore,theMail,theShop.getObjectIdentifier()) ;
   if(theCustomer == null) {
      // Neuer Kunde
      theCustomer =
         (PersistenceGeneric)theStore.newPersistence("OnlineKunde") ;
      theCustomer.setReference("Shop",theShop) ;
      setPersistenceValues(theCustomer,aSMICommand) ;
      theCustomer.create() ;
   } else {
      // Bestehender Kunde
      if(!checkPassword(theCustomer,
            (String)aSMICommand.getValue(C_CustomerPassword) )) {
```

```
            sendError(aSMICommand,
                "Geheimwort des Kunden "
               + theCustomer.getValue("EMail") + " ist falsch" ) ;
            return ;
      }
      setPersistenceValues(theCustomer,aSMICommand) ;
      theCustomer.update() ;
   }
   // Kunden in die Session einfügen.
   aSMICommand.putSessionValue(C_Customer,theCustomer);
   executeContextFollowUp(aSMICommand) ;
}

/** Setzt Kunden. */
public void setCustomer(SMICommand aSMICommand)
      throws OnlineShopException, PersistenceException {
    PersistenceGeneric theCustomer =
       (PersistenceGeneric) aSMICommand.getSessionValue(C_Customer);
    PersistenceGeneric theOrder =
       (PersistenceGeneric) aSMICommand.getSessionValue(C_Order);
    if (!checkMandadoryField(C_CustomerLoginFields, aSMICommand)) {
       sendError(aSMICommand,
          "Nicht alle Pflichtfelder gesetzt ", C_CustomerLoginFields ) ;
       return ;
   }

   String theMail = (String)aSMICommand.getValue(C_CustomerMail) ;
   // Bestehende Kunden ersetzen.
   if(theCustomer != null) {
      if(theMail.equals(theCustomer.getValue("EMail"))) {
         if (!checkPassword(theCustomer,
                  (String)aSMICommand.getValue(C_CustomerPassword) )) {
            sendError(aSMICommand,
                "Geheimwort des Kunden "
               + theCustomer.getValue("EMail") + " ist falsch" ) ;
            return ;
         }
         // Abgleich zwischen Kunden und Bestellung.
         checkCustomer(theCustomer,theOrder) ;
      } else {
         // Kunde hat sich geändert.
         aSMICommand.removeSessionValue(C_Customer);
         theCustomer = null ;
      }
   }
   // Kunde der Session setzen.
   if (theCustomer == null) {
      I_SMIContext theApp = getSMIContext() ;
      I_Store theStore = (I_Store)theApp.getValue(C_Store) ;
      PersistenceGeneric theShop =
      (PersistenceGeneric)theApp.getValue(C_Shop) ;
```

```
      // Bestimme Kunden.
      theCustomer = (PersistenceGeneric)
         getCustomer(theStore,theMail, theShop.getObjectIdentifier()) ;
      if (theCustomer != null) {
         if (!checkPassword(theCustomer,
         (String)aSMICommand.getValue(C_CustomerPassword) )) {
            sendError(aSMICommand,
            "Geheimwort des Kunden "
            + theCustomer.getValue("EMail") + " ist falsch" ) ;
            return ;
         }
       } else {
          sendError(aSMICommand,
              "Sie sind noch nicht Kunde.<BR>"
             + " Bitte tragen Sie Ihre Daten im Kundendialog ein!" ) ;
          return ;
        }
        // Abgleich zwischen Kunden und Order
        checkCustomer(theCustomer,theOrder) ;
        // Merke den Kunden in der Session.
        aSMICommand.putSessionValue(C_Customer,theCustomer);
   }
   HttpServletRequest request = aSMICommand.getRequest() ;
   String theFollowUp = (String)request.getParameter(C_FollowUpCommand) ;
   if(theFollowUp != null) {
      String theTemplate = (String)request.getParameter(C_Template) ;
      aSMICommand.putValue(C_FollowUpCommand,theFollowUp) ;
      aSMICommand.putValue(C_Template,theTemplate) ;
   }
   executeContextFollowUp(aSMICommand) ;
}

/** Test eines Paßworts */
protected boolean checkPassword(PersistenceGeneric aCustomer,
   String aPassword)
      throws PersistenceException {
   String thePassword = (String) aCustomer.getValue("Password");
   return thePassword != null && thePassword.equals(aPassword) ;
}

/** Hole Kunde des Shop über seine Mailadresse  */
// Die Funktion muß in einem synchronized- und
// Transaktionsrahmen genutzt werden.
protected I_Persistence getCustomer(I_Store aStore,
   String aMail, String aShopID)
      throws PersistenceException {
   if(aStore == null || aMail == null || aShopID == null)
      return null ;
   // Hole Kunde des Shops.
   ReferenceVector theCustomers =
      aStore.getRetriever("OnlineKunde").retrieve(
```

```
      " WHERE tO.ShopID ='" + aShopID+ "'" +
      " AND tO.EMail = '" + aMail + "'" ) ;
   if(theCustomers.size() != 1 )
      return null ;
   return (I_Persistence)theCustomers.elementAt(0) ;
}

} // Ende der Klasse
```

*Listing 12.7: Klasse* `OnlineShopCustomerBean`

### 12.2.10 Ausnahmedefinition OnlineShopException

Für den OnlineShop benötigen wir eine Ausnahme-Klasse: die `OnlineShopException`. Wenn eine Ausnahmesituation eintritt, wird eine neue Instanz der `OnlineShopException` erzeugt und die Framework- oder JDK-Ausnahme darin gekappselt ( Listing 12.8).

```
package de.webapp.OnlineShop;

import de.webapp.Framework.Utilities.WebAppException ;

/** Ausnahme für den OnlineShop */
public class OnlineShopException extends WebAppException {

/** Anlage der Ausnahme */
public OnlineShopException() {}
/** Beschreibung des Fehlers ist aus Throwable zu bekommen. *//
public OnlineShopException(Throwable aThrowable) {
   super(aThrowable) ;
}
/** Setzt Nachricht, die die Ausnahme beschreibt. */
public OnlineShopException(String aMessage) {
   super(aMessage) ;
}
/** Setzt Nachricht, die die Exception beschreibt. */
public OnlineShopException(String aMessage,Throwable aThrowable) {
   super(aMessage,aThrowable) ;
}
} // Ende der Klasse
```

*Listing 12.8: Klasse* `OnlineShopException`

## 12.3 Anzeige

Nachdem wir intensiv die Komponenten der Anwendung besprochen haben, definieren wir nun die Dialoge.

### 12.3.1 Code-Fragemente importieren

Schon im StoreBrowser (Kapitel 11) wurden die Server-Side-Includes für `header.html` und `footer.html` vorgestellt. Diese finden auch im OnlineShop ihre Anwendung.

Für die notwendigen Java-`Imports`, sorgt wiederum das Code-Fragment `import.import` (Listing 12.9).

```
<%@ page import = "java.util.Enumeration" %>
<%@ page import = "javax.servlet.ServletException" %>
<%@ page import = "de.webapp.Framework.Utilities.ReferenceVector" %>
<%@ page import = "de.webapp.Framework.Persistence.*" %>
<%@ page import = "de.webapp.Framework.SMI.*" %>
<%@ page import = "de.webapp.OnlineShop.C_OnlineShop" %>
```

*Listing 12.9: Code-Fragment `import.import`*

### 12.3.2 Gesamtbild

Zur Anzeige des OnlineShops haben wir uns für die Aufspaltung der Oberfläche in zwei HTML-Frames entschieden. Im linken Frame befindet sich ein Funktionsmenü, durch das alle Dialoge des OnlineShops erreichbar sind (Listing 12.10 und Abbildung 12.6). Wir nennen diesen Frame `uebersicht`. Im rechten Frame – dem `info`-Frame – werden die Dialoge des OnlineShops dargestellt.

```
<html>
<head>
<title>OnlineShop</title>
</head>
<frameset border=0 cols="210, *">
   <frame name=uebersicht src="OnlineShop?Command=showMenu" scrolling="no"
frameborder="0">
   <frame name=info src="OnlineShop?Command=showShop" scrolling="auto"
frameborder="0">
</frameset>
</html>
```

*Listing 12.10: Definition der Frames des Shops*

Die Anzeige der Startseite erfolgt durch die Kommandos `showMenu` und `showShop` des OnlineShops.

Im Menü sind die Verweise auf den Produktkatalog (`showProducts`), Anzeige des Warenkorbs (`showOrder`), Kundeninformationen (`showCustomer`) und das Anmelden (`loginCustomer`) und Abmelden (`logoutCustomer`) (Listing 12.11) untergebracht. Der Verweis Start lädt den gesamten Shop neu.

```
<html>
<head>
<title>OnlineShop</title>
</head>
<body background="Menue/hintergrund.gif">
   <img src="Menue/shop.gif"><br><br><br>
   <a target=info href="/OnlineShop/OnlineShop?Command=showProducts"><img border=0
src="Menue/katalog.gif" alt="Katalog"></a><br>
   <a target=info href="/OnlineShop/OnlineShop?Command=showOrder"><img border=0
```

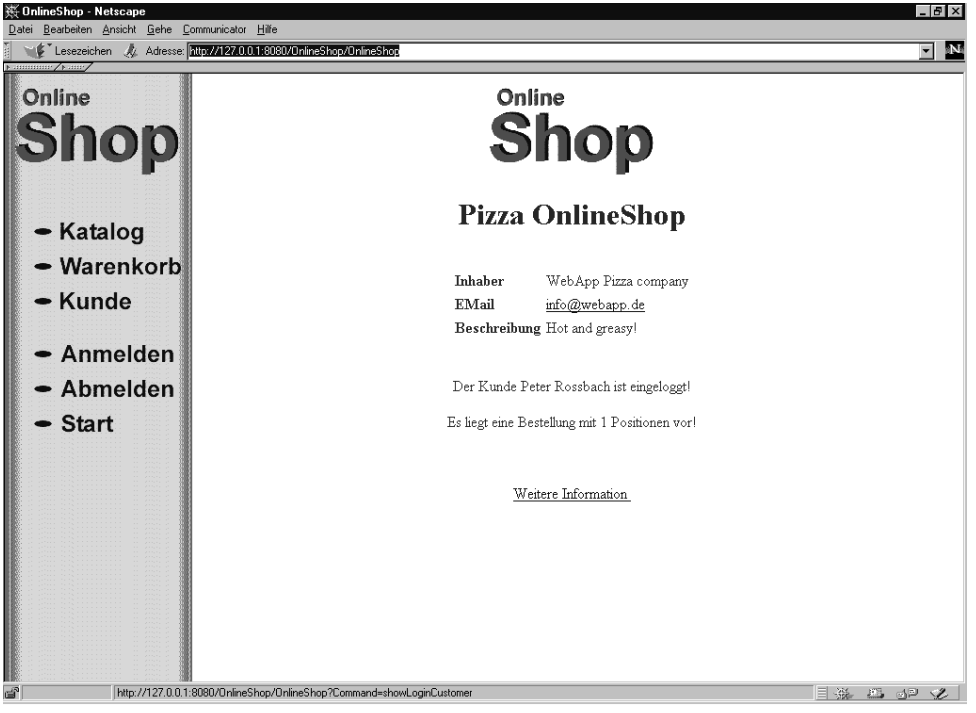


*Abbildung 12.6: Startseite des OnlineShops*

```
src="Menue/warenkorb.gif" alt="Warenkorb"></a><br>
   <a target=info href="/OnlineShop/OnlineShop?Command=showCustomer"><img border=0
src="Menue/kunde.gif" alt="Kunde"></a><br><br>
   <a target=info href="/OnlineShop/OnlineShop?Command=showLoginCustomer"><img
border=0 src="Menue/anmelden.gif" alt="Anmelden"></a><br>
   <a target=_top href="/OnlineShop/OnlineShop?Command=logoutCustomer"><img  border=0
src="Menue/abmelden.gif" alt="Abmelden"></a><br>
   <a target=_top href="/OnlineShop/OnlineShop?Command=startShop"><img border=0
src="Menue/start.gif" alt="Start"></a><br>
</body>
</html>
```

*Listing 12.11: Funktions-Menü des OnlineShops*

Die Aufbereitung der Informationen über den Shop (Inhaber etc.) wird durch eine einfache JSP realisiert (Listing 12.12). Aus der Anfrage wird das `SMICommand` entnommen. Das Kommando liefert uns den `OnlineShopContext` und darüber Zugang zum Shop. Jetzt müssen nur noch die Attribute des Shop-Objekts ausgegeben werden. Falls ein Kunde angemeldet ist, wird dessen Name ebenfalls angezeigt.

Zudem wird sein Warenkorb mit der Anzahl der Bestellpositionen ausgegeben, sofern eine Bestellung vorliegt.

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>

<center>
<%
// Kommando besorgen.
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent") ;
I_SMIContext theSMIContext =
   (I_SMIContext)theSMICommand.getValue(C_OnlineShop.C_Context) ;
PersistenceGeneric theShop =
   (PersistenceGeneric)theSMIContext.getValue(C_OnlineShop.C_Shop) ;
%>
<img border=0 src="Menue/shop.gif"><br><br>
<h1><%=theShop.getValue("Name")%></h1>
<br>
<table>
<tr><td><b>Inhaber</b></td><td><%=theShop.getValue("Inhaber")%></td></tr>
<tr><td><b>EMail</b></td><td>
<ahref='mailto:<%=theShop.getValue("InhaberEMail")%>'><%=theShop.getValue("InhaberEMa
il")%></a></td></tr>
<tr><td><b>Beschreibung</b></td><td><%=theShop.getValue("Beschreibung")%></td></tr>
</table>
<br>
<%
PersistenceGeneric theCustomer=
   (PersistenceGeneric)theSMICommand.getSessionValue(
      C_OnlineShop.C_Customer) ;
if(theCustomer != null) {
%>
<br>Der Kunde <%=theCustomer.getValue("Name")%> ist eingeloggt!<br>
<%}
PersistenceGeneric theOrder =
   (PersistenceGeneric) theSMICommand.getSessionValue(
      C_OnlineShop.C_Order) ;
if(theOrder != null) {
   ReferenceVector theOrders = null ;
   try {
      theOrders =
         (ReferenceVector)theOrder.getReference("Bestellposition") ;
   } catch ( PersistenceException pe ) {
      throw new ServletException( "Modellfehler", pe) ;
   }
%>
<br>Es liegt eine Bestellung mit <%=theOrders.size()%> Positionen vor!<br>
<br>
<%}
String theLink = (String)theShop.getValue("InfoURL") ;
if(theLink!=null) {
%>
<br>
<br>
```

```
<a href="<%=theLink%>">Weitere Informationen<a>
<%}%>
</center>
<%@ include file="footer.html" %>
```

*Listing 12.12: JavaServer-Page* `ShowShop.jsp`

### 12.3.3 Der Produktkatalog

Der Produktkatalog wird seitenweise angezeigt. Links erscheint das Bild des Produkts, rechts die dazugehörigen Informationen. Unterhalb wird eine Beschreibung ausgegeben (Abbildung 12.7). Je nach Seite können wir im Katalog vor- und zurückblättern.

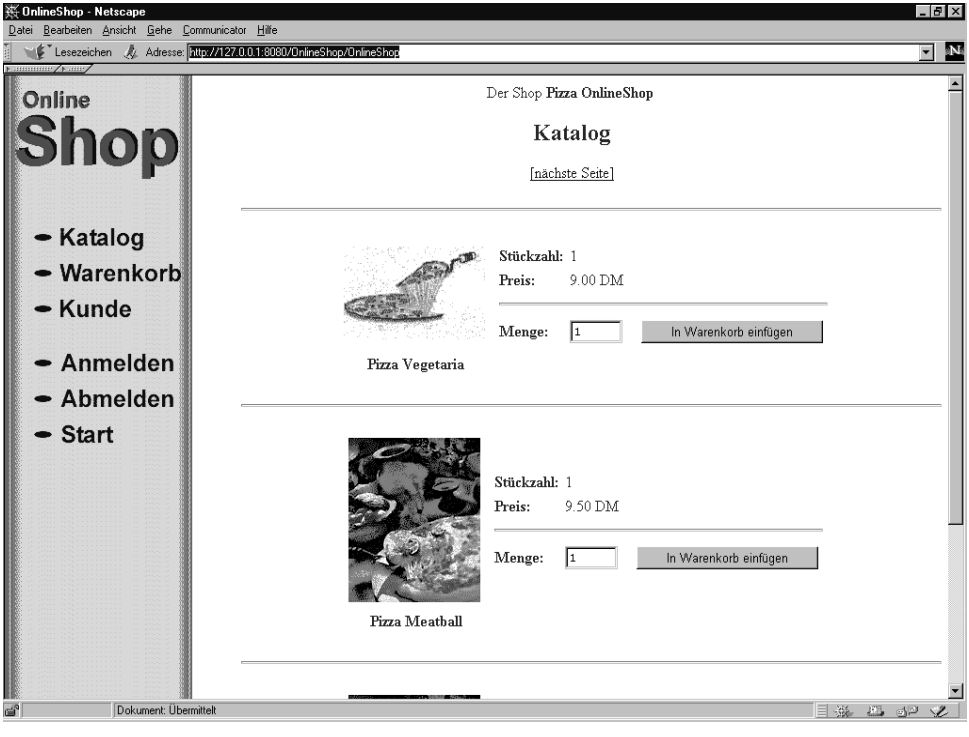


*Abbildung 12.7: Dialog Produktkatalog*

Nach der obligatorischen Bestimmung des Shops aus dem aktuellen Kommando werden die Produkte aus dem Shop angefordert (Listing 12.13), Seitengröße und Anzahl der Produkte bestimmt und entschieden, ob es Verweise für das Blättern geben muß. Anschließend wird der anzuzeigende Katalog-Ausschnitt ausgegeben.

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>

<%
// Kommando besorgen.
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent") ;
I_SMIContext theSMIContext =
   (I_SMIContext)theSMICommand.getValue(C_OnlineShop.C_Context) ;
PersistenceGeneric theShop =
   (PersistenceGeneric)theSMIContext.getValue(C_OnlineShop.C_Shop) ;
ReferenceVector theProducts =
   (ReferenceVector)theSMIContext.getValue(C_OnlineShop.C_Products) ;
%>
<center>
Der Shop <b><%=theShop.getValue("Name")%></b>
<h2>Katalog</h2>
<%
int thePageSize  = 10 ;
int theOffset = 0 ;
try {
   thePageSize =
      ((Integer)theSMIContext.getValue(
         C_OnlineShop.C_PageSize)).intValue() ;
   theOffset =
   Integer.parseInt((String)theSMICommand.getValue(
      C_OnlineShop.C_Offset)) ;
} catch (NumberFormatException nfe) {}
if (theProducts != null && theProducts.size() > thePageSize) {
   if (theOffset > 0) {
      int previous = theOffset - thePageSize;
      out.println("<a href='OnlineShop?Command=showProducts&"
          + C_OnlineShop.C_Offset + "="
          +  previous + "'>[vorherige Seite]</a><br>");
  }
  if (theOffset + thePageSize < theProducts.size()) {
     int next = theOffset + thePageSize;
     out.println("<a href='OnlineShop?Command=showProducts&"
        + C_OnlineShop.C_Offset + "="
        +  next + "'>[nächste Seite]</a><br>");
   }
}
%>
<ul>
<%
   if(theProducts != null) {
      for (int i=theOffset; i<theProducts.size()
         && i < theOffset+thePageSize; i++) {
         PersistenceGeneric theProduct =
            (PersistenceGeneric)theProducts.elementAt(i) ;
         int number = theProducts.size()-i;
         String theImage = (String)theProduct.getValue("BildURL") ;
```

```
         if(theImage == null)
            theImage = "Bild/KeinBild.gif" ;
%>
         <hr>
         <form action="OnlineShop" method="POST">
             <input type="Hidden" name="Command" value="newOrder">
             <input type="Hidden" name="_ProductID"
value="<%=theProduct.getObjectIdentifier()%>">
             <table>
                <tr>
                <td align=center>
                <img src="<%=theImage%>">
                </td>
                <td rowsspan=2 valign=center>
                <table>
                   <tr>
                   <td><b>Stückzahl:</b></td>
                     <td><%=theProduct.getValue("VKMenge")%></td>
                   </tr>
                   <tr>
                   <td><b>Preis:</b></td>
                     <td><%=theProduct.getValue("VKPreis")%> DM </td>
                   </tr>
                   <tr><td colspan=2><hr></td></tr>
                   <tr>
                   <td><b>Menge:</b></td>
                     <td>
                     <input type="Text" name="_Menge" size="5" value="1">
                         
                     <input type="Submit" value="In Warenkorb einfügen">
                     </td>
                   </tr>
                </table>
                </td>
                </tr>
                <tr>
                <td align=center>
                   <b><%=theProduct.getValue("Beschreibung")%></b></td>
                </tr>
           </table>
        </form>
<%
   }
}
%>
</ul>
</center>
<%@ include file="footer.html" %>
```

*Listing 12.13: JavaServer-Page* `ShowProduct.jsp`

### 12.3.4 Anzeige des Warenkorbs

Wenn eine Bestellung im Warenkorb enthalten ist, wird diese angezeigt (Abbildung 12.8); ebenso die zugehörigen Kundendaten. Vor dem Abschicken einer Bestellung kann man noch eine Bemerkung erfassen. Außerdem lassen sich die Bestellpositionen leicht ändern, wobei die Beträge automatisch angepaßt werden (Listing 12.6). Ist alles wie gewünscht, können wir die Bestellung mit der Schaltfläche `Bestellung aufgeben` abschicken.

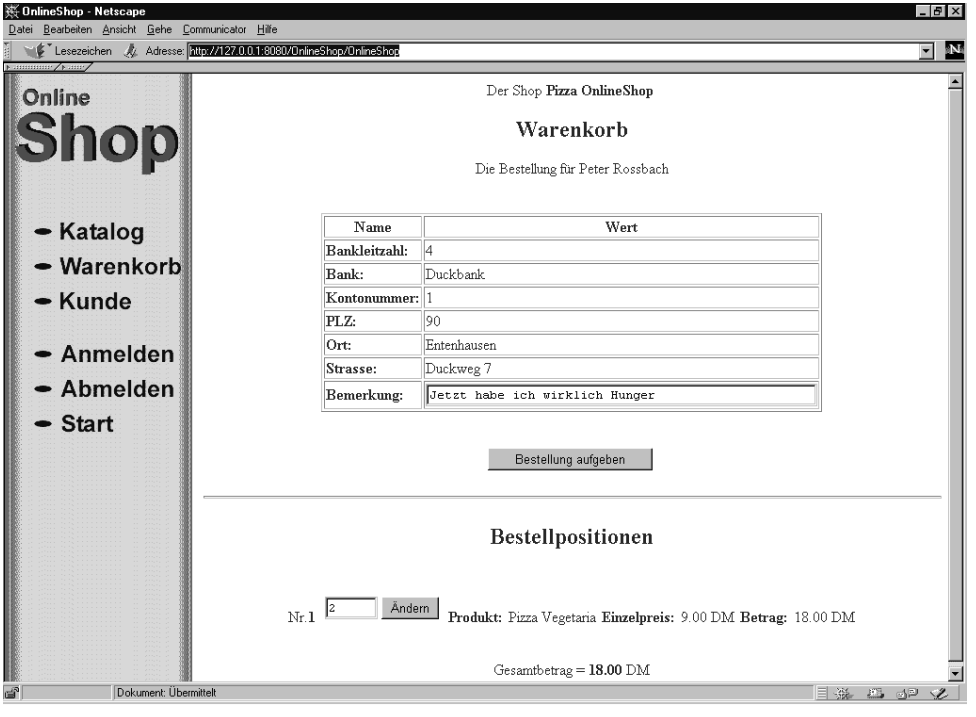


*Abbildung 12.8: Anzeige des Warenkorbs*

Um die geforderte Funktionalität umzusetzen, werden aus der `HttpSession` der Kunde und die Bestellung gelesen und entsprechend formatiert ausgegeben. Die Anzeigefelder werden durch Konstanten aus der Schnittstelle `C_OnlineShop` bestimmt.

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>
<%@ page import = "java.math.BigDecimal" %>

<%
// Kommando besorgen
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent") ;
```

```
I_SMIContext theSMIContext =
   (I_SMIContext)theSMICommand.getValue(C_OnlineShop.C_Context) ;
PersistenceGeneric theOrder =
   (PersistenceGeneric)request.getSession(
      false ).getValue(C_OnlineShop.C_Order) ;
PersistenceGeneric theShop =
   (PersistenceGeneric)theSMIContext.getValue(C_OnlineShop.C_Shop) ;
PersistenceGeneric theCustomer =
   (PersistenceGeneric)theSMICommand.getSessionValue(
      C_OnlineShop.C_Customer) ;
%>
<center>
Der Shop <b><%=theShop.getValue("Name")%></b>
<h2>Warenkorb</h2>
<%
   if(theOrder!= null) {
      ReferenceVector theOrderPosition = null ;
      try {
         theOrderPosition =
            (ReferenceVector)theOrder.getReference("Bestellposition") ;
      } catch ( PersistenceException pe ) {
         throw new ServletException( "Modellfehler", pe) ;
      }
      if(theCustomer != null) {
%>
         Die Bestellung für <%=theCustomer.getValue("Name")%></h2>
    <%} else { %>
        <br>Sie müssen sich noch anmelden,
        bevor Sie eine Bestellung aufgeben können!<br>
 <%}%>
<br><form action="OnlineShop" method="POST">
<input type="Hidden" name="Command" value="sendOrder">
<%
   for (int i = 0; i < C_OnlineShop.C_CustomerAddressFields.length; i++) {
%>
      <input type="Hidden" name="_PA<%=C_OnlineShop.C_CustomerAddressFields[i][0]%>"
      value="<%=theOrder.getValue(C_OnlineShop.C_CustomerAddressFields[i][1])%>">
 <%}%>
<table border=1>
<tr><th>Name</th><th>Wert</th></tr>
<%
   for (int i = 0; i < C_OnlineShop.C_CustomerAddressFields.length; i++) {
%>
<tr>
   <td><b><%=C_OnlineShop.C_CustomerAddressFields[i][0]%>:</b></td>
   <td><%=theOrder.getValue(C_OnlineShop.C_CustomerAddressFields[i][1])%></td>
</tr>
<%}%>
<tr>
<td><b>Bemerkung:</b></td>
<td><input type="Text" name="_PABemerkung" size="50"
```

```
value='<%=theOrder.getValue("Bemerkung")%>'></td>
</tr>
</table>
<br>
<br>
<input type="Submit" value="Bestellung aufgeben">
</form>
<hr>
<br><h2>Bestellpositionen</h2><br>
<%
   if(theOrderPosition != null) {
%>
      <table>
<%
      BigDecimal theGesamtBetrag = new BigDecimal(0.0) ;
      int i = 0 ;
      for (Enumeration e=theOrderPosition.elements();
         e.hasMoreElements();) {
         PersistenceGeneric thePosition =
            (PersistenceGeneric)e.nextElement() ;
         I_Persistence theProduct = null ;
         try {
            theProduct =
               (I_Persistence)thePosition.getReference("Produkt") ;
         } catch (PersistenceException pe) {
            throw new ServletException("Modellfehler", pe) ;
         }
         theGesamtBetrag =
            theGesamtBetrag.add(
               (BigDecimal)thePosition.getValue("Betrag"));
%>
         <tr>
         <td>Nr.<b><%=theProduct.getValue("Nummer")%></b> </td>
         <td>
         <form action="OnlineShop" method="GET">
            <input type="Hidden" name="Command" value="changeOrder">
            <input type="Hidden" name="<%=C_OnlineShop.C_DeleteBestellPosition%>"
value="<%=i%>">
            <table border=1>
            <input type="Text" name="_Menge" size="5"
value='<%=thePosition.getValue("Menge")%>'></td>
            </table>
            <input type="Submit" value="Ändern">
        </form>
        </td>
        <td><b>Produkt:</b></td>
        <td><%=theProduct.getValue("Beschreibung")%></td>
        <td><b>Einzelpreis:</b></td>
        <td><%=theProduct.getValue("VKPreis")%> DM</td>
        <td><b>Betrag:</b></td>
        <td><%=thePosition.getValue("Betrag")%> DM</td>
```

```
        </tr>
<%      i++ ;
      }%>
      </table>
      <br>Gesamtbetrag = <b><%=theGesamtBetrag%></b> DM<br>
  <%}
  } else {%>
Keine Bestellung aktiv !<br>
<br>Auswahl der Bestellpositionen durch Produktkatalog.
<%}%>
</center>
<%@ include file="footer.html" %>
```

*Listing 12.14: Die JavaServer-Page ShowOrder.jsp*

### 12.3.5 Anzeige der Kundeninformation

Die Anzeige der Kundeninformation ist ein Standard-HTML-Formular (Abbildung 12.9). Innerhalb einer Tabelle werden alle Eingabefelder aufgelistet. Die Eingabe des Geheimworts geschieht dabei wie üblich verdeckt.


*Abbildung 12.9: Anzeige der Kundendaten*

Das Ändern der eigenen Daten ist auf Knopfdruck möglich.

Um einen Überblick über die eigenen Bestellungen zu erhalten, werden alle Bestellungen absteigend sortiert ausgeben (Listing 12.15).

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>
<%@ page import = "java.sql.Timestamp" %>
<%@ page import = "java.text.SimpleDateFormat" %>
<%@ page import = "java.util.TimeZone" %>

<%
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent") ;
I_SMIContext theSMIContext =
   (I_SMIContext)theSMICommand.getValue(C_OnlineShop.C_Context) ;
I_Store theStore = (I_Store)theSMIContext.getValue(C_OnlineShop.C_Store) ;
PersistenceGeneric theCustomer = (PersistenceGeneric)
   theSMICommand.getSessionValue(C_OnlineShop.C_Customer) ;
PersistenceGeneric theShop = (PersistenceGeneric)
   theSMIContext.getValue(C_OnlineShop.C_Shop) ;
%>
<center>
Der Shop <b><%=theShop.getValue("Name")%></b>
<br>
<h2>Kunde</h2>
<form target=_top action="OnlineShop" method="POST">
<input type="Hidden" name="Command" value="changeCustomer">
<table border=1>
<tr><th>Name</th><th>Wert</th></tr>
<%
for(int i = 0; i < C_OnlineShop.C_CustomerInputFields.length; i++) {
   Object theCustomerValue ;
   if(theCustomer != null)
      theCustomerValue =
         theCustomer.getValue(C_OnlineShop.C_CustomerInputFields[i][1]) ;
   else
      theCustomerValue = null ;
%>
   <tr>
   <td><b><%=C_OnlineShop.C_CustomerInputFields[i][0]%>:</b></td>
   <td><input type="<%=C_OnlineShop.C_CustomerInputFields[i][3]%>"
name="_PA<%=C_OnlineShop.C_CustomerInputFields[i][1]%>"
size="<%=C_OnlineShop.C_CustomerInputFields[i][2]%>" value="<%=theCustomerValue%>"></
td>
   </tr>
<%}%>
</table>
<br>
<br>
<input type="Submit" value="Ihre Kundendaten ändern">
</form>
<hr>
<%
```

```
   if(theCustomer != null) {
      out.println("<h2>Bisherige Bestellungen</h2><br>") ;
      ReferenceVector theOrders = null ;
      try {
         theOrders =
            (ReferenceVector)theCustomer.getReference("Bestellung") ;
      } catch (PersistenceException pe ) {
      throw new ServletException "OnlineShop Modell fehlerhaft!" ,pe);
      }
      out.println("<table width=70" + "%" + ">");
      out.println("<tr><th align=left><b>Bestelldatum</b></th>"
         + "<th align=left><b>Bemerkung</b></th></tr>") ;
      TimeZone tz = TimeZone.getTimeZone("ECT");
      SimpleDateFormat dateformat =
         new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
      dateformat.setTimeZone(tz);
      for (Enumeration e = theOrders.elements(); e.hasMoreElements() ;) {
         PersistenceGeneric theOrder =
            (PersistenceGeneric) e.nextElement() ;
%>
         <tr>
         <td><%=dateformat.format(theOrder.getValue("Bestelldatum"))%></td>
         <td><%=theOrder.getValue("Bemerkung")%></td>
        </tr>
<%   }
     out.println("</table>");
   } else
      try {
         theCustomer =
            (PersistenceGeneric)theStore.newPersistence("OnlineKunde") ;
      } catch (PersistenceException pe ) {
      throw new ServletException(
         "Konnte neuen Kunden nicht anlegen! " ,pe) ;
      }
%>
</center><!<%@ include file="footer.html" %>
```

*Listing 12.15: JavaServer-Page ShowCustomer.jsp*

### 12.3.6 Anmelden eines Kunden

Ist ein Kunde bereits registriert, kann er sich mit seiner E-Mail-Adresse und seinem Geheimwort dem System gegenüber authentifizieren. Dies geschieht in der Anmeldung. Sie ist mit einem normalen HTML-Formular realisiert (Listing 12.16).

```
<html>
<head>
<title>OnlineShop</title>
</head>
<BODY text="#800000" link="#000080" vlink="#000080" alink="#FF0000">
<center>
<h1>Anmelden eines Kunden</h1><br>
```

```
<form action="OnlineShop" method="POST">
<input type="Hidden" name="Command" value="setCustomer">
<table border=1>
<tr><td><b>Mailadresse</b></td>
    <td><input type="Text" name="_PAEMail" size="50"></td>
</tr>
<tr><td><b>Paßwort</b></td>
    <td><input type="Password" name="_PAPassword" size="8"></td>
</tr>
</table>
<br>
<br>
<input type="Submit" value="Anmeldung">
</form>
</center>
</body>
</html>
```

*Listing 12.16: Anmeldung eines Kunden mit* `kundeLogin.html`

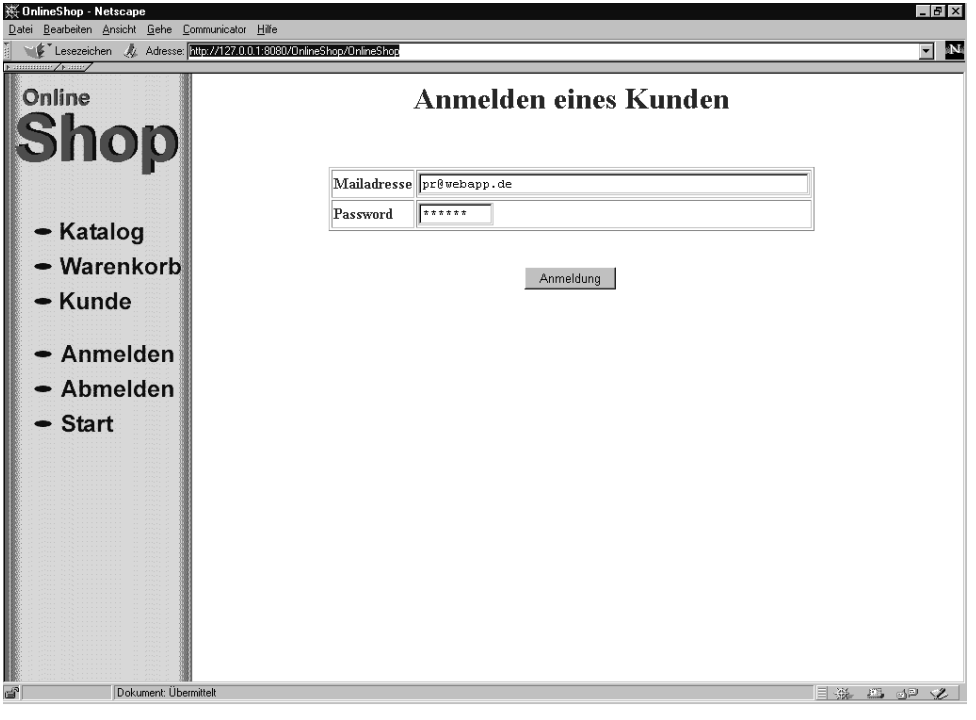


*Abbildung 12.10: Anmelden eines Kunden*

Soll jedoch eine Bestellung aufgegeben werden, obwohl der Kunde noch nicht angemeldet ist, wird der Einsatz einer JSP unerläßlich. Für diesen Fall haben wir einen

Anmeldungszwischendialog eingeschoben, der nach der gültigen Eingabe der Kundenidentifikation, zur Bestellungsanzeige zurückkehrt.

Der Trick in dieser JSP besteht darin, daß in den Hintergrundfeldern (`hidden fields`) des Formulars das Folgekommando kodiert ist.

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>

<%
// Kommando beseorgen.
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent") ;
I_SMIContext theSMIContext =
   (I_SMIContext)theSMICommand.getValue(C_OnlineShop.C_Context) ;
String theMessage=
   (String)theSMICommand.getValue(C_OnlineShop.C_Message) ;
Object[] theArguments =
   (Object[])theSMICommand.getValue(C_OnlineShop.C_Arguments) ;
PersistenceGeneric theShop =
   (PersistenceGeneric)theSMICommand.getValue(C_OnlineShop.C_Shop) ;
%>
<center>
<h1>Anmelden eines Kunden</h1><br>
<form action="OnlineShop" method="POST">
<input type="Hidden" name="Command" value="setCustomer">
<input type="Hidden" name="<%=C_OnlineShop.C_FollowUpCommand%>" value="display">
<input type="Hidden" name="<%=C_OnlineShop.C_Template%>" value="ShowOrder.jsp">
<table border=1>
<tr><td><b>Mailadresse</b></td>
    <td><input type="Text" name="_PAEMail" size="50"></td>
</tr>
<tr><td><b>Paßwort</b></td>
     <td><input type="Password" name="_PAPassword" size="8"></td>
</tr>
</table>
<br>
<br>
<input type="Submit" value="Anmeldung">
</form>
<hr>
</center>
<%@ include file="footer.html" %>
```

*Listing 12.17: Anmelden eines Kunden, wenn eine Bestellung erfolgt und in dieser `HttpSession` kein Kunde angemeldet ist.*

## 12.4 Konfigurationen

Nachdem wir die Einzelteile kodiert haben, führen wir nun alles durch die Konfigurationen zusammen. Zuerst werden dazu die Tabellens in der Datenbank angelegt, dann die Store-Konfiguration und die `PersistenceGeneric`-Objekte beschrieben. Im nächsten Schritt werden die Kommandos des OnlineShops (Abbildung 12.2) mit den Beans (Abbildung 12.4) in einer SMI-Definitionsdatei verbunden. Als letztes wird die Anwendung in der Servlet-Engine jo! registriert.

### 12.4.1 Store-Konfiguration

Zunächst werden alle Datenbank-Tabellen angelegt. Als Basis dient die Tabelle OnlineShop – in ihr werden die Shopinformationen gespeichert. Wir erinnern uns: Jeder Shop besitzt eine Anzahl an Produkten und Kunden. Jeder Kunde kann viele Bestellungen mit vielen Bestellpositionen aufgeben. Jede Position ist genau einem Produkt des Shops zugeordnet. Listing 12.18 zeigt die erforderlichen SQL-Anweisungen.

```
// Shop
CREATE TABLE OnlineShop (
   ObjectIdentifier CHAR(16) NOT NULL ,
   ObjectVersion INTEGER,
   Nummer VARCHAR(30),
   Name VARCHAR,
   Beschreibung VARCHAR,
   Inhaber VARCHAR,
   InhaberEMail VARCHAR,
   InfoURL VARCHAR,
   InfoEMail VARCHAR,
   BestellEMail VARCHAR,
   PRIMARY KEY (ObjectIdentifier),
);
INSERT INTO TypeNumber VALUES ( 'OnlineShop' , 151 );

// Produkt
CREATE TABLE OnlineProdukt (
   ObjectIdentifier CHAR(16) NOT NULL,
   ObjectVersion INTEGER,
   Nummer VARCHAR (30),
   Stichworte VARCHAR (50),
   Beschreibung VARCHAR,
   VKMenge INTEGER,
   VKPreis DECIMAL,
   InfoURL VARCHAR,
   BildURL VARCHAR,
   ShopID CHAR(16) NOT NULL,
   PRIMARY KEY (ObjectIdentifier),
   FOREIGN KEY (ShopID) REFERENCES OnlineShop (ObjectIdentifier)
);
```

```
INSERT INTO TypeNumber VALUES ( 'OnlineProdukt' , 152 );

// Kunde
CREATE TABLE OnlineKunde (
   ObjectIdentifier CHAR(16) NOT NULL,
   ObjectVersion INTEGER,
   Name VARCHAR (50),
   Email VARCHAR,
   Password VARCHAR (8),
   PLZ CHAR(5),
   Ort VARCHAR (50),
   Strasse VARCHAR (50),
   Bank VARCHAR,
   Bankleitzahl VARCHAR (20),
   Kontonummer VARCHAR (20),
   ShopID CHAR(16) NOT NULL,
   PRIMARY KEY (ObjectIdentifier),
   FOREIGN KEY (ShopID) REFERNECES OnlineShop (ObjectIdentifier)
);
INSERT INTO TypeNumber VALUES ( 'OnlineKunde' , 153 );

// Bestellung
CREATE TABLE OnlineBestellung (
   ObjectIdentifier CHAR(16) NOT NULL,
   ObjectVersion INTEGER,
   Bestelldatum TIMESTAMP,
   Bank VARCHAR,
   Bankleitzahl VARCHAR (20),
   Kontonummer VARCHAR (20),
   PLZ CHAR(5),
   Ort VARCHAR (50),
   Strasse VARCHAR (50),
   Bemerkung VARCHAR,
   KundeID CHAR(16) NOT NULL,
   ShopID CHAR(16) NOT NULL,
   PRIMARY KEY (ObjectIdentifier),
   FOREIGN KEY (KundeID) REFERENCES OnlineKunde (ObjectIdentifier),
   FOREIGN KEY (ShopID) REFERENCES OnlineShop (ObjectIdentifier)
);
INSERT INTO TypeNumber VALUES ( 'OnlineBestellung' , 154);

// Bestellposition
CREATE TABLE OnlineBestellposition (
   ObjectIdentifier CHAR(16) NOT NULL,
   ObjectVersion INTEGER,
   Menge INTEGER,
   Betrag DECIMAL,
   ProduktID CHAR(16) NOT NULL,
   BestellungID CHAR(16) NOT NULL,
```

```
   PRIMARY KEY (ObjectIdentifier),
   FOREIGN KEY (BestellungID) REFERENCES OnlineBestellung (ObjectIdentifier),
   FOREIGN KEY (ProduktID) REFERENCES OnlineProdukt (ObjectIdentifier)
);
INSERT INTO TypeNumber VALUES ( 'OnlineBestellposition' , 155) ;
```

*Listing 12.18: SQL-Anweisungen zum Anlegen der Tabellen des OnlineShops*

Nachdem die Tabellen nun angelegt worden sind, müssen wir noch die Store-Konfigurationsdatei `OnlineShop.store` anlegen (Listing 12.19 bis Listing 12.24). Hier wird die Verbindung zur Datenbank in der Sektion `JDBCConnection` definiert. Im Anschluß werden die Definitionen der Geschäftsobjekte in jeweils separaten Dateien eingebunden.

```
{
   StoreClass = "de.webapp.Framework.SMICommandListener.SMIStore" ;
   JDBCConnection = {
      // Beispiel-Konfiguration
      DriverClass = "solid.jdbc.SolidDriver" ;
      URLConnect = "jdbc:solid://localhost:1313" ;
      Properties = {
         user = "web" ;
         password = "app" ;
      }
   } ;
   Types =
   {
      Shop =             #include("Shop.persistence");
      Kunde =            #include("Kunde.persistence");
      Produkt =          #include("Produkt.persistence");
      Bestellung =       #include("Bestellung.persistence");
      Bestellposition =  #include("Bestellposition.persistence");
   }
}
```

*Listing 12.19: Store-Konfiguration `OnlineShop.store`*

```
{
   Type ={
      Name = "OnlineShop" ;
      Class = "de.webapp.Framework.SMICommandListener.SMIPersistenceGeneric";
      Entity = "OnlineShop";
      EntityAlias = "t0" ;
      Attributes = (
         "ObjectIdentifier",
         "ObjectVersion",
         { Name = "Nummer" ;       Generic = "YES" ; Key = "YES" ;},
         { Name = "Name" ;         Generic = "YES" ; },
         { Name = "Beschreibung" ; Generic = "YES" ; },
         { Name = "Inhaber" ;      Generic = "YES" ; },
         { Name = "InhaberEMail" ; Generic = "YES" ; },
         { Name = "InfoURL" ;      Generic = "YES" ; },
         { Name = "BestellEMail" ; Generic = "YES" ; },
```

```
      ) ;
      Associations = (
      { Name = "Produkt" ; Type = "ToMany" ; Generic = "YES" ;
        Modified = "YES" ;
        ResultType = "OnlineProdukt" ; Key = "ShopID" ;
        QualifierExtension = " ORDER BY t0.Nummer " ; },
      { Name = "Kunde" ;   Type = "ToMany" ; Generic = "YES" ;
        Modified = "YES" ;
         ResultType = "OnlineKunde" ;   Key = "ShopID" ; },
    ) ;
   } ;
}
```

*Listing 12.20: Definition des GenericPersistence- OnlineShop: Shop.persistence*

```
{
   Type = {
      Name = "OnlineProdukt" ;
      Class = "de.webapp.Framework.SMICommandListener.SMIPersistenceGeneric";
      Entity = "OnlineProdukt";
      EntityAlias = "t0" ;
      Attributes =
      (
         "ObjectIdentifier" ,
         "ObjectVersion",
         { Name = "Nummer" ; Generic = "YES" ; },
         { Name = "Stichworte" , Generic = "YES" ; },
         { Name = "Beschreibung" ; Generic = "YES" ; },
         { Name = "VKMenge" ; Generic = "YES" ;
           TypeClass = "java.lang.Integer" ; },
         { Name = "VKPreis" ; Generic = "YES" ;
           TypeClass = "java.math.BigDecimal" ; },
         { Name = "InfoURL" ; Generic = "YES" ; },
         { Name = "BildURL" ; Generic = "YES" ; },
         { Name = "ShopID" ; Generic = "YES" ; }
      ) ;
      Associations = (
      { Name = "Shop" ; Type = "ToOne" ; Generic = "YES" ;
        ResultType = "OnlineShop" ;
      }
    ) ;
   } ;
}
```

*Listing 12.21: Definition von OnlineProdukt in Produkt.persistence*

```
{
   Type = {
      Name = "OnlineKunde" ;
      Class = "de.webapp.Framework.SMICommandListener.SMIPersistenceGeneric";
      Entity = "OnlineKunde";
      EntityAlias = "t0" ;
```

```
      Attributes = (
         "ObjectIdentifier",
         "ObjectVersion",
         { Name = "Name" ; Generic = "YES" ; },
         { Name = "EMail" ; Generic = "YES" ; },
         { Name = "Password" ; Generic = "YES" ; },
         { Name = "PLZ" ; Generic = "YES" ; },
         { Name = "Ort" ; Generic = "YES" ; },
         { Name = "Strasse" ; Generic = "YES" ;},
         { Name = "Bankleitzahl" ; Generic = "YES" ; },
         { Name = "Bank" ; Generic = "YES" ; },
         { Name = "Kontonummer" ; Generic = "YES" ; },
         { Name = "ShopID" ; Generic = "YES" ; }
      ) ;
      Associations = (
      { Name = "Bestellung" ; Type = "ToMany" ; Generic = "YES" ;
        Modified = "YES" ;
        ResultType = "OnlineBestellung" ; Key = "KundeID" ;
        QualifierExtension = " ORDER BY t0.Bestelldatum DESC" ; },
      { Name = "Shop" ; Type = "ToOne" ; Generic = "YES" ;
        ResultType = "OnlineShop" ; }
    ) ;
   } ;
}
```

*Listing 12.22: Definition von `OnlineKunde` in `Kunde.persistence`*

```
{
   Type =
   {
      Name = "OnlineBestellung" ;
      Class = "de.webapp.Framework.SMICommandListener.SMIPersistenceGeneric";
      Entity = "OnlineBestellung";
      EntityAlias = "t0" ;
      Attributes = (
         "ObjectIdentifier",
         "ObjectVersion",
         { Name = "Bestelldatum" ; Generic = "YES" ;
           TypeClass = "java.sql.Timestamp" ; },
         { Name = "PerNachname" ; Generic = "YES" ;
           TypeClass = "java.lang.Integer" ; },
         { Name = "Bankleitzahl" ; Generic = "YES" ; },
         { Name = "Bank" ; Generic = "YES" ;},
         { Name = "Kontonummer" ; Generic = "YES" ; },
         { Name = "PLZ" ; Generic = "YES" ; },
         { Name = "Ort" ; Generic = "YES" ; },
         { Name = "Strasse" ; Generic = "YES" ; },
         { Name = "Bemerkung" ; Generic = "YES" ; },
         { Name = "KundeID" ; Generic = "YES" ; }
         { Name = "ShopID" ; Generic = "YES" ; },
      ) ;
      Associations = (
```

```
      { Name = "Bestellposition" ; Type = "ToMany" ; Generic = "YES" ;
        Modified = "YES" ;
        ResultType = "OnlineBestellposition" ; Key = "BestellungID" ;},
      { Name = "Kunde" ; Type = "ToOne" ; Generic = "YES" ;
        ResultType = "OnlineKunde" ; },
      { Name = "Shop" ; Type = "ToOne" ; Generic = "YES" ;
      ResultType = "OnlineShop" ;}
      ) ;
   } ;
}
```

*Listing 12.23: Definition von `OnlineBestellung` in `Bestellung.persistence`*

```
{
   Type = {
      Name = "OnlineBestellposition" ;
      Class = "de.webapp.Framework.SMICommandListener.SMIPersistenceGeneric";
      Entity = "OnlineBestellposition";
      EntityAlias = "tO" ;
      Attributes = (
         "ObjectIdentifier",
         "ObjectVersion",
         { Name = "Menge" ;     Generic = "YES" ;
           TypeClass = "java.lang.Integer" ;},
         { Name = "Betrag" ;    Generic = "YES" ;
           TypeClass = "java.math.BigDecimal" ; },
         { Name = "ProduktID" ; Generic = "YES" ; },
         { Name = "BestellungID" ; Generic = "YES" ; }
      ) ;
    Associations = (
       { Name = "Produkt" ; Type = "ToOne" ; Generic = "YES" ;
          ResultType = "OnlineProdukt" ; },
       { Name = "Bestellung" ; Type = "ToOne" ; Generic = "YES" ;
          ResultType = "OnlineBestellung" ; }
      ) ;
   } ;
}
```

*Listing 12.24: Definition von `OnlineBestellposition` in `Bestellposition.persistence`*

### 12.4.2 SMI-Konfiguration

Jetzt müssen wir noch die SMI-Definitionsdatei erstellen. Der Inhalt dieser Datei legt fest, welche Kommandos auf welche Methoden und welchen Listener gelenkt werden. Außerdem können für jedes Kommando weitere Standardparameter definiert werden.

Zu Beginn werden das Default-Kommando und kommandoübergreifende Standard-Parameter angegeben sowie die Listener mit Namen und Klasse festgelegt. Danach folgen die Kommandodefinitionen.

```
{
   DefaultCommand = "display";
   Values = {
      _FollowUpCommand = "display";
      _TemplateName      = "index.html";
   }
   Listener = {
      OnlineShopBean = {
         Class = "de.webapp.OnlineShop.OnlineShopBean" ;
      },
      OnlineShopProductBean = {
         Class = "de.webapp.OnlineShop.OnlineShopProductBean" ;
      },
      OnlineShopOrderBean = {
         Class = "de.webapp.OnlineShop.OnlineShopOrderBean" ;
      },
      OnlineShopCustomerBean = {
         Class = "de.webapp.OnlineShop.OnlineShopCustomerBean" ;
      },
      DisplayBean = {
         Class = "de.webapp.Framework.SMICommandListener.DisplayBean";
      }
   };
   Commands   = {
      // Anzeige
      display   = {
         MethodName = "display";
         Listener = "DisplayBean" ;
      } ;
      // Anzeige des Shops
      // Öffne Frame und zeige Menü und Shopinformation.
      startShop   = {
         MethodName = "display";
         Listener = "DisplayBean" ;
         Values = {
            _TemplateName = "index.html" ;
         } ;
      } ;
      // Zeige Menü.
      showMenu= {
         MethodName = "display";
         Listener = "DisplayBean" ;
         Values = {
            _TemplateName = "Menue/Uebersicht.html" ;
         } ;
      } ;
      // Zeige Shopinformation.
      showShop= {
         MethodName = "executeContextFollowUp";
         Listener = "OnlineShopBean" ;
         Values = {
```

```
      _TemplateName = "ShowShop.jsp" ;
   } ;
} ;
// Anzeige des Produktkatalogs.
showProducts= {
   MethodName = "showProducts";
   Listener = "OnlineShopProductBean" ;
   Values = {
      _TemplateName = "ShowProduct.jsp" ;
   } ;
} ;
// Bestellung
// Neue Bestellung in Warenkorb aufnehmen.
newOrder= {
   MethodName = "newOrder";
   Listener = "OnlineShopOrderBean" ;
   Values = {
      _TemplateName = "orderOneProdukt.html" ;
   } ;
} ;
// Zeige aktuelle Bestellung.
showOrder= {
   MethodName = "executeContextFollowUp";
   Listener = "OnlineShopOrderBean" ;
   Values = {
      _TemplateName = "ShowOrder.jsp" ;
   } ;
} ;
// Ändern der Bestellposition.
 changeOrder= {
   MethodName = "changeOrder";
   Listener = "OnlineShopOrderBean" ;
   Values = {
      _TemplateName = "ShowOrder.jsp" ;
   } ;
} ;
// Bestellung in den Shop aufnehmen.
sendOrder= {
   MethodName = "sendOrder";
   Listener = "OnlineShopOrderBean" ;
   Values = {
      _TemplateName = "sendOrder.html" ;
   } ;
} ;
// Befehle des Kunden
// Zeige Kundeninformation.
showCustomer= {
   MethodName = "executeContextFollowUp";
   Listener = "OnlineShopCustomerBean" ;
```

```
         Values = {
            _TemplateName = "ShowCustomer.jsp" ;
         } ;
      } ;
      // Ändere Kunden.
      changeCustomer= {
         MethodName = "changeCustomer";
         Listener = "OnlineShopCustomerBean" ;
         Values = {
            _TemplateName = "index.html" ;
         } ;
      } ;
      // Zeige Anmeldung eines Kundens.
      showLoginCustomer= {
         MethodName = "display";
         Listener = "DisplayBean" ;
         Values = {
            _TemplateName = "loginCustomer.html" ;
         } ;
      } ;
      // Kunde anmelden.
      setCustomer= {
         MethodName = "setCustomer";
         Listener = "OnlineShopCustomerBean" ;
         Values = {
            _TemplateName = "ShowShop.jsp" ;
         } ;
      } ;
      // Kunde abmelden.
      logoutCustomer= {
         MethodName = "refreshShop";
         Listener = "OnlineShopBean" ;
         Values = {
            _TemplateName = "index.html" ;
         } ;
      };
   }
}
```

*Listing 12.25: SMI-Definitionsdatei des OnlineShops:* `OnlineShop.smi`

Für den OnlineShop wird beim Start der Anwendung der Store und Shop in den Kontext eingetragen. Diesem Zweck dient die Klasse `OnlineShopContext`. In der Datei `OnlineShop.con` (Listing 12.26) steckt die notwendige Definition für den `SMIContextManager` (Kapitel 8).

```
{
   Class = "de.webapp.OnlineShop.OnlineShopContext";
   Values = {
      _Storename = "OnlineShop.store";
```

```
        _Shopnummer = "1";
        _PageSize = "3";
    }
}
```

*Listing 12.26: Konfigurationsdatei `OnlineShop.con` für den `OnlineShopContext`*

Als Abschluß aller Definitionen fehlt noch die Registrierung bei der Servlet-Engine. Der Eintrag für jo! ist in Listing 12.27 beschrieben.

```
# OnlineShop
servlet.OnlineShop.code=de.webapp.OnlineShop.OnlineShopServlet
servlet.OnlineShop.initArgs=EventSwitch=OnlineShop,Context=OnlineShop
servlet.OnlineShop.aliases=/OnlineShop/OnlineShop
```

*Listing 12.27: OnlineShop-Eintrag in der `servlet.properties`-Datei von jo!*

Der OnlineShop sollte nach dem Start von jo! unter dem URL `http://<hostname>:<port>/OnlineShop/OnlineShop` bereitstehen.

# 13 Chat

In unserem letzten Beispiel werden wir einen einfachen webbasierten Chat-Raum entwickeln. Auch hier müssen wir zunächst Anforderungen und Abläufe definieren. Die Anwendungsfälle (Abbildung 13.1). zeigen, welche Vorstellung wir von dem Chat haben. Ein Chat-Nutzer soll folgende Handlungen ausführen können:

- sich registrieren, um seinen Namen zu schützen
- den Chat-Raum betreten und wieder verlassen
- Nachrichten an ausgewählte Teilnehmer senden
- Nachrichten an alle Teilnehmer senden
- seine persönlichen Daten (Name, Paßwort und Sendefarbe) ändern

Besonderes Merkmal dieses Chat-Raums soll sein, daß der Nutzer nur über einen JavaScript- und framefähigen Browser verfügen muß. Java in Form eines Applets auf der Client-Seite ist nicht erforderlich.

Um das Beispiel einfach zu gestalten, sehen wir davon ab, mehrere Chat-Räume zu eröffnen.

## 13.1 Abläufe

Nach dem Start des Chats soll als erstes ein Login-Dialog angezeigt werden. Hier hat der Nutzer die Wahl, sich entweder anzumelden oder sich zunächst zu registrieren (Abbildung 13.2). Zur Information des Nutzers wird außerdem eine Liste der Leute angezeigt, die sich im Raum befinden.

Entscheidet er sich für das Registrieren, erscheint ein Dialog, in dem er seinen Namen, sein Paßwort sowie eine Sendefarbe eingeben muß. Für den Fall, daß der gewählte Name bereits belegt ist oder ein anderer Fehler vorliegt, erscheint eine Fehlermeldung. Verläuft die Registrierung erfolgreich, gelangt der Nutzer zu einem Menü-Dialog, der das Betreten des Raumes, das Abmelden und Ändern der Nutzerdaten erlaubt. Zu diesem wäre er auch nach dem Einloggen gelangt. Auch hier wird eine aktuelle Liste der Chat-Teilnehmer angezeigt.

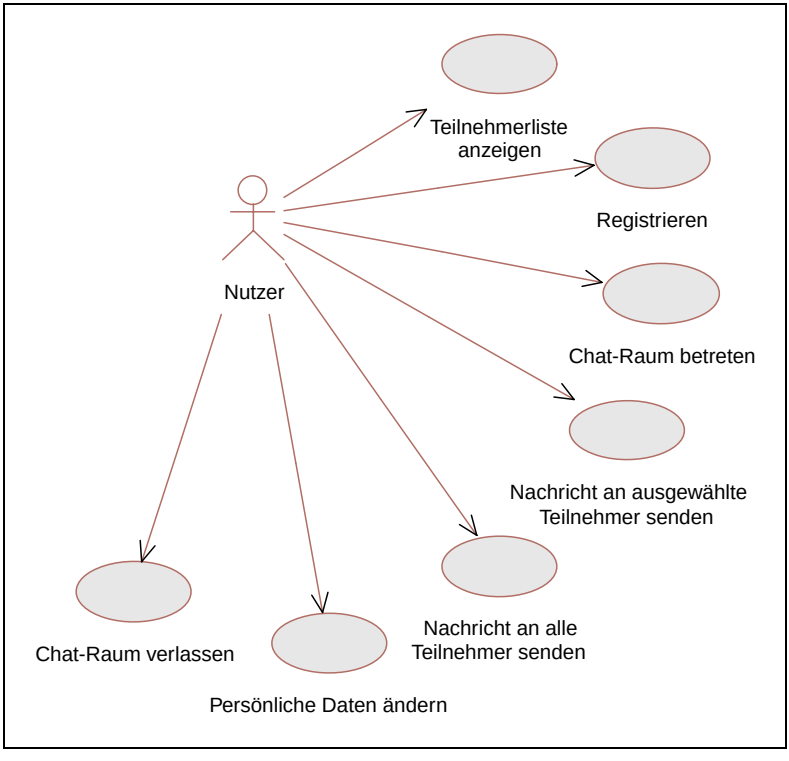


*Abbildung 13.1: Anwendungsfälle des Chats*

Beim Betreten des Raums wird ein Frameset mit zwei Frames geöffnet. Der rechte Frame zeigt fortlaufend die gesendeten Nachrichten an. Der linke Frame enthält einen Sende-Dialog mit einem Eingabefeld, einer Farbwahl-Dropdown-Liste, einer Nutzerliste mit Checkboxes und einem Sende-Knopf. Außerdem befindet sich im Dialog noch ein Verweis zum Verlassen des Raumes.

Das Senden einer Nachricht führt zur Anzeige der Nachricht im rechten Frame aller Nutzer, deren Checkboxes angekreuzt waren. Dabei wird die Nachricht in der gerade ausgewählten Farbe gedruckt.

Nach dem Verlassen des Raumes wird dem Nutzer wieder der Menü-Dialog präsentiert. Wählt er den Daten-Ändern-Verweis, gelangt er zu einem Dialog ähnlich dem Registrierungsdialog. Nach dem erfolgreichen Ändern seiner Daten gelangt der Nutzer wieder zum Menü-Dialog. Von dort aus kann er sich ausloggen. Anschließend wird wieder der Login-Dialog angezeigt.

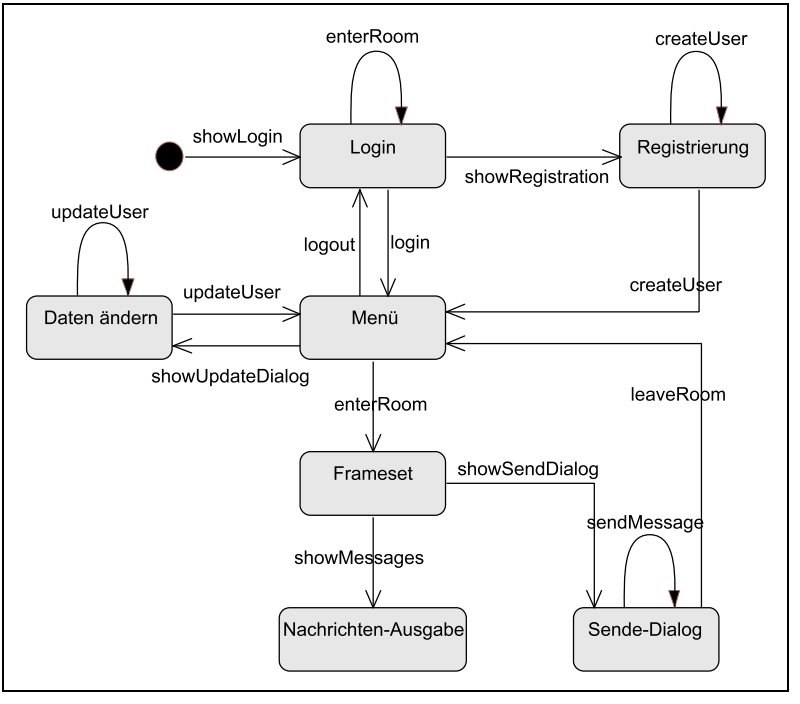


*Abbildung 13.2: Ablaufdiagramm für den Chat*

Die sich ergebenden Kommandos sind in Tabelle 13.1 noch einmal genau aufgelistet.

| **Kommando** | **Parameter** | **Bedeutung** |
|---|---|---|
| `showLogin` | - | Es wird ein Login-Dialog angezeigt. Auslösbare Kommandos sind `enterRoom` und `showRegistration`. |
| `showRegistration` | - | Zeigt einen Dialog zum Eingeben der persönlichen Daten. Mögliches Folgekommando ist `createUser`. |
| `createUser` | Persönliche Daten (Name, Paßwort, Sendefarbe) | Legt einen Nutzer an. Tritt dabei ein Fehler auf, wird dieser angezeigt und eine erneute Eingabe der Daten ermöglicht. Tritt kein Fehler auf, wird der Menü-Dialog angezeigt. Der Nutzer ist nun authentifiziert. Mögliche Kommandos sind `showUpdateDialog`, `logout` und `enterRoom`. |
| `login` | Name und Paßwort | Authentifiziert den Nutzer. Passen Name und Paßwort zusammen und entsprechen einem `User`-Objekt, wird der Menü-Dialog angezeigt. Ist das nicht der Fall, erscheint eine Fehlermeldung und der Nutzer kann Name und Paßwort erneut eingeben. |

*Tabelle 13.1: Beschreibung der Kommandos des Chats*

| Kommando | Parameter | Bedeutung |
|---|---|---|
| `enterRoom` | - | Registriert den Nutzer im Raum. Anschließend wird ein Frameset aufgerufen, das zwei Frames öffnet. Der linke Frame wird mit dem Kommando `showSendDialog`, der rechte mit dem Kommando `showMessages` aufgerufen. |
| `showMessages` | - | Zeigt die Nachrichten für den Nutzer. |
| `showSendDialog` | - | Zeigt ein Formular zum Senden einer Nachricht. Von diesem Dialog aus können die Kommandos `sendMessage` und `leaveRoom` aufgerufen werden. |
| `sendMessage` | Nutzer, an den die letzte Nachricht gesendet wurde | Zeigt das Sendeformular wie in `showSendDialog` beschrieben. Zusätzlich sind die Checkboxes der Nutzer angekreuzt, an die die letzte Nachricht adressiert war. Mögliche folgende Kommandos sind die gleichen wie in `showSendDialog`. |
| `showUpdateDialog` | - | Zeigt einen Dialog zum Ändern der Nutzerdaten. `updateUser` ist das einzige Kommando, das von diesem Dialog aus ausgelöst werden kann. |
| `updateUser` | Persönliche Daten (Name, Paßwort, Sendefarbe) | Ändert die Daten eines Nutzers. Tritt dabei ein Fehler auf, wird der Eingabedialog erneut angezeigt und eine Eingabe der Daten ermöglicht. Tritt kein Fehler auf, wird der Menü-Dialog angezeigt. |
| `leaveRoom` | - | Löscht die Registrierung des Nutzers im Raum und führt zurück zum Menü-Dialog. Mögliche Aktionen sind dann `showUpdateDialog`, `logout` und `enterRoom`. |
| `logout` | - | Läßt die Authentifizierung verfallen und führt zum Login-Dialog. |

*Tabelle 13.1: Beschreibung der Kommandos des Chats (Fortsetzung)*

Beim genauen Durchlesen der Kommando-Beschreibungen ist Ihnen sicher aufgefallen, daß wir zwei Ausprägungen des Anmeldens kennen. Um überhaupt am Chat teilnehmen zu können, muß sich der Nutzer authentifizieren. Das bedeutet, daß er seinen Namen und sein Paßwort eingeben muß. Paßt das Paßwort zum Namen, hat der Nutzer sich erfolgreich authentifiziert. Das bedeutet jedoch nicht, daß er den Chat-Raum betreten hat. Erst das explizite Betreten des Raums führt auch zur Teilnahme am Chat. Entsprechend führt das Verlassen des Chat-Raums nicht zum Erlöschen der Authentifizierung. Erst das explizite Abmelden führt zu einem Erlöschen der Authentifizierung.

Wir nehmen diese Unterscheidung aus zweierlei Gründen vor. Zum einen ist das An- und Abmelden ein oft benötigter Mechanismus, bei dem es sich lohnt, ihn hier noch einmal beispielhaft zu verwirklichen. Zum anderen schaffen wir durch die Trennung von Login und Betreten des Raums die Möglichkeit, relativ einfach weitere Räume hinzufügen zu können.

## 13.2 Design der Klassen

Bevor wir uns ans Kodieren machen, müssen wir feststellen, welche Objekte am Chat beteiligt sind. Zunächst einmal ist da natürlich der Nutzer – er wird mit einem `User`-Objekt abgebildet, das in einem Store gespeichert werden kann. Weiter benötigen wir einige Objekte für die Steuerung des Chats. Es erscheint uns sinnvoll, die Ablaufsteuerung (Anmelden, Abmelden und Authentifizieren) des Chats in dem Bean `ChatRoom` zu kapseln. Des weiteren sollen der Mechanismus zum Versenden von Nachrichten genauso vom Bean `ChatDispatcher` und die Funktionen zum Anlegen und Ändern von Nutzer-Objekten vom Bean `ChatRegistration` verwirklicht werden. Abbildung 13.3 zeigt das Klassendiagramm des Chats. Die einzelnen Klassen werden im folgenden vorgestellt.

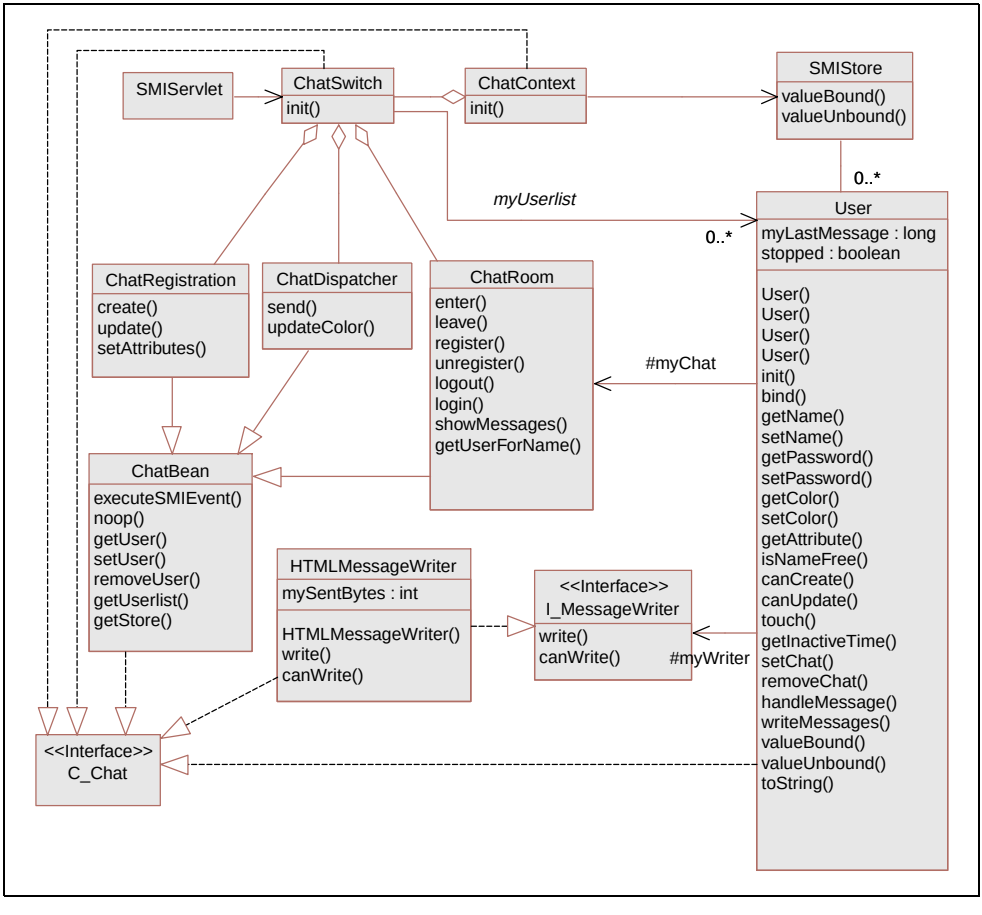


*Abbildung 13.3: Klassendiagramm des Chats. Vererbungsbeziehungen, das `DisplayBean` sowie die Schnittstellen `HttpSessionBindingListener` und `I_SMIContextBindingListener` wurden weggelassen.*

### 13.2.1 Schnittstelle für Konstanten

Wie bereits in den vorherigen beiden Kapiteln ausgeführt, benutzt auch der Chat eine Schnittstelle zur Definition von Konstanten. Da diese in allen folgenden Klassen benötigt werden, ist die Schnittstelle `C_Chat` bereits hier abgedruckt (Listing 13.1), obwohl einige Konstanten erst später erklärt werden.

```
package de.webapp.Chat ;

public interface C_Chat {

/** Konstante für den Nutzer */
public static final String C_User = "_User";
/** Konstante für den Store */
public static final String C_Store = "_Store";
/** Konstante für den StoreName */
public static final String C_StoreName = "_StoreName";
/** Konstante für den DefaultStoreName */
public static final String C_DefaultStoreName = "Chat.store";
/** Konstante für die Nutzerliste */
public static final String C_Userlist = "_Userlist";
/** Konstante für den Nutzername */
public static final String C_Username = "_Username";
/** Konstante für das Paßwort */
public static final String C_Password = "_Password";
/** Konstante für das Paßwort */
public static final String C_Password2 = "_Password2";
/** Konstante für die Farbe */
public static final String C_Color = "_Color";
/** Konstante für die Empfänger */
public static final String C_To = "_To";
/** Konstante für die Nachricht */
public static final String C_Message = "_Message";
/** Konstante für "an alle" */
public static final String C_All = "_All";
/** Parameter für Folgekommando  */
public static final String C_FollowUpCommand = "_FollowUpCommand";
/** Parameter für Template der JSP- und HTML-Anzeige  */
public static final String C_Template = "_TemplateName";
/** Parameter für Fehlermeldungen */
public static final String C_ErrorMessage = "_ErrorMessage";

/** Maximale Anzahl an Bytes, die von einem HTMLMessageWriter
    geschrieben werden dürfen */
public static final int C_MaxBytes = 4096;
/** Maximale Zeit, bevor ein Nutzer automatisch ausgeloggt wird */
public static final long C_MaxInactiveTime = 1000 * 60 * 15;

/** Farbkonstanten */
public static final String C_Colors[] = new String[] {
   "#000000", "#ff0000", "#0000ff",
```

```
   "#ffff00", "#008080", "#800000",
   "#ff00ff"
};
public static final String C_ColorNames[] = new String[] {
   "schwarz", "rot", "blau",
   "gelb", "teal", "maroon",
   "fuchsia"
};

} // Ende der Schnittstelle
```

*Listing 13.1: Die Schnittstelle* `C_Chat`

### 13.2.2 ChatContext

Alle Beans benötigen Zugriff auf bestimmte Ressourcen. An erster Stelle steht der Store, in dem die Nutzer-Objekte gespeichert sind. Obwohl wir nur einen einzelnen Chat-Raum mit nur einem `I_SMIEventSwitch` entwickeln wollen, ist es sinnvoll, den Store im `I_SMIContext` zu hinterlegen. Bei einer späteren Erweiterung des Chats auf mehrere Räume mit jeweils eigenen `I_SMIEventSwitches` könnten diese sich den Store teilen. Der Store sollte zur Initialisierung des `I_SMIContextes` geöffnet und bei der Freigabe des `I_SMIContextes` wieder geschlossen werden. Um dies zu erreichen, machen wir zweierlei. Zum einen schreiben wir eine Klasse `ChatContext` (Listing 13.2), die von `SMIContext` erbt. Sie öffnet in ihrer `init()`-Methode den Store und hinterlegt ihn unter dem Schlüssel `C_Store`. Zum anderen benutzen wir einen `SMIStore` aus dem Paket `de.webapp.Framework.SMICommandListener`. Wird ein solcher Store von einer `HttpSession`, einem `I_SMIEventSwitch` oder einen `I_SMIContext` entbunden, wird er automatisch geschlossen.

```
package de.webapp.Chat ;

import de.webapp.Framework.ConfigManager.*;
import de.webapp.Framework.SMI.*;
import de.webapp.Framework.StoreFactory.StoreFactory;
import de.webapp.Framework.Persistence.*;
import java.util.Hashtable;

public class ChatContext extends SMIContext implements C_Chat {

/** Öffne Store des Chats */
public void init(String aContextName,
   Hashtable aValues) throws SMIException {
   super.init(aContextName, aValues);
   // Lese Storenamen
   String theStorename = (String)getValue(C_StoreName);
   // Greife gegebenenfalls auf Standardnamen zurück
   if(theStorename == null) {
      theStorename = C_DefaultStoreName;
   }
   I_Store theStore = null;
```

```
   // Öffne den Store
   try {
      Configuration theStoreConfig =
         ConfigManager.getConfigManager().getConfiguration(
            aContextName, theStorename, true);
      theStore = StoreFactory.getInstance(theStoreConfig);
   }
   catch (PersistenceException pe) {
      throw new SMIException(pe);
   }
   // Hinterlege den Store als Wert dieses Kontextes
   // unter dem Schlüssel C_Store
   putValue(C_Store, theStore);
}
} // Ende der Klasse
```

*Listing 13.2: Die Klasse `ChatContext`*

### 13.2.3 ChatSwitch

Nach der Verwaltung des Stores geklärt ist, wenden wir uns den Nutzern zu. Zur Verwaltung der Teilnehmer eines Chats, ist es notwendig, über eine Liste mit allen Nutzern des Raums zu verfügen. Diese Liste soll allen `I_SMICommandListenern` dieses Chat-Raums zur Verfügung stehen und wird daher an den `I_SMIEventSwitch` gehängt. Genau wie im Fall des `ChatContext` erbt `ChatSwitch` (Listing 13.3) von `SMIEventSwitch` und überschreibt die `init()`-Methode. Die Nutzerliste – eine einfache Hashtabelle – wird unter dem Schlüssel `C_Userlist` hinterlegt.

```
package de.webapp.Chat ;

import de.webapp.Framework.SMI.* ;
import java.util.Hashtable;
public class ChatSwitch extends SMIEventSwitch implements C_Chat {

/** Fügt dem Switch eine Nutzerliste hinzu. */
public void init(String aSwitchName, I_SMIContext aSMIContext)
   throws SMIException {
   super.init(aSwitchName, aSMIContext);
   putValue(C_Userlist, new Hashtable());
}
} // Ende der Klasse
```

*Listing 13.3: Die Klasse `ChatSwitch`*

### 13.2.4 ChatBean

Um von unseren noch zu schreibenden `I_SMICommandListenern` komfortabel auf Nutzerliste und Store zugreifen zu können, schreiben wir eine Oberklasse `ChatBean`, die über entsprechende Methoden verfügt. Außerdem wird im `ChatBean` die Verwaltung des Nutzer-Objekts gekapselt sowie die `executeSMIEvent()`-Methode überschrieben, um die Nutzerliste als Standardwert im `SMICommand` setzen und Folgekommandos automatisch

ausführen zu können. Aus Performance-Gründen verzichten wir darauf, jeden Aufruf eines Kommandos mit dem Store-Objekt zu synchronisieren. Statt dessen synchronisieren wir lediglich die tatsächlichen Manipulationen sowie das Lesen. Auswirkung dieser Lösung ist, daß Nutzer-Objekte nach dem Lesen beziehungsweise Manipulieren und vor der Ausgabe von anderen Threads verändert werden können. Für unseren Chat ist ein solches Verhalten jedoch akzeptabel, da der Store dadurch nicht in einen inkonsistenten Zustand gelangen kann.

Bleibt noch zu erwähnen, wie die Authentifizierung der Nutzer erfolgt. Jeder Nutzer wird durch ein Nutzer-Objekt repräsentiert werden. So lange er also eingeloggt ist, muß sein Nutzer-Objekt einem bestimmten Bereich zugeordnet sein. Daher muß das Nutzer-Objekt in der `HttpSession` hinterlegt werden. Dies geschieht mittels der Methode `setUser()`. Um das Objekt wieder zu erlangen, benutzen wir `getUser()`; um es aus der Session zu entfernen, `removeUser()`. In allen drei Fällen wird als Schlüssel zur `HttpSession` die Konstante `C_User` benutzt.

Bemerkenswert ist noch die Methode `noop()`. Sie wird benötigt, wenn für ein Kommando bereits die in `executeSMIEvent()` kodierte Funktionalität ausreicht. Dies ist beispielsweise der Fall, wenn die Nutzerliste angezeigt werden soll.

```
package de.webapp.Chat;

import java.util.Hashtable;
import de.webapp.Framework.SMI.*;
import de.webapp.Framework.Persistence.*;

/** Basisklasse für weitere ChatBeans */
public class ChatBean extends SMICommandListener implements C_Chat {
/** Setzt die Nutzerliste und führt Folgekommando aus */
public void executeSMIEvent(SMIEvent aSMIEvent) {
   SMICommand aSMICommand = (SMICommand)aSMIEvent;
   // Nutzerliste setzen
   aSMICommand.putValue(C_Userlist, getUserlist());
   // Kommando ausführen
   super.executeSMIEvent(aSMICommand);
   // Gegebenenfalls Folgekommando ausführen
   String theFollowUpCommand =
     (String)aSMICommand.getValue(C_FollowUpCommand);
   if (theFollowUpCommand != null) {
      aSMICommand.setCommand(theFollowUpCommand);
      aSMICommand.removeValue(C_FollowUpCommand);
      getSMIEventSwitch().executeSMIEvent(aSMICommand);
   }
}
/** Leere Methode, falls executeSMIEvent() ausreicht */
public void noop(SMICommand aSMICommand) {}
/** Gibt das Nutzer-Objekt aus der Session zurück */
protected User getUser(SMICommand aSMICommand) {
```

```
   return (User)aSMICommand.getSessionValue(C_User);
}
/** Hinterlegt ein Nutzer-Objekt in der Session */
protected void setUser(SMICommand aSMICommand, User aUser) {
   aSMICommand.putSessionValue(C_User, aUser);
}
/** Entfernt ein Nutzer-Objekt aus der Session */
protected void removeUser(SMICommand aSMICommand) {
   aSMICommand.removeSessionValue(C_User);
}
/** Gibt die Nutzerliste des Switches zurück */
protected Hashtable getUserlist() {
   return (Hashtable)getSMIEventSwitch().getValue(C_Userlist);
}
/** Gibt den Store des I_SMIContextes zurück */
protected I_Store getStore() {
   return (I_Store)getSMIEventSwitch().getSMIContext().getValue(C_Store);
}
} // Ende der Klasse
```

*Listing 13.4: Die Klasse `ChatBean`*

### 13.2.5 Nachrichten

In einem Chat dreht sich alles um das Versenden von Nachrichten. Wir wollen diese als Objekte auffassen und durch die Klasse `Message` abbilden. Jede `Message` kennt einen Absender und einen Nachrichtentext (Listing 13.5).

```
package de.webapp.Chat ;
/** Kapselt eine Nachricht */
public class Message {

protected User myFrom;
protected String myText;

/** Konstruktor */
public Message(User aFrom, String aText) {
   myFrom = aFrom;
   myText = aText;
}
/** Gibt den Absender zurück */
public User getFrom() {
   return myFrom;
}
/** Gibt den Nachrichtentext zurück */
public String getText() {
   return myText;
}
} // Ende der Klasse
```

*Listing 13.5: Die Klasse `Message`*

### 13.2.6 Nutzer-Objekt

Nachdem wir mit `ChatContext`, `ChatSwitch`, `ChatBean` und `Message` eine Infrastruktur geschaffen haben, fehlt noch das `User`-Objekt (Listing 13.8), bevor wir mit der Kodierung der `I_SMICommandListener` beginnen können.

Da Objekte der Klasse `User` in einem Store gespeichert werden sollen, muß `User` die Schnittstelle `I_Persistence` implementieren. Natürlich realisieren wir `I_Persistence` nicht neu, sondern leiten `User` von `Persistence` ab. Um ein funktionierendes persistentes Objekt zu erhalten, müssen wir die entsprechenden Konstruktoren, die `bind()`-Methode und die Zugriffsmethoden auf die Attribute kodieren.

Zuvor müssen wir jedoch festlegen, über welche Attribute `User` überhaupt verfügen soll. Aus unseren bisherigen Überlegungen ergeben sich die Attribute Name, Paßwort und Sendefarbe. Die entsprechenden Methoden heißen `getName()`, `setName(String)`, `getPassword()`, `setPassword(String)`, `getColor()` und `setColor(Integer)`. Die Farben sind in `C_Chat` (Listing 13.1) definiert.

Bei jeder Manipulation eines der persistenten Attribute wird die Methode `setModified()` (aus `Persistence`) aufgerufen, um den Zustandswechsel (Abschnitt 9.4.3.3) zu markieren.

Weiterhin wichtig für das korrekte Funktionieren der Persistenz sind die Methoden `canCreate()` und `canUpdate()`. Sie stellen in `User` sicher, daß keine zwei Nutzer den selben Namen tragen.

Über die Verwaltung seiner persistenten Attribute hinaus, kennt jedes `User`-Objekt noch weitere Methoden zum Empfangen und Schreiben von `Messages`. Da wir es mit einem Browser als Client zu tun haben, ist dieser Vorgang nicht einfach zu verwirklichen.

Um fortlaufend in ein Browser-Fenster schreiben zu können, muß die Verbindung offengehalten werden. Dies sollte jedoch nicht unendlich lange so bleiben, da sonst irgendwann der Speicher des Clients überläuft. Nachdem also eine gewissen Anzahl an Zeichen oder Nachrichten geschrieben worden ist, muß das Schreiben beendet und einen neue Verbindung aufgebaut werden. Durch dieses Vorgehen ist gewährleistet, daß der Client alte Nachrichten aus seinem Speicher entfernen kann.

Zwischen dem Schließen der alten und dem Öffnen der neuen Verbindung entsteht jedoch ein Verbindungsloch, in dem keine Nachrichten geschrieben werden können. Da wir keine Nachrichten verlieren wollen, entscheiden wir uns für einen asynchronen Schreibmechanismus.

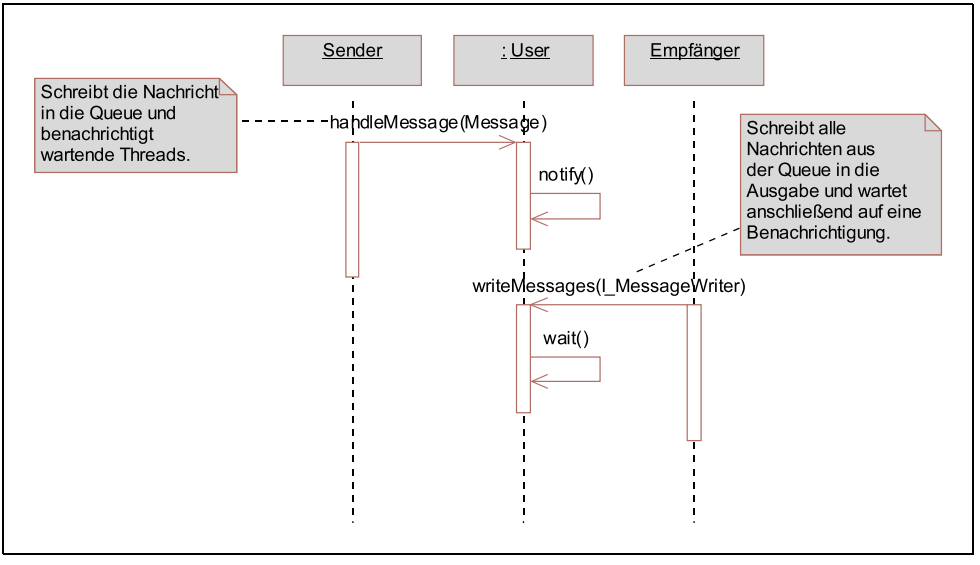


*Abbildung 13.4: Asynchrone Nachrichtenzustellung des Chats*

Nachrichten für einen Nutzer werden zunächst über `handleMessage(Message)` in die Nachrichtenschlange `myMessageQueue` geschrieben. Dann wird über `notify()` ein eventuell in der `writeMessages()`-Methode wartender Schreib-Thread benachrichtigt, der den Inhalt der Schlange in den Nachrichten-Frame des Nutzers schreibt. Ist kein Schreib-Thread vorhanden, passiert nichts. Ist einer vorhanden, der sich gerade in einem Verbindungsloch befindet, wird dem Thread erlaubt, die `writeMessages(I_MessageWriter)-Methode` zu verlassen. So wird Platz für einen neuen Schreib-Thread geschaffen.

Im Grunde fungiert das `User`-Objekt also wie ein FIFO-Puffer (first in, first out) in den ein Thread über `handleMessage()` eine Nachricht einfügt und aus dem ein anderer Thread über `writeMessages()` die jeweils vorhandenen Nachrichten wieder entfernt.

Wie bereits aus der Signatur der Methode `writeMessages(I_MessageWriter)` ersichtlich, wird zum Schreiben der Nachrichten ein `I_MessageWriter` (Listing 13.6) benutzt. Listing 13.7 zeigt eine Realisierung des Writers für HTML.

```
package de.webapp.Chat;

import java.io.IOException;

/** Definiert einen Nachrichten-Schreiber */
public interface I_MessageWriter {
/** Schreibt eine Nachricht */
public void write(Message aMessage) throws IOException;
```

```
/** Zeigt an, ob dieser I_MessageWriter noch schreiben kann */
public boolean canWrite();
} // Ende der Schnittstelle
```

*Listing 13.6: Die Schnittstelle `I_MessageWriter`*

```
package de.webapp.Chat;

import java.io.*;

public class HTMLMessageWriter implements I_MessageWriter, C_Chat {

public HTMLMessageWriter(Writer aWriter) {
   myWriter = aWriter;
}

protected int mySentBytes;
protected Writer myWriter;

/** Schreibt eine Nachricht mit dem aktuellen Writer. */
public void write(Message aMessage) throws IOException {
   StringBuffer theOutput = new StringBuffer();
   theOutput.append(aMessage.getFrom().getName());
   theOutput.append(": <font color=\""
      + C_Colors[aMessage.getFrom().getColor().intValue()]+ "\">");
   theOutput.append(aMessage.getText());
   theOutput.append("</font><p>");
   myWriter.write(theOutput.toString());
   myWriter.flush();
   mySentBytes += theOutput.length();
}
/** Überprüft, ob noch geschrieben werden kann. */
public boolean canWrite() {
   if (myWriter instanceof PrintWriter) {
      ((PrintWriter)myWriter).println();
      return (!((PrintWriter)myWriter).checkError() && mySentBytes < C_MaxBytes);
   }
   try {
      myWriter.flush();
   }
   catch (IOException ioe) {
      return false;
   }
   return true;
}
} // Ende der Klasse
```

*Listing 13.7: Die Klasse `HTMLMessageWriter`*

Als letztes müssen wir uns noch um die Beziehung des `User`-Objekts zum Chat Gedanken machen. Da der Nutzer an die `HttpSession` gebunden ist, wird er nach einer gewissen inaktiven Zeit automatisch abgemeldet. Außerdem hat er die Möglichkeit sich

selbst abzumelden. Das bedeutet, daß der Nutzer und der Chat Referenzen aufeinander besitzen müssen. Entsprechend verfügt `User` über die Methoden `setChat(ChatRoom)` und `removeChat()`, um die Assoziation setzen und entfernen zu können. Die Gegenstücke im ChatRoom heißen `register(User)` und `unregister(User)` – doch dazu später mehr (Abschnitt 13.2.9).

Bleibt zu überlegen, was passiert, wenn ein Nutzer automatisch aus der `HttpSession` entfernt wird. Da `User` die Schnittstelle `javax.servlet.http.HttpSessionBindingListener` implementiert, verfügt es über die Methoden `valueBound()` und `valueUnbound()`. Das `User`-Objekt wird also benachrichtigt, wenn es an die `HttpSession` gebunden beziehungsweise wieder von ihr entbunden wird. So ist es möglich, daß das `User`-Objekt automatisch die Methode `unregister(User)` des Chat-Raums aufruft, wenn es aus der Session entfernt wird.

Ebenso machen wir uns den Mechanismus für das Referenzzählen (Abschnitt 9.4.3) zunutze. Wird das Objekt gebunden, rufen wir `retain()` auf und erhöhen so den internen Referenzzähler. Beim Entbinden des Objekts rufen wir `release()` auf.

```
package de.webapp.Chat;

import java.util.*;
import java.io.*;
import java.sql.*;
import javax.servlet.http.*;
import de.webapp.Framework.Persistence.*;
import de.webapp.Framework.Log.*;

/** Nutzer des Chats */
public class User extends Persistence
   implements C_Chat, HttpSessionBindingListener {
/** Name des Nutzers */
protected String myName;
/** Paßwort des Nutzers */
protected String myPassword;
/** Sendefarbe des Nutzers */
protected Integer myColor;

/** Chatraum, in dem sich der Nutzer befindet */
protected transient ChatRoom myChat;
/** Nachrichtenschlange */
protected transient Vector myMessageQueue;
/** Zeitpunkt, zu dem die letzte Nachricht gesendet wurde */
protected transient long myLastMessage;
/** Zeigt an, ob das asynchrone Schreiben der Nachrichten gestoppt wurde */
protected transient boolean stopped;
/** Aktuelle benutzter I_MessageWriter */
protected transient I_MessageWriter myWriter;

/** Geschützter -Konstruktor */
```

```
protected User() {}
/** Konstruktor mit Peer */
public User(I_PersistencePeer aPeer) throws PersistenceException {
   super(aPeer);
   init();
}
/** Konstruktor mit Peer und ResultSet */
public User(I_PersistencePeer aPeer, ResultSet rs)
      throws PersistenceException {
   super(aPeer, rs);
   init();
}
/** Konstruktor mit Peer, ResultSet und ObjectIdentifier (für MS-Access) */
public User(I_PersistencePeer aPeer, ResultSet rs,
   String aObjectIdentifier)
      throws PersistenceException  {
   this(aPeer);
   this.bind(rs, aObjectIdentifier);
}
/** Initialisierung */
protected void init() {
   myMessageQueue = new Vector();
   myChat = null;
   myWriter = null;
   stopped = false;
}
/** Binden eines ResultSets */
public int bind(ResultSet rs , int offset)
      throws PersistenceException, SQLException {
   offset = super.bind(rs, offset);
   myName = (String)rs.getObject(++offset);
   myPassword = (String)rs.getObject(++offset);
   myColor = (Integer)rs.getObject(++offset);
   return offset;
}
/** Gibt den Namen zurück */
public String getName() {
   return myName;
}
/** Setzt den Namen */
public void setName(String aName) {
   myName = aName;
   // Zustandsänderung!
   setModified();
}
/** Gibt das Paßwort zurück */
public String getPassword() {
   return myPassword;
}
/** Setzt das Paßwort */
public void setPassword(String aPassword) {
```

```
   myPassword = aPassword;
   // Zustandsänderung!
   setModified() ;
}
/** Gibt die Farbe zurück */
public Integer getColor() {
   return myColor;
}
/** Setzt die Farbe */
public void setColor(Integer aColor) {
   myColor = aColor;
   setModified() ;
}
/** Prüft, ob ein Nutzer-Objekt erzeugt werden darf */
public boolean canCreate() throws PersistenceException {
   return isNameFree();
}
/** Prüft, ob ein Nutzer-Objekt persistent geändert werden darf */
public boolean canUpdate() throws PersistenceException {
   return isNameFree();
}
/** Zeigt an, ob ein Name bereits von einem anderen Objekt belegt ist */
protected boolean isNameFree() throws PersistenceException {
   I_Persistence sameName =
      getPeer().getRetriever().retrieve(getAttribute("Name"), myName);
   return (sameName == null || sameName == this);
}
/** Hilfsmethode zum Erlangen einen PersistenceAttribute-Objekts */
protected PersistenceAttribute getAttribute(String aAttributeName) {
   return getPeer().getType().getAttributeWithName(aAttributeName);
}
/** Versieht das Nutzer-Objekt mit einem Zeitstempel */
public void touch() {
   myLastMessage = System.currentTimeMillis();
}
/** Gibt die Zeit an, die seit dem letzten Aufruf
    von touch() vergangen ist */
public long getInactiveTime() {
   return (System.currentTimeMillis()-myLastMessage);
}
/** Setzt den Chatraum dieses Nutzers.
    Entspricht dem Betreten des Chats */
public void setChat(ChatRoom aChat) {
   stopped = false;
   myChat = aChat;
   touch();
}
/** Entfernt die Assoziation zum Chat. Entspricht Verlassen des Chats */
public void removeChat() {
   myChat = null;
   // asynchrones Schreiben der Nachrichten stoppen
```

```
   stopped = true;
   synchronized (this) {
      // wartenden Schreib-Thread benachrichtigen
      notify();
   }
}
/** Schreibt eine Nachricht in die Nachrichtenschlange und
    benachrichtigt einen wartenden Thread */
public void handleMessage(Message aMessage) {
   myMessageQueue.addElement(aMessage);
   synchronized (this) {
      notify();
   }
}
/** Schreibt Nachrichten mit dem übergebenen I_MessageWriter bis
    canWrite() false zurückgibt oder stopped true ist */
public synchronized void writeMessages(I_MessageWriter aWriter) {
   // wartende Threads, die nicht mehr schreiben können
   // (canWrite() == false) die Chance zum verlassen dieser Methode
   // und myWriter = null zu setzen
   notify();
   // Falls ein anderer Writer noch funktionstüchtig
   // ist, Ausnahme auslösen
   if (myWriter != null) throw new IllegalStateException();
   // Writer übernehmen
   myWriter = aWriter;
   try {
       // Schreibschleife
       while(myWriter.canWrite() && !stopped) {
         // Nachrichtenschlange leerschreiben
         while (myMessageQueue.size() > 0 && myWriter.canWrite()
             && !stopped) {
             Message theMessage = (Message)myMessageQueue.elementAt(0);
             myWriter.write(theMessage);
             myMessageQueue.removeElementAt(0);
         }
         try {
             // Mindestens alle 10 Sekunden aufwachen, um zu testen,
             // ob der Writer noch funktioniert und das asynchrone
             // Schreiben noch nicht gestoppt wurde.
             wait(10000);
         }
         catch (InterruptedException ie) {}
      }
   }
   catch (IOException ioe) {
      if (Log.isLog()) {
         Log.log(ioe);
      }
   }
   // Writer auf null setzen
```

```
   myWriter = null;
}
/** Beim Binden dieses Objekts an die HttpSession den Referenzzähler erhöhen */
public void valueBound(HttpSessionBindingEvent e) {
   if (Log.isLog(C_Log.METHOD)) {
      Log.log("User " + getName()
         + " wurde an die HttpSession gebunden.", C_Log.METHOD);
   }
   retain();
}
/** Beim Entbinden dieses Objekts von der HttpSession den Referenzzähler erniedrigen,
beim Chat abmelden und die Chatassoziation löschen */
public void valueUnbound(HttpSessionBindingEvent e) {
   if (Log.isLog(C_Log.METHOD)) {
      Log.log("User " + getName()
         + " wurde von der HttpSession entbunden.", C_Log.METHOD);
   }
   release();
   if (myChat != null) myChat.unregister(this);
   removeChat();
}
} // Ende der Klasse
```

*Listing 13.8: Die Klasse `User`*

### 13.2.7 ChatDispatcher

Als ersten `I_SMICommandListener` kümmern wir uns um den `ChatDispatcher` (Listing 13.9). Er soll nur ein einziges Kommando abbilden, nämlich `sendMessage`. Wie auch die anderen `I_SMICommandListener` erbt `ChatDispatcher` von `ChatBean`. Das Kommando `sendMessage` wird durch die Methode `send(SMICommand)` verwirklicht.

Zunächst wird überprüft, ob der Nutzer, der eine Nachricht verschicken möchte, überhaupt angemeldet ist. Ist das nicht der Fall, wird der Wert von `C_Template` auf den Wert von `"notLoggedIn"` umgesetzt und die Methode kehrt zurück.

Ist der Nutzer eingeloggt, wird anschließend seine `touch()`-Methode aufgerufen, um zu kennzeichnen, daß er versucht hat, eine Nachricht zu senden. Dann werden die Parameter `C_To`, `C_Message`, `C_All` und `C_Color` aus dem `SMICommand` gelesen. Für den Fall, daß sich die Sendefarbe des Nutzers geändert hat, wird die Veränderung persistent gespeichert.

Mit den gewonnenen Daten wird eine `Message` instantiiert. Zum Ausliefern werden noch die `handleMessage()`-Methoden derjenigen Nutzer aufgerufen, an die die Nachricht adressiert ist.

```
package de.webapp.Chat;

import java.util.*;
import java.io.*;
```

```
import de.webapp.Framework.SMI.*;
import de.webapp.Framework.Persistence.*;
import de.webapp.Framework.Log.*;

/** Übernimmt das Versenden von Nachrichten */
public class ChatDispatcher extends ChatBean implements C_Chat {

/** Sendet eine Nachricht */
public void send(SMICommand aSMICommand) throws PersistenceException {
   // Nutzer überhaupt noch eingeloggt?
   User theUser = getUser(aSMICommand);
   if (theUser == null) {
      aSMICommand.putValue(C_Template,
         aSMICommand.getValue("userNotLoggedIn"));
      return;
   }
   // Zeitstempel setzen
   theUser.touch();
   // Parameter aus der Anfrage lesen
   // theTo enthält die ObjectIdentifier der Adressaten
   String[] theTo = aSMICommand.getRequest().getParameterValues(C_To);
   // theText enthält die Nachricht
   String theText = (String)aSMICommand.getValue(C_Message);
   // wenn theToAll != null ist, ist die Nachricht an alle
   String theToAll = (String)aSMICommand.getValue(C_All);
   // theColor enthält die Farbe als Integer
   Integer theColor = new Integer((String)aSMICommand.getValue(C_Color));
   if (!theColor.equals(theUser.getColor())) {
      // gegebenenfalls Farbe persistent speichern
      updateColor(aSMICommand);
   }
   // Nachricht nur versenden, wenn es auch Adressaten gibt
   if (theTo != null || theToAll != null) {
      Message theMessage = new Message(theUser, theText);
      if (theToAll==null) {
         // Private Messages einzeln zustellen
         // eine Kopie an den Absender soll in jedem Fall dabei sein
         theMessage.getFrom().handleMessage(theMessage);
         for (int i=0; i < theTo.length; i++) {
            User aUser = (User)getUserlist().get(theTo[i]);
            if (aUser != null && aUser != theMessage.getFrom())
               aUser.handleMessage(theMessage);
         }
      }
      else {
         // allgemeine Methode für alle
         for (Enumeration e = getUserlist().elements();
            e.hasMoreElements();) {
            User aUser = (User)e.nextElement();
            if (aUser != null) aUser.handleMessage(theMessage);
         }
```

```
      }
   }
}
/** Speichert das NutzerObjekt mit neuer Farbe */
protected void updateColor(SMICommand aSMICommand)
      throws PersistenceException {
   // Store besorgen
   I_Store theStore = getStore();
   User theUser = getUser(aSMICommand);
   synchronized(theStore) {
      theUser.setColor(
         new Integer((String)aSMICommand.getValue(C_Color)));
      try {
         theStore.begin();
         theUser.update();
         theStore.commit();
      }
      catch (Throwable t) {
         theStore.rollback();
         if (t instanceof PersistenceException) {
            throw (PersistenceException)t;
         }
         throw new SMIEventListenerException(t);
      }
   }
}
} // Ende der Klasse
```

*Listing 13.9: Die Klasse `ChatDispatcher`*

### 13.2.8 ChatRegistration

Die Kommandos zur Manipulation eines Nutzer-Objekts `createUser` und `updateUser` sollen von der Klasse `ChatRegistration` umgesetzt werden. Die entsprechenden Methoden heißen `create()` und `update()`.

In `create()` wird zunächst über `newPersistence()` ein leeres `User`-Objekt vom Store erlangt, das dann über die Hilfsmethode `setAttributes()` mit Attributwerten versorgt wird. Geht alles gut, wird das Objekt über `create()` in der Datenbank und über `setUser()` in der `HttpSession` hinterlegt. Bei eventuellen Fehlern wird als `C_Template` erneut die Eingabemaske gesetzt und eine Fehlermeldung unter `C_ErrorMessage` hinterlegt. Genauso funktioniert auch `update()`. Hier wird jedoch das Objekt nicht aus dem Store geholt, sondern aus der Session gelesen. Das Hinterlegen des Objekts in der Session erübrigt sich also.

```
package de.webapp.Chat;

import java.util.*;
import java.io.*;
```

```
import de.webapp.Framework.SMI.*;
import de.webapp.Framework.Persistence.*;
import de.webapp.Framework.Log.*;

/** Manipulation eines Nutzer-Objekts */
public class ChatRegistration extends ChatBean implements C_Chat {
/** Erzeugt ein Nutzer-Objekt und hinterlegt es in der HttpSession */
public void create(SMICommand aSMICommand) throws PersistenceException {
   // Store besorgen.
   I_Store theStore = getStore();
   User theUser;
   // Store nach neuem Persistence des Typs "User" fragen
   theUser = (User)theStore.newPersistence("User");
   // Attribute setzen und eventuell Fehlermeldung ausgeben
   if (!setAttributes(aSMICommand, theUser)) {
      aSMICommand.putValue(C_ErrorMessage,
        "Ihre Eingaben waren fehlerhaft. "
        + "Bitte versuchen Sie es nocheinmal.");
      aSMICommand.putValue(C_Template, aSMICommand.getValue("failure"));
      return;
   }
   // Store sichern
   synchronized(theStore) {
      try {
         theStore.begin();
         theUser.create();
         theStore.commit();
         // User in der HttpSession hinterlegen -> ChatBean
         setUser(aSMICommand, theUser);
      }
      // wenn canCreate() false zurückgibt, wird eine
      // PersistenceRuntimeException ausgelöst.
      // => Fehlermeldung ausgeben.
      catch (PersistenceRuntimeException pre) {
         theStore.rollback();
         aSMICommand.putValue(C_ErrorMessage,
             "Der Name, den Sie gewählt haben, ist belegt.");
         aSMICommand.putValue(C_Template,
             aSMICommand.getValue("failure"));
      }
      catch (Throwable t) {
         theStore.rollback();
         if (t instanceof PersistenceException) {
             throw (PersistenceException)t;
         }
         throw new SMIEventListenerException(t);
      }
   }
}
/** Speichert Änderungen eines Nutzer-Objekts in der Datenbank */
public void update(SMICommand aSMICommand) throws PersistenceException {
```

```
   // Nutzer aus der HttpSession lesen
   User theUser = getUser(aSMICommand);
   // falls kein Nutzer vorhanden, Fehlermeldung ausgeben
   if (theUser == null) {
      aSMICommand.putValue(C_Template,
         aSMICommand.getValue("userNotLoggedIn"));
      return;
   }
   // Store besorgen.
   I_Store theStore = getStore();
   // Attribute setzen und eventuell Fehlermeldung ausgeben
   if (!setAttributes(aSMICommand, theUser)) {
      aSMICommand.putValue(C_ErrorMessage,
        "Ihre Eingaben waren fehlerhaft. "
        + "Bitte versuchen Sie es noch einmal.");
      aSMICommand.putValue(C_Template, aSMICommand.getValue("failure"));
      return;
   }
   // Store sichern
   synchronized(theStore) {
      try {
         theStore.begin();
         theUser.update();
         theStore.commit();
      }
      // wenn canUpdate() false zurückgibt, wird eine
      // PersistenceRuntimeException ausgelöst.
      // => Fehlermeldung ausgeben.
      catch (PersistenceRuntimeException pre) {
         theStore.rollback();
         aSMICommand.putValue(C_ErrorMessage,
             "Der Name, den Sie gewählt haben, ist belegt.");
         aSMICommand.putValue(C_Template,
             aSMICommand.getValue("failure"));
      }
      catch (Throwable t) {
         theStore.rollback();
         if (t instanceof PersistenceException) {
             throw (PersistenceException)t;
         }
         throw new SMIEventListenerException(t);
      }
   }
}
/** Setzt die Attribute eines Persistence und gibt false zurück,
    falls die Werte nicht sinnvoll sind */
protected boolean setAttributes(SMICommand aSMICommand, User aUser) {
   // Werte aus dem Kommando lesen.
   String thePassword1 = (String)aSMICommand.getValue(C_Password);
   String thePassword2 = (String)aSMICommand.getValue(C_Password2);
   String theUsername = (String)aSMICommand.getValue(C_Username);
```

```
    Integer theColor = new Integer((String)aSMICommand.getValue(C_Color));
    // Sinnhaftigkeit überprüfen
    if (thePassword1 == null
        || theUsername == null
        || !thePassword1.equals(thePassword2)) return false;
    if (thePassword1.length() == 0
        || theUsername.length() == 0) return false;
    // Werte setzen
    aUser.setPassword(thePassword1);
    aUser.setName(theUsername);
    aUser.setColor(theColor);
    return true;
}
} // Ende der Klasse
```

*Listing 13.10: Die Klasse `ChatRegistration`*

### 13.2.9 ChatRoom

Mit dem `ChatRoom` (Listing 13.11) kommen wir zur letzten zu beschreibenden Klasse. Sie kapselt die komplette Verwaltung des Raums; somit ist sie verantwortlich für Authentifizierung sowie das Betreten und Verlassen des Raums.

Beginnen wollen wir mit der Authentifizierung. Sie erfolgt auf zweierlei Arten. Zum einen, wenn der Nutzer sich registriert, diesen Fall haben wir bereits mit der `ChatRegistration`-Klasse erläutert; zum anderen, wenn der Nutzer bereits registriert ist, im Login-Dialog Name und Paßwort eingibt und anschließend das `login`-Kommando sendet. Das Kommando führt zum Aufruf der Methode `login(SMICommand)`. In `ihr` wird als erstes überprüft, ob der Nutzer sich bereits authentifiziert hat. Ist dies der Fall, kehrt die Methode sofort zurück. Hat der Nutzer sich noch nicht authentifiziert, werden die Parameter `C_Username` und `C_Password` aus dem Kommando gelesen. Anschließend wird der Store über die Hilfsmethode `getUserForName(String)` nach einem passenden `User`-Objekt für den angegebenen Namen gefragt. Ist kein `User` mit dem Namen im Store registriert oder stimmt sein Paßwort nicht mit dem eingegebenen überein, wird eine Fehlermeldung unter `C_ErrorMessage` hinterlegt und als `C_Template` der Wert von `"errorOnLogin"` gesetzt. Tritt kein Fehler auf, wird das erlangte Nutzer-Objekt über `setUser()` in der `HttpSession` hinterlegt. Der Nutzer ist nun authentifiziert.

Möchte er nun den Chat-Raum betreten, sendet er das Kommando `enterRoom`. Das Kommando wird auf die Methode `enter(SMICommand)` abgebildet. In ihr wird zunächst überprüft, ob ein Nutzer authentifiziert ist. Ist dies nicht der Fall, wird als `C_Template` der Wert von `"userNotLoggedIn"` gesetzt und die Methode per `return` verlassen. Existiert jedoch ein authentifizierter Nutzer, wird die `register(User)`-Methode aufgerufen. Sie fügt das `User`-Objekt der Nutzerliste des Chats hinzu und assoziiert den Nutzer mittels `setChat(ChatRoom)` mit dem Chat. Als Schlüssel für die Nutzerliste wird dabei übrigens der `ObjectIdentifier` des `User`-Objekts benutzt.

Wenn der Nutzer sich im Raum befindet, kann er im rechten Frame Nachrichten lesen. Hierfür zeichnet sich das Kommando `showMessages` verantwortlich. Es sorgt für den Aufruf der gleichnamigen Methode `showMessages(SMICommand)`. In ihr wird überprüft, ob der Nutzer länger als die erlaubte Zeit passiv mitgelesen hat. Ist dies der Fall, wird er über `logout(SMICommand)` ausgeloggt. Er ist anschließend gegenüber dem System nicht mehr authentifiziert und nicht mehr im Raum registriert.

Natürlich kann man den Raum auch auf normalem Weg verlassen. Dazu dient das Komando `leaveRoom`. Es führt zu einem Aufruf der Methode l`eave(SMICommand)`. In ihr wird die Methode `removeChat()` des Users sowie die Methode `unregister(User)` aufgerufen. `unregister(User)` entfernt den Nutzer aus der Nutzerliste.

Bleibt als letzte Methode `logout(SMICommand)` zu erklären. Wie nicht anders zu erwarten, wird sie von dem Kommando `logout` aufgerufen. Sie ruft zunächst die Methode `leave(SMICommand)` auf und entfernt anschließend über `removeUser(SMICommand)` das `User`-Objekt aus der HttpSession.

```
package de.webapp.Chat;

import java.util.*;
import java.io.*;

import de.webapp.Framework.SMI.*;
import de.webapp.Framework.Persistence.*;
import de.webapp.Framework.Log.*;

/** Verwaltet den ChatRoom */
public class ChatRoom extends ChatBean implements C_Chat {

/** Authentifiziert den Nutzer anhand seines Namens und Paßworts */
public void login(SMICommand aSMICommand) throws PersistenceException {
   User theUser = getUser(aSMICommand);
   // Testen, ob sich der Nutzer bereits authentifiziert hat.
   if (theUser != null) {
      if (Log.isLog(C_Log.METHOD,
         getSMIEventSwitch().getSMIContext().getName())) {
         Log.log("User " + theUser.getName()
            + " ist bereits authentifiziert.", C_Log.METHOD,
            getSMIEventSwitch().getSMIContext().getName());
      }
      return;
   }
   // Name und Paßwort aus dem SMICommand lesen
   String theName = (String)aSMICommand.getValue(C_Username);
   String thePassword = (String)aSMICommand.getValue(C_Password);
   // User aus dem Store holen
   theUser = getUserForName(theName);
   // Testen, ob ein passendes User-Objekt gefunden wurde
   if (theUser == null) {
```

```
      if (Log.isLog(C_Log.METHOD,
         getSMIEventSwitch().getSMIContext().getName())) {
         Log.log("User " + theName + " existiert nicht.",
            C_Log.METHOD, getSMIEventSwitch().getSMIContext().getName());
      }
      // Fehlermeldung setzen
      aSMICommand.putValue(C_ErrorMessage,
         "Es ist kein Nutzer mit dem Namen "
         + theName + " registriert. Bitte versuchen Sie es noch einmal.");
      // Template setzen
      aSMICommand.putValue(C_Template,
          aSMICommand.getValue("errorOnLogin"));
      return;
   }
   // Passwort des Nutzers überprüfen.
   if (!(theUser.getPassword()).equals(thePassword)) {
      if (Log.isLog(C_Log.METHOD,
         getSMIEventSwitch().getSMIContext().getName())) {
         Log.log("User " + theName
           + " hat das falsche Paßwort eingegeben.", C_Log.METHOD,
           getSMIEventSwitch().getSMIContext().getName());
      }
      // Fehlermeldung setzen
      aSMICommand.putValue(C_ErrorMessage,
         "Falsches Paßwort. Bitte versuchen Sie es noch einmal.");
      // Template setzen
      aSMICommand.putValue(C_Template,
         aSMICommand.getValue("errorOnLogin"));
      return;
   }
   // Der Nutzer existiert und hat das richtige Paßwort angegeben.
   // Also: User-Objekt in der Session hinterlegen.
   // Der User ist somit generell authentifiziert
   setUser(aSMICommand, theUser);
}
/** Ruft register() auf, falls der Nutzer sich authentifiziert hat */
public void enter(SMICommand aSMICommand) throws PersistenceException {
   User theUser = getUser(aSMICommand);
   if (theUser != null) {
      register(theUser);
      if (Log.isLog(C_Log.MODULE,
         getSMIEventSwitch().getSMIContext().getName())) {
         Log.log("User " + theUser.getName()
            + " hat den Raum betreten.", C_Log.MODULE,
            getSMIEventSwitch().getSMIContext().getName());
      }
   }
   else {
      aSMICommand.putValue(C_Template,
         aSMICommand.getValue("userNotLoggedIn"));
   }
```

```
}
/** Wird zum Verlassen des Raumes aufgerufen */
public void leave(SMICommand aSMICommand) {
   User theUser = getUser(aSMICommand);
   if (theUser != null) {
      // Assoziation des Users mit
      // diesem Chat löschen
      theUser.removeChat();
      // Nutzer aus der Nutzerliste streichen
      unregister(theUser);
   }
}
/** Löscht die Authentifizierung des Nutzers */
public void logout(SMICommand aSMICommand) {
   leave(aSMICommand);
   removeUser(aSMICommand);
}
/** Nutzer in die Nutzerliste eintragen */
public void register(User aUser) {
   // Sicherheitshalber testen, ob der Nutzer nicht bereits
   // am Chat teilnimmt.
   if (!getUserlist().containsKey(aUser.getObjectIdentifier())) {
      // Dem Nutzer seinen Chatraum mitteilen.
      aUser.setChat(this);
      getUserlist().put(aUser.getObjectIdentifier(), aUser);
   }
}
/** Nutzer aus dem Raum entfernen */
public void unregister(User aUser) {
   Hashtable theUserlist = getUserlist();
   if (theUserlist != null)
      theUserlist.remove(aUser.getObjectIdentifier());
   if (Log.isLog(C_Log.MODULE,
      getSMIEventSwitch().getSMIContext().getName())) {
      Log.log("User " + aUser.getName()
         + " hat den Raum verlassen.", C_Log.MODULE,
         getSMIEventSwitch().getSMIContext().getName());
   }
   aUser = null;
}
/** Gibt die Meldungen aus */
public void showMessages(SMICommand aSMICommand) throws IOException {
   User theUser = getUser(aSMICommand);
   // Überprüfen, ob der Nutzer länger als die erlaubte
   // Zeit passiv war. Wenn ja, ausloggen!
   if (theUser != null)
      if (theUser.getInactiveTime() > C_MaxInactiveTime)
        logout(aSMICommand);
   // Testen, ob der Nutzer noch eingeloggt ist
   if (getUser(aSMICommand) == null) {
      aSMICommand.putValue(C_Template,
```

```
            aSMICommand.getValue("userNotLoggedIn"));
      }
   }
   /** Sucht einen Nutzer mit einem bestimmten Namen aus dem Store */
   protected User getUserForName(String aName) throws PersistenceException {
      User theUser = null;
      // Store besorgen.
      I_Store theStore = getStore();
      // Nutzer aus dem Store lesen.
      synchronized(theStore) {
         try {
            theStore.begin() ;
            theUser = (User)theStore.persistenceWithKey(aName,
               "User", "Name");
            theStore.commit() ;
         }
         catch (Throwable t) {
            theStore.rollback();
            if (t instanceof PersistenceException) {
               throw (PersistenceException)t;
            }
            throw new SMIEventListenerException(t);
         }
      }
      return theUser;
   }
   } // Ende der Klasse
```

*Listing 13.11: Die Klasse* `ChatRoom`

## 13.3 Anzeige

Nachdem wir uns ausführlich um die Anwendungslogik gekümmert haben, kommen wir nun zur Anzeige beziehungsweise Ansteuerung des Chats.

### 13.3.1 Code-Fragmente importieren

Genau wie in den anderen Beispielen nutzen wir auch im Chat den Import-Mechanismus der GNUJSP, um Fragmente einzubinden. Die zu importierenden Fragmente sind wiederum `header.html` (Listing 13.12), `footer.html` (Listing 13.13) und `import.import` (Listing 13.14).

```
<HTML>
<HEAD>
   <TITLE>Chat</TITLE>
</HEAD>
<BODY text="#800000" link="#000080" vlink="#000080" alink="#FF0000">
```

*Listing 13.12: Code-Fragment* `header.html`

```
</BODY>
</HTML>
```

*Listing 13.13: Code-Fragment* `footer.html`

```
<%@ page import ="de.webapp.Chat.*"%>
<%@ page import ="de.webapp.Framework.SMI.*"%>
<%@ page import ="java.util.*"%>
```

*Listing 13.14: Code-Fragment* `import.import`

### 13.3.2 Login-Dialog

Der Login-Dialog (Abbildung 13.5) fordert den Nutzer auf, seinen Namen und sein Paßwort einzugeben oder sich registrieren zu lassen wofür eigentlich keine JavaServer-Page nötig wäre. Doch die Seite kann mehr. Standardmäßig wird unten auf der Seite eine Liste mit den aktuell im Raum befindlichen Nutzern angezeigt. In Klammern hinter dem Namen steht dabei die Zeit, die der jeweilige Nutzer passiv war – also nur gelesen hat. Außerdem ist die Seite in der Lage, nach einem mißglückten Loginversuch eine Fehlermeldung sowie den beim letzten Versuch benutzten Login-Namen anzuzeigen.

Um all diese Eigenschaften umzusetzen, wird zunächst das `SMICommand` aus der Anfrage gelesen (Listing 13.15). Anschließend werden die Nutzerliste und eine mögliche Fehlermeldung vom Kommando erfragt. Die Fehlermeldung wird unmittelbar ausgegeben, die Nutzerliste erst am Ende der Seite.

Als Namen der Parameter Name und Paßwort werden die Konstanten `C_Username` und `C_Password` aus `C_Chat` benutzt.

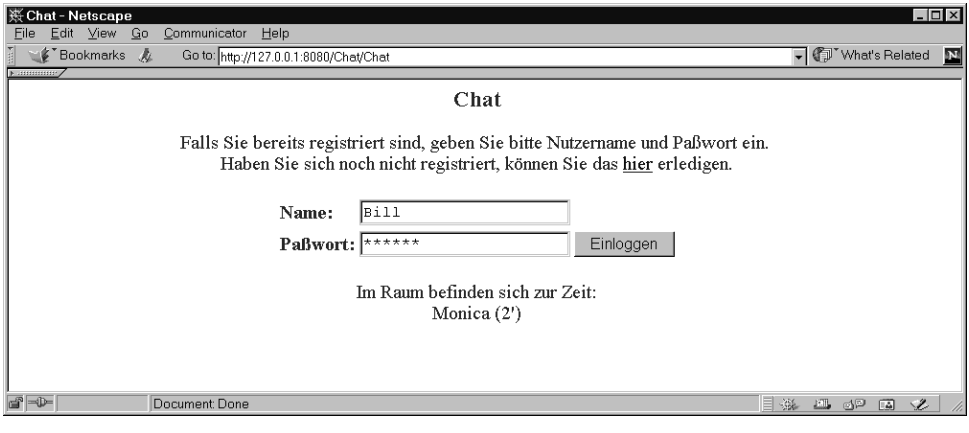


*Abbildung 13.5: Der Login-Dialog*

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>

<%
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent");
Hashtable theUserList =
   (Hashtable)theSMICommand.getValue(C_Chat.C_Userlist);
String theErrorMessage =
   (String)theSMICommand.getValue(C_Chat.C_ErrorMessage);
%>
<center>
<h3>Chat</h3>
<form action="Chat" method="POST">
<input type="Hidden" name="Command" value="login">
<%
if (theErrorMessage != null) out.println(theErrorMessage);
else {
%>
Falls Sie bereits registriert sind, geben Sie bitte Nutzername
und Paßwort ein.
<%
}
%>
<br>
Haben Sie sich noch nicht registriert, können Sie das
<a href="Chat?Command=showRegistration">hier</a> erledigen.<p>
<table border=0>
<tr>
<td><b>Name:</b></td>
<td colspan=2><input type="Text" name="<%=C_Chat.C_Username%>" size="20"
value="<%=(String)theSMICommand.getValue(C_Chat.C_Username)%>"></td>
</tr>
<tr>
<td><b>Paßwort:</b></td>
<td><input type="Password" name="<%=C_Chat.C_Password%>" size="20"></td>
<td><input type="Submit" value="Einloggen"></td>
</tr>
</table>
</form>
<p>
Im Raum befinden sich zur Zeit:<br>
<%
for (Enumeration e = theUserList.elements();e.hasMoreElements();) {
   User aUser = (User)(e.nextElement());
   long min = aUser.getInactiveTime() / (1000 * 60);
   String idle = String.valueOf(min);
   out.println(aUser.getName() + " (" + idle + "')<br>");
}
```

```
%>
</center>
<!--#include file="footer.html"-->
```

*Listing 13.15: Quellcode zum Login-Dialog: `login.jsp`*

### 13.3.3 Nutzerdaten-Dialog

Im Chat gibt es zwei Dialoge zum expliziten Manipulieren der Nutzerdaten: die Registrierung und der Nutzerdaten-Ändern-Dialog. Beide sind so ähnlich, daß wir sie in der JavaServer-Page `register.jsp` (Abbildung 13.6 und Listing 13.16) zusammenfassen. Abhängig davon, ob das Kommando ein Nutzer-Objekt enthält, wird ein Einfügen- oder ein Ändern-Dialog angezeigt. Im Ändern-Fall werden zudem die aktuellen Daten als Standardwerte gesetzt.

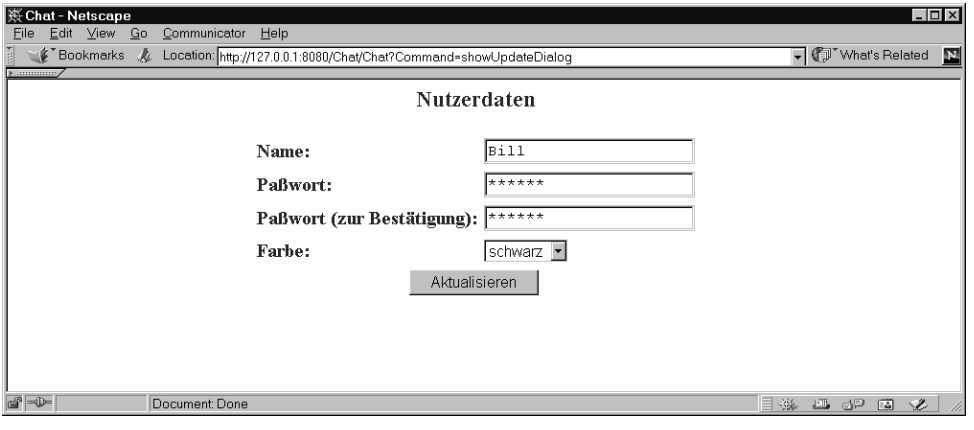


*Abbildung 13.6: Der Dialog zum Manipulieren der Nutzerdaten*

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>
<%
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent");
User theUser = (User)theSMICommand.getSessionValue(C_Chat.C_User);
String theButtonLabel = "Anlegen";
String theCommand = "createUser";
String theName = (String)theSMICommand.getValue(C_Chat.C_Username);
String thePassword = (String)theSMICommand.getValue(C_Chat.C_Password);
Integer theColor = null;
try {
   theColor = new Integer((String)theSMICommand.getValue(C_Chat.C_Color));
}
catch (NumberFormatException nfe) {}
if (theUser != null) {
   theButtonLabel = "Aktualisieren";
   theCommand = "updateUser";
```

```
   theName = theUser.getName();
   thePassword = theUser.getPassword();
   theColor = theUser.getColor();
}
if (theColor == null) theColor = new Integer(0);
%>
<center>
<h3>Nutzerdaten</h3>
<%=(String)theSMICommand.getValue(C_Chat.C_ErrorMessage)%>
<form action="Chat" method="POST">
<input type="Hidden" name="Command" value="<%=theCommand%>">
<table>
<tr>
<td><b>Name:</b></td>
<td><input type="Text" name="_Username" size="20" value="<%=theName%>"></td>
</tr>
<tr>
<td><b>Paßwort:</b></td>
<td><input type="Password" name="_Password" size="20" value="<%=thePassword%>"></td>
</tr>
<tr>
<td><b>Paßwort (zur Bestätigung):</b></td>
<td><input type="Password" name="_Password2" size="20"></td>
</tr>
<tr>
<td><b>Farbe:</b></td>
<td>
<select name="_Color">
<%
for (int i=0; i<C_Chat.C_Colors.length; i++) {
   if (i == theColor.intValue()) {
      out.println("<option selected value=\"" + i + "\">"
         + C_Chat.C_ColorNames[i]);
   }
   else {
      out.println("<option value=\"" + i + "\">"
         + C_Chat.C_ColorNames[i]);
   }
}
%>
</select>
</td>
</tr>
</table>
<input type="Submit" value="<%=theButtonLabel%>">
</form>
</center>
<!--#include file="footer.html"-->
```

*Listing 13.16: Quellcode für die Dialoge zum Manipulieren der Nutzerdaten:* `register.jsp`

### 13.3.4 Menü-Dialog

Nach dem Anmelden beziehungsweise erfolgreichem Registrieren gelangt der Nutzer zum Menü-Dialog (Abbildung 13.7 und Listing 13.17). Hier werden dem Nutzer drei Möglichkeiten präsentiert. Er kann den Raum betreten, die Nutzerdaten ändern oder sich abmelden. Alle drei Möglichkeiten sind gewöhnliche Verweise. Nur die Liste der Nutzer am Ende der Seite erfordert den Einsatz von JSP. Die Kodierung ist dabei identisch zum Login-Dialog (13.3.2).

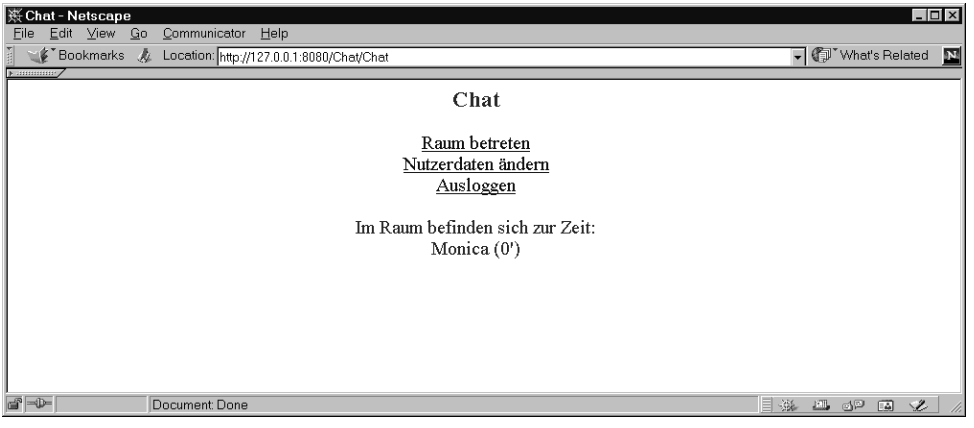


*Abbildung 13.7: Der Menü-Dialog*

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>
<%
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent");
Hashtable theUserList =
   (Hashtable)theSMICommand.getValue(C_Chat.C_Userlist);
%>
<center>
<h3>Chat</h3>
<a href="Chat?Command=enterRoom">Raum betreten</a><br>
<a href="Chat?Command=showUpdateDialog">Nutzerdaten ändern</a><br>
<a href="Chat?Command=logout">Abmelden</a><br>
<p>
Im Raum befinden sich zur Zeit:<br>
<%
for (Enumeration e = theUserList.elements();e.hasMoreElements();) {
   User aUser = (User)(e.nextElement());
   long min = aUser.getInactiveTime() / (1000 * 60);
   String idle = String.valueOf(min);
   out.println(aUser.getName() + " (" + idle + "')<br>");
}
```

```
%>
</center>
<!--#include file="footer.html"-->
```

*Listing 13.17: Quellcode zum Menü-Dialog:* `menu.jsp`

### 13.3.5 Frameset

Beim Betreten des Raums wird zunächst ein Frameset aufgebaut. Die Kodierung ist denkbar einfach – es handelt sich um reines HTML (Listing 13.18).

```
<html>
<head>
<title>Chat</title>
</head>
<frameset cols="50%, *">
    <frame src="Chat?Command=showSendDialog" marginwidth="10" marginheight="10"
scrolling="auto" frameborder="no">
    <frame src="Chat?Command=showMessages" marginwidth="10" marginheight="10"
scrolling="auto" frameborder="no">
</frameset>
</html>
```

*Listing 13.18: HTML-Code für das Frameset* `frame.html`

### 13.3.6 Sende-Dialog

Der Sende-Dialog zeigt eine Maske zur Eingabe von Nachrichten (Abbildung 13.8 und Listing 13.19). Der Nutzer kann dabei festlegen, in welcher Farbe die Nachricht erscheinen und an wen sie geschickt werden soll. Da die Sendefarbe ein Attribut des Nutzers ist, muß zunächst einmal das `request`-Objekt nach dem Kommando und das wiederum nach dem Nutzer gefragt werden. Mit einer Schleife über die möglichen Farben werden dann die `<option>`-Auszeichnungen ausgegeben. Dabei wird die Farbe des Nutzers mit dem Attribut `selected` versehen.

Nach der Ausgabe der Farben wird die Liste der Nutzer mit vorangestellten Checkboxen dargestellt. Die Checkboxen der Nutzer, an die die letzte Nachricht geschickt wurde, sind dabei mit `checked` markiert. Als Wert einer Checkbox wird aus Gründen der Eindeutigkeit der `ObjectIdentifier` eines Nutzers und nicht sein Name benutzt.

Am Ende der Seite geben wir noch eine Verweis zum Ausloggen aus.

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>

<form action="Chat" method="POST">
<input type=hidden name="Command" value="sendMessage">
<textarea name="_Message" cols=20 rows=5 wrap=soft></textarea>
<p>
<input type=submit value="Senden">
<p>
```

```
<select name="_Color">
<%
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent");
Hashtable theUserList =
   (Hashtable)theSMICommand.getValue(C_Chat.C_Userlist);
User theUser = (User)theSMICommand.getSessionValue(C_Chat.C_User);
for (int i=0; i<C_Chat.C_Colors.length; i++) {
   if (i == theUser.getColor().intValue()) {
      out.println("<option selected value=\"" + i + "\">"
         + C_Chat.C_ColorNames[i]);
   }
   else {
      out.println("<option value=\"" + i + "\">"
         + C_Chat.C_ColorNames[i]);
   }
}
%>
</select>
<p>
<%
String[] theTo = request.getParameterValues(C_Chat.C_To);
String[] theAll = request.getParameterValues(C_Chat.C_All);
if (theTo == null || theAll != null)
   out.println("<input type=checkbox name=\"_All\" checked>An alle<br>");
else
   out.println("<input type=checkbox name=\"_All\">An alle<br>");
for (Enumeration e = theUserList.elements();e.hasMoreElements();) {
   User aUser = (User)(e.nextElement());
   String checked = new String("");
   if (theTo != null) {
      for (int i=0; (i < theTo.length)
         && (!(checked.equals("checked"))); i++){
         if (aUser.getObjectIdentifier().equals(theTo[i])) {
            checked = "checked";
         }
      }
   }
   long min = aUser.getInactiveTime() / (1000 * 60);
   String idle = String.valueOf(min);
   out.println("<input type=checkbox name=\"_To\" value=\""
      + aUser.getObjectIdentifier() + "\" " + checked + ">"
      + aUser.getName() + " (" + idle + "')<br>");
}
%>
</form>
<p>
<a href="Chat?Command=leaveRoom" target="_top">Ausgang</a>
<!--#include file="footer.html"-->
```

*Listing 13.19: Quellcode zum Sende-Dialog:* `sendform.jsp`

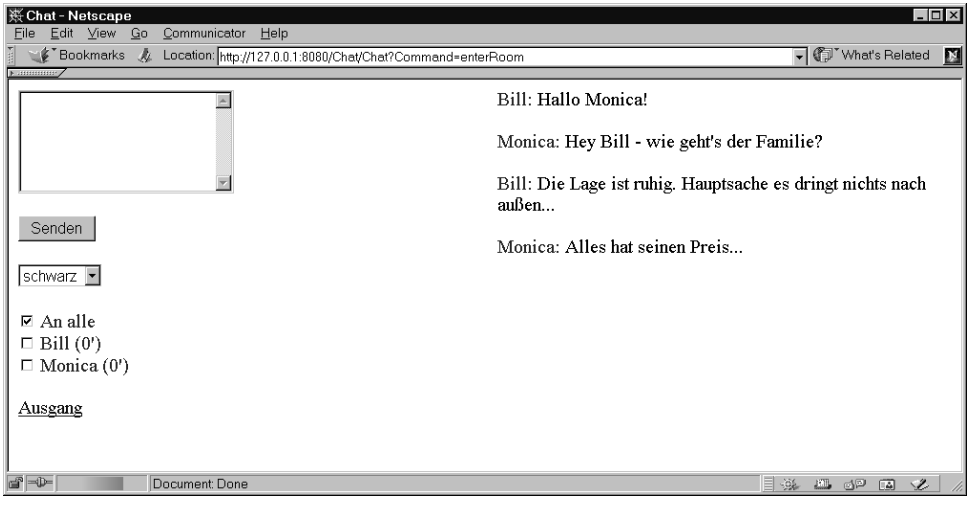


*Abbildung 13.8: Der Sende-Dialog*

### 13.3.7 Nachrichten-Ausgabe

Um die Nachrichten-Ausgabe wie gefordert zu bewältigen, müssen wir ein wenig tricksen. Im Gegensatz zu den anderen Seiten kommt hier nicht der normale HTML-Kopf `header.html` zum Zug, sondern die Datei `outputheader.html` (Listing 13.20). Der Unterschied besteht in dem JavaScript mit der Funktion `refreshWindow()`. Der Aufruf dieser Funktion bewirkt ein erneutes Laden der Seite.

```
<HTML>
<HEAD>
   <TITLE>Chat</TITLE>
<SCRIPT LANGUAGE="JavaScript">
<!--
function refreshWindow() {
   location = "Chat?Command=showMessages";
}
//-->
</SCRIPT>
</HEAD>
<BODY text="#800000" link="#000080" vlink="#000080" alink="#FF0000">
```

*Listing 13.20: Code-Fragment `outputheader.html`*

Das Code-Fragment `outputheader.html` wird anstelle von `header.html` in die Datei `messages.jsp` (Listing 13.21) eingebunden. Nach dem Aufruf der Seite über das Kommando `showMessages` wird zunächst einmal das Nutzer-Objekt aus dem Kommando gelesen. Anschließend wird die Methode `writeMessages(I_MessageWriter)` des Nutzers aufgerufen. Wie wir uns erinnern, ist dies die Methode, die Nachrichten asynchron mittels eines `I_MessageWriters` schreibt. Da wir HTML schreiben müssen, übergeben wir also

als I_MessageWriter einen HTMLMessageWriter, der mit dem PrintWriter der JavaServer-Page initialisiert wurde. Der Aufruf von writeMessages() kehrt nicht zurück, bevor unser HTMLMessageWriter nicht mehr schreiben kann – canWrite() also false zurückgibt. Nachdem er zurückkehrt, wird drei Sekunden gewartet, bevor die JavaScript-Funktion refreshWindow() aufgerufen wird. Dadurch können wir die Datei komplett an den Browser liefern, so daß dieser keine Fehlermeldung wegen einer unterbrochenen Übertragung ausgeben muß.

Für den Fall, daß der Aufruf von writeMessage() wegen einer IllegalStateException mißlingt, geben wir eine Fehlermeldung aus.

```
<%@ include file="header.html" %>
<%@ include file="import.import" %>
<%
SMICommand theSMICommand = (SMICommand)request.getAttribute("SMIEvent");
User theUser = (User)theSMICommand.getSessionValue(C_Chat.C_User);
try {
   theUser.writeMessages(new HTMLMessageWriter(out));
%>
<SCRIPT LANGUAGE="JavaScript">
<!--
setTimeout("refreshWindow()",3000);
//-->
</SCRIPT>
<%
}
catch (IllegalStateException ise) {
%>
Sie sind bereits eingeloggt.
Die Ausgabe kann nur in einem Fenster erfolgen.
<%
}
%>
<!--#include file="footer.html"-->
```

*Listing 13.21: Quellcode der Nachrichten-Ausgabe:* `messages.jsp`

## 13.4 Konfiguration

Nachdem wir die Einzelteile kodiert haben, führen wir nun alle Teile über die Konfiguration zusammen. Bevor wir uns der SMI-Definitionsdatei widmen, wollen wir noch einen Blick auf die Store-Konfiguration werfen.

### 13.4.1 Store-Konfiguration

Zunächst muß natürlich die Tabelle für die Nutzer angelegt werden. Den SQL-Anweisungen entspricht Listing 13.22. Als Tabellennamen nutzen wir ChatUser statt User, da User in einigen Datenbanken ein reservierter Tabellenname ist. Nach dem Anlegen der

Tabelle fügen wir den Typ `User` und seine (willkürlich gewählte) Typnummer `200` noch in die Tabelle `TypeNumber` ein.

```
// Tabelle ChatUser
CREATE TABLE ChatUser (
   ObjectIdentifier CHAR(16) NOT NULL,
   ObjectVersion INTEGER,
   Name VARCHAR,
   Password VARCHAR,
   Color INTEGER,
   PRIMARY KEY (ObjectIdentifier)
);
INSERT INTO TypeNumber VALUES ('User', 200);
```

*Listing 13.22: SQL-Anweisungen zum Anlegen der ChatUser-Tabelle: `Chat.sql`*

Nachdem die Datenbank-Tabellen sangelegt sind, müssen wir noch die Store-Konfigurationsdatei `Chat.store` (Listing 13.23) anlegen. Aus Gründen der Übersichtlichkeit haben wir die Definition des Typs `User` in die separate Datei `User.persistence` (Listing 13.24) ausgelagert.

```
{
   StoreClass = "de.webapp.Framework.SMICommandListener.SMIStore";
   // Beispiel-Verbindungsdaten
   JDBCConnection = {
      DriverClass = "sun.jdbc.odbc.JdbcOdbcDriver";
      URLConnect = "jdbc:odbc:webapp";
      Properties = {
         user = "web" ;
         password = "app" ;
      }
   };
   Types = {
      User = #include("User.persistence");
   }
}
```

*Listing 13.23: Store-Konfigurationsdatei `Chat.store`*

```
{
   Classes = {
      FactoryClass = "de.webapp.Framework.Persistence.PersistenceFactoryMSAccess";
   };
   Type = {
      Name = "User";
      Class = "de.webapp.Chat.User";
      Entity = "ChatUser";
      EntityAlias = "tO";
      Attributes = (
         "ObjectIdentifier",
         "ObjectVersion",
         "Name",
```

```
            "Password",
            // da Color kein String ist, muß das Attribut
            // genauer definiert werden.
            {
                Name = "Color";
                TypeClass = "java.lang.Integer";
            },
        );
    }
}
```

*Listing 13.24: Definition des Typs* `User`: `User.persistence`

### 13.4.2 SMI-Konfiguration

Jetzt müssen wir nur noch die SMI-Definitionsdatei erstellen. Ihr Inhalt legt wiederum das Zusammenspiel zwischen Kommandos und Methoden der einzelnen Listener fest. Da wir einen eigenen `I_SMIEventSwitch` benutzen möchten, muß dessen Klassenname zusätzlich unter dem Schlüssel `EventSwitchClass` angegeben werden. Außerdem müssen wir eine zusätzliche Konfigurationsdatei `Chat.con` (Listing 13.26) anlegen, die den `I_SMIContext` definiert. Ansonsten ist alles wie gehabt.

```
{
   // Grundlegenden Konfiguration
   EventSwitchClass = "de.webapp.Chat.ChatSwitch";
   // Definition der Standardwerte
   DefaultCommand = "showLogin";
   Values = {
      _FollowUpCommand = "display";
      _TemplateName = "login.jsp";
   }
   // Definition der Listener
   Listener = {
      ChatRoom = {
         Class = "de.webapp.Chat.ChatRoom";
      },
      ChatRegistration = {
         Class = "de.webapp.Chat.ChatRegistration";
      },
      ChatDispatcher = {
         Class = "de.webapp.Chat.ChatDispatcher";
      },
      DisplayBean = {
         Class = "de.webapp.Framework.SMICommandListener.DisplayBean";
      }
   };
   // Definition der Kommandos
   Commands = {
      // Anzeige
      display = {
         MethodName = "display";
         Listener = "DisplayBean";
```

```
}
//Zeige Anmeldung
showLogin = {
   MethodName = "noop";
   Listener = "ChatRoom";
   Values = {
      _TemplateName = "login.jsp"
   }
}
// Authentifizierung
enterRoom = {
   MethodName = "enter";
   Listener = "ChatRoom";
   Values = {
      _TemplateName = "frame.html"
      userNotLoggedIn = "notloggedin.html"
   }
}
// Anzeige einer Nachricht
showMessages = {
   MethodName = "showMessages";
   Listener = "ChatRoom";
   Values = {
      _TemplateName = "messages.jsp"
      userNotLoggedIn = "notloggedin.html"
   }
}
// Anzeige der Nachricheneingabe
showSendDialog = {
   MethodName = "noop";
   Listener = "ChatRoom";
   Values = {
      _TemplateName = "sendform.jsp"
   }
}
// Sende Nachricht
sendMessage = {
   MethodName = "send";
   Listener = "ChatDispatcher";
   Values = {
      _TemplateName = "sendform.jsp"
      userNotLoggedIn = "notloggedin.html"
   }
}
// Eingabe der persönlichen Daten
showRegistration = {
   MethodName = "noop";
   Listener = "ChatRegistration";
   Values = {
      _TemplateName = "register.jsp"
   }
}
```

```
      // Anzeige des Nutzeränderungsdialogs
      showUpdateDialog = {
         MethodName = "noop";
         Listener = "ChatRegistration";
         Values = {
            _TemplateName = "register.jsp"
         }
      }
      // Erzuege neuen Nutzer
      createUser = {
         MethodName = "create";
         Listener = "ChatRegistration";
         Values = {
            failure = "register.jsp"
            _TemplateName = "menu.jsp"
         }
      }
     // Änderung eines Nutzers
      updateUser = {
         MethodName = "update";
         Listener = "ChatRegistration";
         Values = {
            _TemplateName = "menu.jsp"
            failure = "register.jsp"
         }
      }
      // Abmelden
      logout = {
         MethodName = "logout";
         Listener = "ChatRoom";
      }
      // Anmelden
      login = {
         MethodName = "login";
         Listener = "ChatRoom";
         Values = {
            _TemplateName = "menu.jsp"
            errorOnEnter = "login.jsp"
         }
      }
      // verlassen des Raums
      leaveRoom = {
         MethodName = "leave";
         Listener = "ChatRoom";
         Values = {
            _TemplateName = "menu.jsp"
         }
      }
   }
}
```

*Listing 13.25: SMI-Definitionsdatei des Chats*

```
{
   Class = "de.webapp.Chat.ChatContext";
   Values = {
      _StoreName = "Chat.store";
   }
}
```

*Listing 13.26: Konfigurationsdatei für den `I_SMIContext: Chat.con`*

Als letztes fehlt noch die Registrierung bei der Servlet-Engine. Der Eintrag für jo! ist in Listing 13.27 abgedruckt.

```
# Chat
servlet.Chat.code=de.webapp.Framework.SMI.SMIServlet
servlet.Chat.initArgs=EventSwitch=Chat,Context=Chat
servlet.Chat.aliases=/Chat/Chat
```

*Listing 13.27: Chat-Eintrag in der `servlet.properties`-Datei von jo!*

Der Chat sollte nach dem Start unter dem URL `http://<hostname>:<port>/Chat/Chat` bereitstehen!

# 14 Schlußbemerkungen

In den vorherigen Kapiteln haben Sie gesehen, wie man mit dem WebApp-Framework webbasierte Anwendungen entwickelt. Natürlich hat es sich dabei um vergleichsweise kleine Beispiele gehandelt. Jedoch haben sie die Fähigkeiten von SMI zur Modularisierung auch bei weitem nicht ausgenutzt. So wurde in keinem der Beispiele mehr als ein I_SMIEventSwitch beziehungsweise SMIServlet benötigt. Dennoch haben wir demonstriert, daß ein Applikationsmodell wie SMI mit Persistence als Grundlage den Entwicklungsprozeß stark vereinfachen kann und zu besseren Anwendungen führt als die bloße Entwicklung mit Servlets und JDBC.

Durch die inhärente Abstraktion SMIs erhalten Sie die Möglichkeit, das Verhalten ganzer Anwendungen über Konfigurationsdateien zu steuern. Sie werden in die Lage versetzt, anspruchsvolle webbasierte Anwendungen zu entwickeln, die auch langfristig Bestand haben werden.

Wesentlich zur Eleganz der Realisierungen hat zudem das Persistence-Framework beigetragen. Wir haben gezeigt, daß es möglich ist, mit seiner Hilfe sehr leicht einen schlanken, objektorientierten und webbasierten Datenbank-Browser zu programmieren. Ebenso ist der Code-Anteil zur Realisierung von Persistenz in den anderen beiden Beispielen bemerkenswert gering.

Teil Drei dieses Buches demonstrierte zudem die Anwendung eines Entwicklungsprozeßmodells, das Ihnen die Analyse vereinfacht und die Realisierung Ihrer Anwendung stark beschleunigen kann. Es fördert außerdem die wichtige Trennung von Anwendungslogik und Anzeige und trägt so entscheidend zur Zukunftsfähigkeit Ihrer Anwendungen bei.

Wir hoffen, daß Ihnen dieses Buch einige neue Einsichten vermittelt hat. Unser größter Erfolg wäre es, wenn wir Sie überzeugen konnten, ihre Anwendungen künftig ein wenig anders zu schreiben.

# Anhang

# A ConfigFileReader-Dateiformat

Der Name einer Datei des ConfigFileReader-Formats (Kapitel 4) sollte mit dem Suffix .cfg enden. Die Datei selber muß der Backus-Naur-Form (BNF) aus Listing A.1 genügen:

```
body : hashtable
     | vector
literal : STRING
        | INTEGER
        | DOUBLE
        | MINUS INTEGER
        | MINUS DOUBLE
        | TRUE
        | FALSE
object : literal
       | hashtable
       | vector
       | includestatement literal
       | includestatement hashtable
       | includestatement vector
hashtable : OPENCURLYBRACE hashelements CLOSECURLYBRACE
          | OPENCURLYBRACE CLOSECURLYBRACE
hashelements : NAME IS object
             | hashelements NAME IS object
vector : OPENBRACE vectorelements CLOSEBRACE
       | OPENBRACE CLOSEBRACE
vectorelements : object
               | vectorelements object
includestatement : INCLUDE OPENBRACE STRING CLOSEBRACE
```

*Listing A.1: BNF des `ConfigFileReader`-Formats*

Dabei gelten die Definitionen aus Listing A.2. Java-konforme Kommentare, Kommata, Semikolons, Leerzeichen, Tabulatoren etc. werden beim Parsen ignoriert.

```
STRING: beliebige Zeichenkette
INTEGER: [0-9]+ (ganze Zahl)
DOUBLE: [0-9]+\.[0-9]+ (Dezimalbruch)
MINUS: '-'
TRUE: 'true'
FALSE: 'false'
OPENBRACE: '('
CLOSEBRACE: ')'
OPENCURLYBRACE: '{'
CLOSECURLYBRACE: '}'
NAME: [A-Za-z_][A-Za-z_0-9]* (Bezeichner)
IS: '='
INCLUDE: '#include'
```

*Listing A.2: Definition der Tokens*

# B Konfigurationsmöglichkeiten des Persistence-Frameworks

Das Persistence-Framework (Kapitel 9) benötigt für seine korrekte Funktion ein Modell. Der `Store` wird auf Basis einer Beschreibung des Modells durch die `StoreFactory` generiert. Ein Prinzip des WebApp-Frameworks ist es, daß die verschiedenen Funktionsbereiche über Schnittstellen definiert sind und erst zur Laufzeit daraus ein funktionsfähiges Ganzes gebildet wird. Die Konfiguration des Persistence-Frameworks trägt diesem Prinzip Rechnung. So sind alle `Store`- und `Peer`-Dienste in der Konfiguration austauschbar. Eine Anpassung an eigene Dienste ist sehr einfach möglich. Natürlich verfügt die `StoreFactory` über Voreinstellungen.

| **Parameter** | **Optional** | **Erklärung** |
|---|---|---|
| `Name` | Nein | Name des Stores |
| `StoreClass` | Ja | Klasse der Instanz des Stores<br>**Voreinstellung:** `de.webapp.Framework.Persistence.Store` |
| `ConnectionClass` | Ja | Verbindung zur Datenbank<br>**Voreinstellung:** `de.webapp.Framework.Persistence.StoreConnection` |
| `TransactorClass` | Ja | Klasse des `Transactors`<br>**Voreinstellung:** `de.webapp.Framework.Persistence.StoreTransactor` |
| `ModifierClass` | Ja | Klasse des Modifiers<br>**Voreinstellung:** `de.webapp.Framework.Persistence.StoreModifier` |
| `FactoryClass` | Ja | Klasse der Factory<br>**Voreinstellung:** `de.webapp.Framework.Persistence.StorePersistenceFactory` |
| `AutonumberClass` | Ja | Klasse zur automatischen Numerierung<br>**Voreinstellung:** `de.webapp.Framework.Persistence.StoreOIDAutonumber` |
| `JDBCConnection` | Nein | Angabe der Verbindungsparameter einer JDBC-`Connection` |
| `Types` | Nein | Angabe der `Persistence`-Typen (Tabelle 9.4) |

*Tabelle B.1: Parameter der `Store`-Konfiguration*

Die vollständige Konfigurationsdatei `Restaurant.store` aus dem Kapitel 9 sieht also folgendermaßen aus:

```
{
   Name = "Restaurant" ;
   StoreClass = "de.webapp.Framework.Persistence.Store" ;
   FactoryClass = "de.webapp.Framework.Persistence.StorePersistenceFactory" ;
   AutonumberClass = "de.webapp.Framework.Persistence.StoreOIDAutonumber" ;
   ModifierClass = "de.webapp.Framework.Persistence.StoreModifier" ;
   TransactorClass = "de.webapp.Framework.Persistence.StoreTransactor" ;
   ConnectionClass = "de.webapp.Framework.Persistence.StoreConnection" ;
   JDBCConnection = {
      // Beispiel-Verbindung.
      DriverClass = "solid.jdbc.SolidDriver" ;
      URLConnect = "jdbc:solid://localhost:1313" ;
      Properties = {
         user = "web" ;
         password = "app" ;
      }
   } ;
   Types = {
      Restaurant = #include("Restaurant.persistence") ;
      Gericht = #include("Gericht.persistence") ;
   }
} // Ende der Konfiguration
```

*Listing B.1: Vollständige Konfiguration eines `Stores`*

Die Parameter `JDBCConnection` und `Types` besitzen noch weitere Parameter. Für eine JDBC-Datenbankverbindung muß durch die Angabe des Parameters `DriverClass` die Klasse des JDBC-Adapters definiert werden. Mit `URLConnect` wird eine spezielle Datenquelle der gewählten Datenbank angegeben. Durch `Properties` lassen sich noch Parameter, wie die Anmeldung, beschreiben.

Der Parameter `Types` definiert die einzelnen `Persistence`-Typen des `Stores`. Im Beispiel wurden die Daten auf zwei separate Dateien verteilt.

| Parameter | Optional | Erklärung |
|---|---|---|
| `Classes;PeerClass` | Ja | Klasse des Peers<br>Voreinstellung: `de.webapp.Framework.Persistence.PersistencePeer` |
| `Classes;TypeClass` | Ja | Klasse für die Typbeschreibung<br>Voreinstellung: `de.webapp.Framework.Persistence.PersistenceType` |

*Tabelle B.2: Die Parameter eines `PersistencePeers`*

| Parameter | Optional | Erklärung |
|---|---|---|
| `Classes;CacherClass` | Ja | Klasse für den Cache<br>Voreinstellung: `de.webapp.Framework.Persistence.PersistenceCache` |
| `Classes;RetrieverClass` | Ja | Klasse für die Suche nach Objekten<br>Voreinstellung: `de.webapp.Framework.Persistence.PersistenceRetriever` |
| `Classes;FactoryClass` | Ja | Klasse zum Erzeugen von neuen `Persistence`-Objekten und zur Vermittlung der Transformation in ein `Persistence`-Objekt aus der Datenbank<br>Voreinstellung: `de.webapp.Framework.Persistence.PersistenceFactory` |
| `Classes;ModifierClass` | Ja | Klasse zur Speicherung von Änderungen<br>Voreinstellung: `de.webapp.Framework.Persistence.PersistenceModifier` |
| `Classes;PeerClass` | Ja | Klasse des Peers<br>Voreinstellung: `de.webapp.Framework.Persistence.PersistencePeer` |
| `Type;Name` | Nein | Symbolischer Name des Persistence (muß in der Tabelle `TypeNumber` stehen) |
| `Type;Class` | Nein | Klasse, die diesen Typ von Persistence realisiert |
| `Type;Entity` | Nein | Name der Tabelle |
| `Type;EntityAlias` | Nein | Alias der Entity in Anfragen |
| `Type;Attributes` | Nein | Definition der Attribute |
| `Type;Attributes;Name` | Nein | Name des Attributes |
| `Type;Attributes;Key` | Ja | Ist eindeutiger Schlüssel der Tabelle |
| `Type;Attributes;MainKey` | Ja | Ist Attribut des `ObjectIdentifiers` |
| `Type;Attributes;Generic` | Ja | Markierung generischer Attribute für die Klassen `GenericPersistence` und `GenericRelationPersistence` |
| `Type;Attributes;TypeClass` | Ja | Typ des Attributs<br>Die Angabe muß erfolgen, wenn der Typ nicht `java.lang.String` ist. |
| `Type;Attributes;Property` | Ja | Name des Attributs in der Datenbank |
| `Type;Attributes;Field` | Ja | Name des Feldes für das Attribut in der Klasse |

*Tabelle B.2: Die Parameter eines `PersistencePeers` (Fortsetzung)*

| Parameter | Optional | Erklärung |
|---|---|---|
| `Type;Attributes; GetMethod` | Ja | Name der Zugriffsmethode für das Attribut |
| `Type;Attributes; SetMethod` | Ja | Name der Änderungsmethode für das Attribut |
| `Type;Associations; Name` | Nein | Name der Assoziation<br>Wichtig, da dieser für die Zugriffsmethoden der Anwendung und Klasse genutzt wird. |
| `Type;Associations; Type` | Nein | Kardinalität der Relation (`ToOne`, `ToMany`, `ToManyLink`) |
| `Type;Associations; Modified` | Ja | Falls es sich um eine Aggregation handelt, muß dieser Parameter auf `YES` gesetzt werden. Damit wird dann automatisch die richtige Referenzklasse gewählt. |
| `Type;Associations; ResultType` | Nein | Name der `Persistence`-Objekte, die im Anfrageergebnis enthalten sind (siehe `Type;Attributes;Name`). |
| `Type;Associations; Key` | Nein | Name des Schlüsselattributs (Fremdschlüssel in der eigenen Tabelle (1:1) oder in der des `ResultType` (1:N)) |
| `Type;Associations; ReferenceClass` | Ja | Referenz-Klasse. Normalerweise wird die Klasse über den Typ der Assoziation ersichtlich. |
| `Type;Associations; QualifierExtension` | Ja | Angabe einer statischen Qualifikationsergänzung, z. B. `Order by tO.Name` |
| `Type;Associations; LinkType` | Ja | Angabe des Link-Typs. Ist notwendig für `ToManyLink` |
| `Type;Associations; LinkLeftKey` | Ja | Angabe des Fremdschlüssels – Verknüpfung zur Basistabelle |
| `Type;Associations; LinkRightKey` | Ja | Angabe des Fremdschlüssels – Verknüpfung mit Qualifikationsparameter |

*Tabelle B.2: Die Parameter eines `PersistencePeers` (Fortsetzung)*

Die Angabe der Dienstklassen des `PersistencePeers` ist optional. Bei der Verwendung von MS Access als Datenbank ist die Wahl der `FactoryClass` als `de.webapp.Framework.Persistence.PersistenceFactoryMSAccess` allerdings zwingend. Leider ist der ODBC-Adapter von MS-Access 97 fehlerhaft. Er erlaubt es nicht, zweimal auf das selbe Attribut eines `ResultSets` zuzugreifen.

Die Angabe des Typs ist zwingend. Dazu gehört insbesondere eine Klasse und die entsprechenden Abbildungsparameter auf die Datenbanktabellen. Alle Anweisungen in Kommentarzeichen (`//`) sind optionale Parameter. Einige Anweisungen haben keinen direkten Bezug zum Beispiel und sollen nur die genaue Position einer solchen Anweisung aufzeigen (`// **`).

```
{
   Classes = {
      PeerClass = "de.webapp.Framework.Persistence.PersistencePeer" ;
      TypeClass = "de.webapp.Framework.Persistence.PersistenceType" ;
      CacherClass = "de.webapp.Framework.Persistence.PersistenceCacher" ;
      RetrieverClass = "de.webapp.Framework.Persistence.PersistenceRetriever" ;
      FactoryClass = "de.webapp.Framework.Persistence.PersistenceFactory" ;
      // FactoryClass = "de.webapp.Framework.Persistence.PersistenceFactoryMSAccess"
;
      ModifierClass = "de.webapp.Framework.Persistence.PersistenceModfifier" ;
   }
   Type = {
      Name = "Restaurant" ;
      Class = "de.webapp.Examples.Persistence.Restaurant.Restaurant";
      // ** Class = "de.webapp.Framework.SMICommandListener.SMIPersistenceGeneric";
      Entity = "Restaurant";
      EntityAlias = "t0" ;
      Attributes = (
      {
         Name = "ObjectIdentifier";
         Key = "YES";
         MainKey = "YES";
      },{
         Name = "ObjectVersion";
      },{
         // Generic = "YES" ;
         // Key = "YES";
         Name = "Name";
         TypeClass = "java.lang.String" ;
         // Property = "Name" ;
         // Field = "myName" ;
         // GetMethod = "getName" ;
         // SetMethod = "setName" ;
      },{
         // Generic = "YES"
         Name = "Beschreibung";
         TypeClass = "java.lang.String" ;
      }
      ) ;
      // Die Kurzform ist:
      // Attributes = (
      // "ObjectIdentifier",
      // "ObjectVersion",
      // "Name";
      // "Beschreibung";
      // ) ;
      Associations = (
      {
         Name = "Gericht" ;
         Type = "ToMany" ;
         ResultType = "Gericht" ;
```

```
        Key = "RestaurantID" ;
        // ReferenceClass =
"de.webapp.Framework.Persistence.ToManyModifiedReference" ;
        // Modified = "YES" ;
        // QualifierExtension = " Order By t0.Beschreibung" ;
        // ** LinkType = "LieferantArtikel" ;
        // ** LinkLeftKey = "LieferantID" ;
       // ** LinkRightKey = "ArtikelID" ;
     }
     ) ;
  }
} // Ende der Konfiguration
```

*Listing B.2: Vollständige Konfiguration eines Persistence-Objekts von* `Restaurant`

Jede Beschreibung eines `Persistence`-Objekts besteht aus den Teilen Dienste, Klassen, Attributen und Assoziationen. Die Beschreibung dient dabei einerseits zur Bereitstellung der Zugriffsdienste für den `Store`, andererseits werden die Dienste mit den Parametern für die Abbildung zur relationalen Datenbank versorgt. Durch die Konfiguration des `PersistenceType`-Dienstes jedes `PersistencePeers` steht zur Laufzeit eine leistungsfähige generische Beschreibung jedes Typs zur Verfügung. Grundsätzlich existieren zwei verschiedene Beschreibungstypen. Eine Beschreibung dient der Definition von generischen `Persistence`-Objekten, die andere der Definition von `Persistence`-Objekten von direkten Ableitungen der Klasse `de.webapp.Framework.Persistence.Persistence`. In einer frühen Phase eines Projekts kann auf Basis des generischen `Persistence`-Objekts `PersistenceGeneric` schnell ein Prototyp der Geschäftsobjekte zur Verfügung gestellt werden. Wenn Geschwindigkeit eine große Rolle spielt oder die Semantik eines Geschäftsobjekts in der Anwendung mehr in den Vordergrund gerückt wird, ist es ratsam, echte Klassen bereitzustellen.

# C Deployment-Deskriptor DTD

Die DTD (Document Type Definition) gibt eine Grammatik an, nach der gültige Deployment-Deskriptoren für Servlet-Engines aufgebaut werden müssen. Die im folgenden angegebene DTD stammt aus dem Servlet-API 2.2 Public Release vom 18. August 1999.

```
<!-- Das web-app-Element ist die Basis des Deployment-Deskriptors
für eine Web-Applikation -->

<!ELEMENT web-app (icon?, display-name?, description?, distributable?,
context-param*, servlet*, servlet-mapping*, session-config?, mime-mapping*,
welcome-file-list?, error-page*, resource-ref*, security-constraint*,
login-config?, security-role*, env-entry*, ejb-ref*)>

<!-- Das icon-Element enthält ein small-icon-Element und ein large-icon-Element,
welche die Adressen innerhalb der Web-Applikation für ein kleines und ein großes Bild
repräsentieren. Diese Bilder werden von Werkzeugen mit grafischer Oberfläche benutzt.
Diese Werkzeuge müssen zumindest GIF oder JPEG als Grafikformate unterstützen. -->

<!ELEMENT icon (small-icon?, large-icon?)>

<!-- Das small-icon-Element enthält die Adresse (URI) einer Datei innerhalb der Web-
Applikation, die ein kleines (16x16 Pixel) Icon enthält. -->

<!ELEMENT small-icon (#PCDATA)>

<!-- Das large-icon-Element enthält die Adresse (URI) einer Datei innerhalb der Web-
Applikation, die ein großes (32x32 Pixel) Icon enthält. -->

<!ELEMENT large-icon (#PCDATA)>

<!-- Das display-name-Element enthält einen kurzen Namen, der zum Anzeigen in
grafischen Werkzeugen gedacht ist. -->

<!ELEMENT display-name (#PCDATA)>
<!-- Das description-Element enthält einen beschreibenden Text über das Elter-
Element. -->

<!ELEMENT description (#PCDATA)>
```

```
<!-- Ist das distributable-Element angegeben, so wird damit ausgedrückt, daß diese
Web-Applikation geeignet ist, in einem verteilten Servlet-Container installiert zu
werden. -->

<!ELEMENT distributable EMPTY>

<!-- Das context-param-Element enthält die Deklaration von Initialisierungsparametern
eines ServletContextes dieser Web-Applikation. -->

<!ELEMENT context-param (param-name, param-value, description?)>

<!-- Das param-name-Element enthält den Namen eines Parameters. -->

<!ELEMENT param-name (#PCDATA)>

<!-- Das param-value-Element enthält den Wert eines Parameters. -->

<!ELEMENT param-value (#PCDATA)>

<!-- Das servlet-Element enthält beschreibende Daten über ein Servlet. Falls eine
JSP-Datei angegeben wird und das load-on-startup-Element gesetzt ist, sollte die JSP
von der Engine vorkompiliert und geladen werden. -->

<!ELEMENT servlet (icon?, servlet-name, display-name?, description?,
(servlet-class|jsp-file), init-param*, load-on-startup?, security-role-ref*)>

<-- Das servlet-name-Element enthält den kanonischen Namen des Servlets. -->

<!ELEMENT servlet-name (#PCDATA)>

<!-- Das servlet-class-Element enthält den vollständigen Klassennamen des Servlets. -
->

<!ELEMENT servlet-class (#PCDATA)>

<!-- Das jsp-file-Element enthält den vollständigen Pfad zu einer JSP-Datei innerhalb
der Web-Applikation. -->

<!ELEMENT jsp-file (#PCDATA)>

<!-- Das init-param-Element enthält ein Schlüssel/Wert-Paar als
Initialisierungsparameter des Servlets. -->

<!ELEMENT init-param (param-name, param-value, description?)>

<!-- Das load-on-startup-Element zeigt an, daß dieses Servlet beim Start der Web-
Applikation geladen werden sollte. Optional kann eine ganze Zahl angegeben werden,
mit der die Reihenfolge des Ladens beeinflußt wird. Servlets mit niedrigeren Werten
werden dabei vor Servlets mit höheren Werten geladen. Falls keine Reihenfolge
angegeben wurde, kann der Container die Servlets in beliebiger Reihenfolge laden. -->
```

```
<!ELEMENT load-on-startup (#PCDATA)>

<!-- Das servlet-mapping-Element definiert eine Abbildung zwischen einem Servlet und
einem URL-Muster. -->

<!ELEMENT servlet-mapping (servlet-name, url-pattern)>

<!-- Das url-pattern-Element enthält das URL-Muster der Abbildung. Es muß den Regeln
aus Abschnitt 10 der Servlet-Spezifikation folgen (Kapitel 3.13.1). -->

<!ELEMENT url-pattern (#PCDATA)>

<!-- Das session-config-Element definiert die Session-Parameter für diese Web-
Applikation. -->

<!ELEMENT session-config (session-timeout?)>

<!-- Das session-timeout-Element definiert den Standard-Timeout-Intervall für alle
Sessions dieser Web-Applikation. Der Wert muß in vollen Minuten angegeben werden. -->

<!ELEMENT session-timeout (#PCDATA)>

<!-- Das url-Element definiert einen URL zu einer nicht in der Web-Applikation
enthaltenen, externen Ressource. -->

<!ELEMENT url (#PCDATA)>

<!-- Das mime-mapping-Element definiert eine Abbildung zwischen einer
Dateierweiterung und einem MIME-Typ. -->

<!ELEMENT mime-mapping (extension, mime-type)>

<!-- Das extension-Element enthält eine Zeichenkette, die eine Dateierweiterung
beschreibt. Beispiel: ".txt" -->

<!ELEMENT extension (#PCDATA)>

<!-- Das mime-type-Element enthält einen definierten MIME-Typ. Beispiel: "text/rlain"
-->

<!ELEMENT mime-type (#PCDATA)>

<!-- Die welcome-file-list enthält eine geordnete Liste von Begrüßungs-Dateien.. -->

<!ELEMENT welcome-file-list (welcome-file+)>

<!-- Das welcome-file-Element enthält einen Dateinamen, der als Standard-Dateiname
benutzt werden soll, falls kein anderer Dateiname gegeben ist. Beispiel: "index.html"
-->

<!ELEMENT welcome-file (#PCDATA)>
```

```
<!-- Das error-page-Element enthält eine Abbildung von einem Fehlercode oder
Ausnahme-Typ auf den Pfad einer Ressource in einer Web-Applikation. -->

<!ELEMENT error-page ((error-code | exception-type), location)>

<!-- error-code enthält einen HTTP-Errorcode, beispielsweise: 404 -->

<!ELEMENT error-code (#PCDATA)>

<!-- Das exception-type-Element enthält den vollständigen Java-Klassennamen einer
Exception. -->

<!ELEMENT exception-type (#PCDATA)>

<!-- Das location-Element enthält den URI der Ressource innerhalb der Web-
Applikation. -->

<!ELEMENT location (#PCDATA)>

<!-- Das resource-ref-Element enthält die Beschreibung einer Referenz zu einer
externen Ressource. -->

<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth)>

<!-- Das res-ref-name-Element spezifiziert einen Ressource-Factory Referenznamen. -->

<!ELEMENT res-ref-name (#PCDATA)>

<!-- Das res-type-Element spezifiziert den Typ (beziehungsweise die Java-Klasse) der
Ressource. -->

<!ELEMENT res-type (#PCDATA)>

<!-- Das res-auth-Element zeigt an, ob die Applikations-Komponente sich
programmatisch bei der Ressource einloggt oder ob der Servlet-Container dies
automatisch auf Basis der Principal-Abbildung erledigen soll. Mögliche Werte:
CONTAINER oder SERVLET>

<!ELEMENT res-auth (#PCDATA)>

<!-- Das security-constraint-Element wird benutzt, um Security-Constraints mit einer
oder mehreren Web-Ressourcen-Sammlungen (Web Resource Collections) zu assoziieren. --
>

<!ELEMENT security-constraint (web-resource-collection+, auth-constraint?, user-data-
constraint?)>

<!-- Das web-resource-collection-Element dient dazu, eine Untermenge der Ressourcen
und HTTP-Methoden einer Web-Applikation zu definieren, die Sicherheits-Bedingungen
(Security Constraints) unterliegen. Falls keine HTTP-Methoden angegeben wurden,
unterliegen alle Methoden den Sicherheits-Bedingungen. -->
```

```
<!ELEMENT web-resource-collection (web-resource-name, description?, url-pattern*,
http-method*)>

<!-- Der web-resource-name enthält den Namen einer Web-Ressourcen-Sammlung (Web
Resource Collection). -->

<!ELEMENT web-resource-name (#PCDATA)>

<!-- Das http-method-Element enthält den Namen einer HTTP-Methode (GET | POST |...) -
->

<!ELEMENT http-method (#PCDATA)>

<!-- Das user-data-constraint-Element zeigt an, wie Daten, die zwischen Client und
Engine transportiert werden, geschützt werden sollten. -->

<!ELEMENT user-data-constraint (description?, transport-guarantee)>

<!-- Das transport-guarantee-Element spezifiziert, daß die Kommunikation zwischen
Client und Server NONE, INTEGRAL oder CONFIDENTIAL sein sollte. -->

<!ELEMENT transport-guarantee (#PCDATA)>

<!-- Das auth-constraint-Element zeigt an, welche Nutzer-Rollen Zugang zu dieser
Ressourcen-Sammlung haben. Die hier angegebenen Rollen müssen sich auch in einem
security-role-ref Element befinden. -->

<!ELEMENT auth-constraint (description?, role-name*)>

<!-- Das role-name-Element enthält den Namen einer Sicherheits-Rolle. -->

<!ELEMENT role-name (#PCDATA)>

<!-- Das login-config-Element spezifiziert die Authentifikations-Methode, den Realm-
Namen und die Attribute, die für formularbasierte Authentifikation benutzt werden
sollten. -->

<!ELEMENT login-config (auth-method?, realm-name?, form-login-config?)>

<!-- Das realm-name-Element gibt den Realm-Namen an, der bei HTTP-Basic-
Authentifikation benutzt werden soll. -->

<!ELEMENT realm-name (#PCDATA)>

<!-- Das form-login-config-Element spezifiziert die Login- und Error-Seiten, die bei
einem formularbasierten Login benutzt werden sollten. Falls kein formularbasiertes
Login benutzt wird, werden diese Elemente ignoriert. -->

<!ELEMENT form-login-config (form-login-page, form-error-page)>

<!-- Das form-login-page-Element definiert den Ort innerhalb der Web-Applikation, an
dem sich eine zum Login geeignete Seite befindet. -->
```

```
<!ELEMENT form-login-page (#PCDATA)>

<!-- Das form-error-page-Element bezeichnet den Ort innerhalb der Web-Applikation, an
dem sich eine Fehler-Seite befindet, die angezeigt wird, falls ein Login mißlingt. --
>

<!ELEMENT form-error-page (#PCDATA)>

<!-- Das auth-method-Element dient dazu, den Authentifikation-Mechanismus für diese
Web-Applikation zu definieren. Um auf die geschützten Ressourcen zugreifen zu können,
muß sich der Nutzer zuvor mittels des angegeben Verfahrens ausgewiesen haben.
Erlaubte Werte sind "BASIC", "DIGEST", "FORM" oder "CLIENT-CERT". -->

<!ELEMENT auth-method (#PCDATA)>

<!-- Das security-role-Element bezeichnet eine Sicherheitsrolle dieser Web-
Applikation. -->

<!ELEMENT security-role (description?, role-name)>

<!-- Das role-name-Element enthält den Namen einer Rolle. Dabei muß es sich um eine
nicht-leere Zeichenkette handeln. -->

<!ELEMENT role-name (#PCDATA)>

<!-- Das security-role-ref-Element enthält die Deklaration einer Sicherheits-Rollen-
Referenz im Servlet Code. Der Inhalt von role-name wird im Servlet benutzt, um eine
bestimmte Rolle zu spezifizieren. role-link verknüpft diese im Code benutzte Rolle
mit einer Sicherheits-Rolle. Diese Sicherheits-Rolle muß durch ein security-role-
Element spezifiziert sein. -->

<!ELEMENT security-role-ref (description?, role-name, role-link)>

<!-- Das role-link-Element wird benutzt, um eine Sicherheits-Rollen-Referenz mit
einer Sicherheits-Rolle zu verknüpfen. Das role-link-Element muß daher den Namen
einer Sicherheits-Rolle (security-role) enthalten. -->

<!ELEMENT role-link (#PC-DATA)>

<!-- Das env-entry-Element enthält die Deklaration eines Applikations-Umgebungs-
Eintrages. Nur J2EE fähige Servlet-Engines müssen dieses Element beachten. -->

<!ELEMENT env-entry (description?, env-entry-name, env-entry-value?, env-entry-type)>

<!-- Das env-entry-name-Element enthält den Namen des Umgebungs-Eintrages einer
Applikation. -->

<!ELEMENT env-entry-name (#PCDATA)>

<!-- Das env-entry-value-Element enthält den Wert eines Umgebungs-Eintrages der
Applikation. -->
```

```
<!ELEMENT env-entry-value (#PCDATA)>

<!-- Das env-entry-type-Element enthält des vollständigen Java-Klassennamen des
Umgebungs-Eintrages, der vom Applikationscode erwartet wird. Folgende Werte sind
möglich: java.lang.Boolean, java.lang.String, java.lang.Integer, java.lang.Double,
java.lang.Float. -->

<!ELEMENT env-entry-type (#PCDATA)>

<!-- Das ejb-ref-Element dient dazu, eine Referenz zu einem Enterprise Bean zu
definieren. -->

<!ELEMENT ejb-ref (description?, ejb-ref-name, ejb-ref-type, home, remote, ejb-
link?)>

<!-- Das ejb-ref-name-Element enthält den Namen einer EJB-Referenz. Dies ist der
JNDI-Name, der im Servletcode benutzt werden kann, um eine Referenz auf ein EJB zu
erhalten. -->

<!ELEMENT ejb-ref-name (#PCDATA)>

<!-- Das ejb-ref-type-Element enthält den erwarteten Java Klassentyp des
referenzierten EJB. -->

<!ELEMENT ejb-ref-type (#PCDATA)>

<!-- Das ejb-home-Element enthält den vollständigen Klassennamen des EJB Home-
Interfaces -->

<!ELEMENT home (#PCDATA)>

<!-- Das ejb-remote-Element enthält den vollständigen Klassennamen des EJB Remote-
Interfaces. -->

<!ELEMENT remote (#PCDATA)>

<!-- Das ejb-link-Element dient dazu festzulegen, daß eine EJB-Referenz mit einem EJB
eines J2EE-Applikations-Paket verknüpft ist. Der Wert des ejb-link-Elements muß dem
ejb-name eines EJB des J2EE-Applikations-Paketes entsprechen. -->

<!ELEMENT ejb-link (#PCDATA)>

<!-- Der ID-Mechanismus erlaubt es Werkzeugen, den Informationen in diesem
Deployment-Deskriptor werkzeugspezifische Zusatzinformation hinzuzufügen. Werkzeuge
können somit nicht-standardisierte Informationen in Extra-Dateien hinterlegen und
mittels folgender IDs leicht auf diesen Deployment-Deskriptor und die darin
enthaltenen Informationen verweisen. -->

<!ATTLIST web-app id ID #IMPLIED>
<!ATTLIST icon id ID #IMPLIED>
<!ATTLIST small-icon id ID #IMPLIED>
```

```
<!ATTLIST large-icon id ID #IMPLIED>
<!ATTLIST display-name id ID #IMPLIED>
<!ATTLIST description id ID #IMPLIED>
<!ATTLIST distributable id ID #IMPLIED>
<!ATTLIST context-param id ID #IMPLIED>
<!ATTLIST param-name id ID #IMPLIED>
<!ATTLIST param-value id ID #IMPLIED>
<!ATTLIST servlet id ID #IMPLIED>
<!ATTLIST servlet-name id ID #IMPLIED>
<!ATTLIST servlet-class id ID #IMPLIED>
<!ATTLIST jsp-file id ID #IMPLIED>
<!ATTLIST ser-file id ID #IMPLIED>
<!ATTLIST init-param id ID #IMPLIED>
<!ATTLIST load-on-startup id ID #IMPLIED>
<!ATTLIST servlet-mapping id ID #IMPLIED>
<!ATTLIST url-pattern id ID #IMPLIED>
<!ATTLIST session-config id ID #IMPLIED>
<!ATTLIST session-timeout id ID #IMPLIED>
<!ATTLIST url id ID #IMPLIED>
<!ATTLIST mime-mapping id ID #IMPLIED>
<!ATTLIST extension id ID #IMPLIED>
<!ATTLIST mime-type id ID #IMPLIED>
<!ATTLIST welcome-file-list id ID #IMPLIED>
<!ATTLIST welcome-file id ID #IMPLIED>
<!ATTLIST error-page id ID #IMPLIED>
<!ATTLIST error-code id ID #IMPLIED>
<!ATTLIST exception-type id ID #IMPLIED>
<!ATTLIST location id ID #IMPLIED>
<!ATTLIST resource-ref id ID #IMPLIED>
<!ATTLIST res-ref-name id ID #IMPLIED>
<!ATTLIST res-type id ID #IMPLIED>
<!ATTLIST res-auth id ID #IMPLIED>
<!ATTLIST security-constraint id ID #IMPLIED>
<!ATTLIST web-resource-collection id ID #IMPLIED>
<!ATTLIST web-resource-name id ID #IMPLIED>
<!ATTLIST http-method id ID #IMPLIED>
<!ATTLIST user-data-constraint id ID #IMPLIED>
<!ATTLIST transport-guarantee id ID #IMPLIED>
<!ATTLIST auth-constraint id ID #IMPLIED>
<!ATTLIST role-name id ID #IMPLIED>
<!ATTLIST auth-method id ID #IMPLIED>
<!ATTLIST basic-auth id ID #IMPLIED>
<!ATTLIST form-auth id ID #IMPLIED>
<!ATTLIST form-login-page id ID #IMPLIED>
<!ATTLIST form-error-page id ID #IMPLIED>
<!ATTLIST mutual-auth id ID #IMPLIED>
<!ATTLIST ejb-ref id ID #IMPLIED>
<!ATTLIST ejb-name id ID #IMPLIED>
<!ATTLIST ejb-type id ID #IMPLIED>
<!ATTLIST ejb-home id ID #IMPLIED>
<!ATTLIST ejb-remote id ID #IMPLIED>
<!ATTLIST ejb-link id ID #IMPLIED>
```

# D Literatur


[Behme/Mintert 98] Behme, Henning; Mintert, Stefan: *XML – In der Praxis.*, Bonn: Addison-Wesley 1998.

[Berners-Lee 89] Berners-Lee, Tim: *Information Management. A Proposal.* 1989, http://www.w3.org/History/1989/proposal.html.

[Berners-Lee et al. 96] Berners-Lee, Tim; Fielding, Roy; Frystyk, Henrik: *RFC 1945 – Hypertext Transfer Protocol – HTTP/1.0.* 1996, http://www.cis.ohio-state.edu/htbin/rfc/rfc1945.html.

[Booch 94] Booch, Grady: *Object Oriented Analysis and Design.* 2. Auflage, Redwood City: Benjamin/Cummings 1994.

[Buschmann et al. 96] Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; Stal Michael: A System of Patterns. John Wiley & Sons, 1996 Bonn, Paris

[Campione/Walrath 97] Campione, Mary; Walrath, Kathy: *Das Java Tutorial.* 1. Auflage, Bonn: Addison-Wesley 1997.

[Coplien/Schmidt 95] Coplien, James O.; Schmidt, Douglas C.: *Pattern languages of program design.* Reading,Addison-Wesley, 1995.

[Cornell/Horstmann 96] Cornell, Gary; Horstmann, Cay S.: *Java bis ins Detail.* 1. Auflage, Hannover: Verlag Heinz-Heise, 1996.

[Cox/Novobilski 91] Cox, Brad J.; Novobilski, Andrew J.: *Object Oriented Programming.* Addison-Wesley, 1991 Bonn, Paris, New York

[Date/Darwen 98] Date, Chris J.; Darwen, Hugh: *SQL – Der Standard.* Bonn: Addison-Wesley, 1998.

[Eilebrecht 98] Eilebrecht, Lars: *Apache Web-Server*; 2. Auflage, Bonn: International Thomson Publishing, 1998.

[Farley 98] Farley, Jim: *Java Distributed Computing.* Sebastopol, CA: O'Reilly, 1998.

[Fielding et al. 97] Fielding, Roy; Gettys, Jim; Mogul, Jeffrey; Frystyk, Henrik; Berners-Lee, Tim: *RFC 2068 – Hypertext Transfer Protocol – HTTP/1.1.* 1997, http://www.cis.ohio-state.edu/htbin/rfc/rfc2068.html.

[Fields/Kolb 00] Fields, Dvane K.; Kolb, Mark A.: *Web development with Java Server Pages.* Manny Greenwich 2000

[Flanagan 97] Flanagan, David: *JAVA Examples in a Nutshell.* Sebastopol, CA: O'Reilly, 1997.

[Fowler/Scott 98] Fowler, Martin; Scott, Kendall: *UML konzentriert.* Bonn: Addison-Wesley, 1998.

[Gamma et al. 96] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software.* Bonn: Addison-Wesley, 1996.

[Gosling et al. 97] Gosling, James; Joy, Bill; Steele, Guy: *Java – Die Sprachspezifikation.* 1. Auflage, Bonn: Addison-Wesley 1997.

[Hamilton et al. 97] Hamilton, Graham; Cattel, Rick; Fisher, Maydene: *JDBC Database Access with Java. Reading, MA:* Addison-Wesley, 1997.

[Heid 98a] Heid, Jörn: *Servlets als CGI-Alternative.* In: iX 11/1998. Seite 64.

[Heid 98b] Heid, Jörn: *Serverseitiges Scripting mit JSP.* In: iX 11/1998. Seite 68.

[Heid 97] Heid, Jörn: *Kettenreaktion, Servlets als CGI-Ersatz.* In: iX 11/1997. Seite 166.

[Heuer 92] Heuer, Andreas: *Objektorientierte Datenbanken.* Bonn: Addison-Wesley, 1992.

[Heuer/Saake 97] Heuer, Andreas; Saake, Günther: *Datenbanken.* Königswinter: International Thompson Publishing, 1997.

[Holl/Schorsch 98] Holl, Matthias; Schorsch, Stefan: *Objektidentität und Altdaten.* In ObjektSpektrum Nr. 6, Sigs, 1998

[Hughes et al. 97] Hughes, Merlin; Hughes, Conrad; Shoffner, Michael; Winslow, Maria: *Java Network Programming.* Manning 1997.

[Hunter/Crawford 98] Hunter, Jason; Crawford, William: *JAVA Servlet Programming.* Sebastopol, CA: O'Reilly, 1998.

[JDK1.2 98] JavaSoft: *Java Development Kit API 1.2.* 1998, http://www.javasoft.com/products/jsp.

[JSP 98] JavaSoft: *JavaServer-Pages – Specification 0.92.* 1998, http://java.sun.com/products/jsp.

[Klute 98] Klute, Rainer: *Mehr als Applets.* In: iX 11/1998, Seite 60.

[Lea 97] Lea, Doug: *Concurrent Programming in Java.* Reading, Massachusetts: Addison-Wesley, 1997.

[Lemay 97] Lemany, Laura: *Castanet.Indianopolis, IN:* Sams Net, 1997.

[Merkle 97] Merkle, Berhard: *RMI: Verteilte Java-Objekte.* In: iX 12/1997, Seite 130.

[Moss 98] Moss, Karl: *Java Servlets.* New York, NY: McGraw Hill 1998.

[Musciano/Kennedy 97] Musciano, Chuck; Kennedy, Bill: *HTML – Das umfassende Referenzwerk.* Sebastopol, CA: O'Reilly , 1997.

[NCSA 96] NCSA: *Server Side Includes.* 1996, http://hoohoo.ncsa.edu/docs/tutorials/includes.html.

[Oestereich 98] Oestereich, Bernd: *Objektorientierte Softwareentwicklung mit der Unified Modeling Language.* 4. Auflage, München/Wien, R. Oldenbourg Verlag, 1998.

[Orfali et al. 96] Orfali, Robert; Harkley, Dan; Edwards, Jeri: *The essential distributed objects survival guide.* John Wiley & Sons, 1996.

[Orfali/Harkley 97] Orfali, Robert; Harkley, Dan: *Client/server Programming with Java and CORBA.* John Wiley & Sons ,1997. New York, Breslau, Toronto, Singapur

[Polar 98] Firma IBL: verschiedene Veröffentlichungen zur Modellierung von objekt-relationalen Abbildungen 1998, http://www.ibl.de.

[Reese 97] Reese, George: *Database Programming with JDBC and Java.* O'Reilly 1997. Sebastopol, Tikio, Köln, Paris, Cambridge

[Roßbach/Schreiber 97] Roßbach, Peter; Schreiber, Hendrik: *Suns JavaServer 1.1.* In: iX 12/1997, Seite 70.

[Roßbach/Schreiber 98a] Roßbach, Peter; Schreiber, Hendrik: *Server-Programmierung in Java.* In: Java Spektrum 3/1998, Seite 42.

[Roßbach/Schreiber 98b] Roßbach, Peter; Schreiber, Hendrik: Tutorial *objectHTML.* 1998, http://www.factum-gmbh.de.

[Roßbach/Schreiber 98c] Roßbach, Peter; Schreiber, Hendrik: Referenz *objectHTML.* 1998, http://www.factum-gmbh.de.

[Roßbach et al. 98] Roßbach, Peter; Terfloth, Axel; Mimberg, Dirk: *Whitepaper PerFACT.* 1998, http://www.factum-gmbh.de.

[Stevens 90] Richard, Stevens, W.: *Unix Network Programming.* 10. Auflage, Englewood Cliffs, NJ: Prentice-Hall, 1990.

[Turau 00] Turau, Volker: *Java Server Pages*: Dynamische Generierung von Web-Dokumenten. dpunkt Verlag, Heidelberg 2000

[Vlissides et al. 96] Vlissides, John M.; Coplien, James O.; Kerth, Norman L.: *Pattern languages of program design.* Reading, MA: Addison-Wesley, 1996.

[Wilkinson 95] Wilkinson, Nancy M.: *Using CRC Cards.* Sigs Book, 1995.

[Wirfs-Brock et al. 90] Wirfs-Brock, Rebecca; Wilkerson, Brian; Wiener, Lauran: *Design Object Oriented Software.* Englewood Cliffs, NJ: Prentice-Hall 1990.

# E Abbildungsverzeichnis

# F Tabellenverzeichnis

# G Listingverzeichnis

# Index

## D



## E

## F



## G



## H

## I

## J

**P**

## T



## U



## V



## W

## X



## Z

# Zu den Autoren

Dipl. Inform. Peter Roßbach beschäftigt sich seit 1990 mit objektorientierter Analyse, Design und Programmierung und erprobt die Methoden im praktischen Einsatz. Sein besonderes Interesse liegt in der Entwicklung von dezentralen Informationssytemen. Die Architektur robuster Systeme und der Prozeß ihrer Entwicklung haben ihm und seinen Teams so manche harte Nuß beschert. Ganz besonders die Gestaltung und Realisierung von Frameworks und die Anbindung relationaler Datenbanken an die Welt der Objekte liegt ihm am Herzen. Seit 1997 liegt Peter Roßbachs Konzentration auf der Verwendung von Java für Web-Applikationen. Die Leitung der Entwicklung eines verteilten Content-Management Systems für Handel und Banken, das nun mehr als 300 Standorte mit aktuelle Informationen versorgt, haben ihn zum Schreiben gebracht.

Gerüstet mit diesen Erfahrungen gründet er nun ein eigenes Unternehmen, das sich mit der Realisierung von Produkten und Projekten für Java basierte E-Commerce Lösungen beschäftigt.

Dipl. Journ. Hendrik Schreiber beschäftigt sich seit 1996 intensiv mit der Programiersprache Java. 1997 kam er erstmals in Kontakt mit Servlets und arbeitet seitdem kontinuierlich an der Entwicklung javabasierter Web-Applikationen. Ein Teil dieser Arbeit ist die Realisierung der javabasierten Scriptsprache objectHTML sowie der Weiterentwicklung des Webservers jo!