Prof. Dr. Jörg Desel
unter Mitarbeit von Benjamin Meis
unter Verwendung eines Kurstextes von Alexander Lorenz,
Immo Schulz-Gerlach und Prof. Dr. Hans-Werner Six

Web-Programmierung

Kurseinheiten 1-7

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Studierhinweise

Diese Studierhinweise gliedern sich in die Teile Studiengänge und Vorkenntnisse, Motivation und Einordnung, Gliederung, Sprachgebrauch sowie einen Hinweis zur Bearbeitung.

Studiengänge und Vorkenntnisse

Der Kurs ist für folgende *Studiengänge* vorgesehen: Master Informatik und Master Praktische Informatik, sowie Bachelor Informatik (über Katalog M).

Empfohlene Vorkenntnisse für die erfolgreiche Bearbeitung des Kurses sind vertiefte Kenntnisse in Software Engineering (z.B. Kurs 01793 – „Software Engineering I“) und Erfahrung in der Entwicklung mit der Programmiersprache Java (z.B. Kurs 01618 – „Einführung in die objektorientierte Programmierung“). Konzepte und Techniken aus diesen beiden Bereichen werden in diesem Kurs nicht näher erläutert.

Motivation und Einordnung

Wegen der immer größeren Verbreitung von Web-Anwendungen liegt es nahe, neben dem Kurs 01793 „Software Engineering I“, der sich mit der methodischen Entwicklung von objektorientierten Desktop-Anwendungen beschäftigt, einen Kurs über die Entwicklung von Web-Anwendungen anzubieten. Dabei werden Java-basierte Web-Anwendungen betrachtet, die auf der Java Platform Enterprise Edition (Java EE) in der Version 7 basieren, da diese für große Anwendungen besonders geeignet sind und gute Voraussetzungen für eine an software-technischen Grundsätzen orientierte Entwicklung bieten.

Gliederung

Der Kurs besteht aus *sieben Kurseinheiten*:

Kurseinheit 1	Basistechnologien für Web-Anwendungen
Kurseinheit 2	Sprachen, Technologien, Medien und Anwendungen
Kurseinheit 3	Java EE: Servlet und JavaServer Page
Kurseinheit 4	Softwarearchitekturmuster und Softwarearchitekturen für Web-Anwendungen
Kurseinheit 5	Web-Framework JavaServer Faces
Kurseinheit 6	Java EE: Enterprise JavaBeans
Kurseinheit 7	Java EE: Entity-Klassen und Entwurfsmuster

Zum Sprachgebrauch

Beim Korrekturlesen dieses Ihnen vorliegenden Textes bin ich selbst über die vielen Anglizismen gestolpert; manche Sätze zwingen fast ausschließlich englische Wörter in eine deutsche Syntax. Die Frage ist nun, was gehört ins Deutsche übersetzt, und was bleibt als technischer Ausdruck besser englisch? Einige ursprünglich englische Wörter kann man als eingedeutscht ansehen, dazu gehören zum Beispiel Software, Internet oder auch das Web. Andere Begriffe haben in der Fach-Community (wieder so ein Anglizismus!) eine merkwürdige Wanderung durchgemacht. So unterscheidet man Anwendungsprogramme und Systemprogramme. Aus Anwendungsprogrammen wurden Anwendungen oder sogar Applikationen – als Übersetzung des englischen „application programs“. Nun sind „Anwendungen“ und „Applikationen“ als Worte im Deutschen aber bereits anders belegt, man denkt eher an Krankengymnastik bzw. an Stickereien. Deshalb habe ich versucht, derartig missverständliche Wortbildungen zu vermeiden. An anderen Stellen geht es um Request, Response, Header usw. Dies alles ließe sich zwar auch auf Deutsch formulieren, aber der genau definierte technische Zusammenhang ginge dadurch verloren. Aus diesem Grunde habe ich immer dort die englischen Ausdrücke belassen, wo es um ein zuvor (oder kurz danach) definiertes technisches Konzept geht. Ich denke, dadurch wird die Verständlichkeit sogar eher größer. Außerdem macht es ja wenig Sinn eine Sprache vorzugeben, die in Fachkreisen völlig unüblich ist, und Sie bei Gesprächen in der beruflichen Praxis dazu zwingt, alles wieder zurück zu übersetzen.

Mir ist bewusst, dass im Zusammenhang mit Gender Mainstreaming (... und dazu fällt mir beim besten Willen keine gelungene deutsche Übersetzung ein) maskuline und feminine Personenbezeichnungen vermieden werden sollen und alles geschlechter-

gerecht formuliert werden sollte. Übliche Vorschläge dazu sind die Verwendung von „Studierenden“ statt „Studenten“ oder eben die Aufzählung beider Formen „Professoren und Professorinnen“. Die so gestalteten Prüfungsordnungen und andere Dokumente sind zwar politically correct, aber kaum mehr lesbar. Da die Lesbarkeit bei Kurstexten aber einen höheren Stellenwert hat, habe ich hier auf strikt geschlechtergerechte Sprache verzichtet. Mein Kompromiss ist die Verwendung des generischen Maskulinums im Plural (Ärzte können Frauen und Männer sein) und des generischen Femininums im Singular.

Hinweis

Es wird dringend empfohlen, die Übungsaufgaben zu bearbeiten, da nur durch eine *aktive* Auseinandersetzung mit dem Kursmaterial ein gefestigtes, dem Stoff angemessenes Verständnis erreicht werden kann. Die aktive Auseinandersetzung ermöglicht Ihnen zusätzlich das Gelernte in die Praxis umzusetzen und hilft Ihnen die mündliche Prüfung erfolgreich zu absolvieren.

Diese Seite bleibt aus technischen Gründen frei!

Inhaltsverzeichnis

Diese Seite bleibt aus technischen Gründen frei!

Kurseinheit 1

Basistechnologien für Web-Anwendungen

Kapitel 1 beginnt mit einer Motivation der Kursthematik und einem Überblick über den Kurs. In Kapitel 2 werden die für den Kurs benötigten Basistechnologien eingeführt. Diese umfassen im Wesentlichen die Begriffe Web-Anwendung, Web-Server, Client-Server-Architektur, URI und URL, die Protokolle IP, TCP (UDP) und HTTP, die Dokumentenformate HTML, XHTML und XML, statischer, dynamischer und dynamisch erzeugter Inhalt sowie das Session-Konzept. Ohne die Beherrschung dieser Basistechnologien ist die nachfolgende Kursmaterie kaum zu verstehen.

Diese Seite bleibt aus technischen Gründen frei!

1. Motivation und Überblick

1.1. Motivation

Software Engineering befasst sich mit der systematischen Entwicklung großer Softwaresysteme mit dem Ziel, Software hoher Qualität, innerhalb eines vorgegebenen Budgetrahmens und zum geplanten Zeitpunkt zu produzieren. Die Softwaresysteme können dabei ganz unterschiedlicher Art sein und für ganz unterschiedliche Anwendungsbereiche erstellt werden. Wegen der immer größeren Verbreitung und Akzeptanz des Internets sind *Web-Anwendungen* in Privathaushalten und Industrie inzwischen nicht mehr wegzudenken. Es liegt daher nahe, nach dem Kurs 01793 „Software Engineering I“, der sich mit der methodischen Entwicklung von objektorientierten Desktop-Anwendungen beschäftigt, einen entsprechenden Kurs über die Entwicklung von Web-Anwendungen anzubieten.

Bei einer Web-Anwendung interagiert eine Benutzerin über einen Browser mit einem Anwendungsprogramm, das in der Regel auf einem entfernten Computer läuft, ohne dieses jemals bei sich installieren zu müssen. Dieses Anwendungsprogramm wird – wie die Hardware – Server genannt. Unter einem Client versteht man je nach Kontext den Computer, der über das Internet mit einem Server verbunden ist, oder auch das Anwendungsprogramm, das auf einem solchen Computer läuft. Wir alle verwenden Web-Anwendungen z.B. für die Fahrplanauskunft, um uns über das Kinoprogramm zu informieren oder um Konzertkarten zu bestellen. Über einen Firmen-Server, der so gut wie immer zur Verfügung steht, ist eine Firma für ihre Kunden rund um die Uhr und von überall auf der Welt aus erreichbar und kann z.B. Bestellungen ohne dafür eingesetztes Personal entgegennehmen. Vorteilhaft ist dabei, dass Software nur einmal zentral auf dem Firmen-Server eingespielt werden muss und dann für alle Benutzer zur Verfügung steht. Für die Kunden ergeben sich Vorteile durch die bequeme Bedienung des Programms, aber auch dadurch, dass sie außer der Standardausstattung ihrer Computer keinerlei zusätzliche Software benötigen. Wegen dieser und weiterer Gründe haben sich Web-Anwendungen auf breiter Front durchgesetzt.

1.2. Überblick

Gegenstand des Kurses ist die methodische Entwicklung von Web-Anwendungen. Wir fokussieren auf Java EE-basierte Web-Anwendungen. *Java EE* steht für *Java Platform Enterprise Edi-*

tion, eine von der Firma Sun Microsystems[1] herausgegebenen Spezifikation für eine Standardarchitektur für Java-basierte Anwendungen [JEE/Spec], [JEE/Tut]. In diesem Kurs beziehen wir uns auf Version 7 der Java EE Spezifikation[2]. Ältere Versionen der Plattform[3] waren noch als *Java 2 Platform, Enterprise Edition (J2EE)* bekannt.

Java EE umfasst zum einen *Enterprise JavaBeans (EJBs)* und *Entities*, mit denen der Kern[4] eines Anwendungsprogramms implementiert wird. Einen weiteren wesentlichen Teil stellen die *Web-Komponenten* dar, mit denen die *Web-Schicht* – das ist der Server-seitige Teil der Benutzungsschnittstelle – realisiert wird. Unter einer Web-Komponente versteht man entweder ein Servlet oder eine Web-Seite, das/die mit Hilfe der JSP-Technologie (*JavaServer Pages*) oder der JSF-Technologie (*JavaServer Faces*) erstellt wurde. Ein Servlet steuert die Abarbeitung der durch den Browser übermittelten Client-Anfrage, wobei gewöhnlich der Anwendungskern aufgerufen wird und schließlich das gewünschte Ergebnis liefert. Eine JavaServer Page (JSP) bereitet das Ergebnis Browser-gerecht auf, so dass es nach Übertragung der Benutzerin auf dem Bildschirm angezeigt werden kann. JSF (JavaServer Faces) basiert auf Servlets und stellt zusätzliche Funktionalitäten zur Verfügung.

In *Kurseinheit 1* werden im Anschluss an diesen Überblick zunächst grundlegende Konzepte wie z.B. Web-Anwendung, Web-Server, Client-Server-Architektur, URI und URL, die Protokolle IP, UDP, TCP und HTTP, die Dokumentenformate HTML, XHTML und XML, statischer, dynamischer und dynamisch erzeugter Inhalt sowie das Session-Konzept erläutert.

Um eine ganzheitliche Perspektive auf das Web-Umfeld zu schaffen, werden in *Kurseinheit 2* die gebräuchlichsten Programmiersprachen und Technologien, wie Cascading Style Sheets (CSS), JavaScript, PHP, ASP.NET, Ajax, jQuery und HTML5 kurz erläutert. Darüber hinaus wird auf verschiedene Bildformate, wie JPEG, PNG, GIF, BMP eingegangen und die grundlegende Funktionsweise von Browsern beleuchtet.

In der *Kurseinheit 3* wird zunächst ein Überblick über die Java EE-Spezifikation gegeben. Die Java EE-Webkomponenten *Servlet* und die Technologie JavaServer Pages werden näher ausgeführt. Hiernach werden in *Kurseinheit 4* verschiedene Softwarearchitekturen für Web-Anwendungen diskutiert, darunter Schichtenarchitekturen sowie die Model-1- und Model-2-Architecture. Die *Kurseinheit 5* führt die Leserin in das Web-Framework *JavaServer Faces* ein und beschreibt, wie mit JSF Benutzungsoberflächen erzeugt werden können. Die *Kurseinheit 6* behandelt mit den *Enterprise Java Beans (EJBs)* die ersten für den Anwendungskern relevanten Komponenten. Die *Kurseinheit 7* schließt den Kurs mit der Vorstellung von Entity-Klassen als Teil des Anwendungskerns ab und erwähnt einige nützliche Entwurfsmuster.

[1] Die Firma Sun Microsystems wurde im Jahr 2010 von der Firma Oracle übernommen

[2] Die Version 7 der Java EE Spezifikation wurde im Mai 2013 veröffentlicht

[3] Vor Version 5

[4] Der Anwendungskern ist der zentrale Teil eines Anwendungsprogramms ohne Benutzungsschnittstelle und ohne persistente Datenhaltung

2. Basistechnologien

Aus technischer Sicht besteht das *World Wide Web* (kurz *Web* oder *WWW*) aus weltweit verteilten *Servern*, auf die ebenfalls verteilte *Clients* zugreifen können. Die Kommunikation zwischen den Beteiligten erfolgt über das Internet. Die räumliche Verteilung erfordert effiziente Protokolle und robuste Verfahren zum Datenaustausch. In diesem Kapitel werden wir einige wichtige Begriffe und Konzepte klären sowie die zugrunde liegenden Techniken einführen.

2.1. Web-Anwendung und Web-Server

Da es in diesem Kurs um die Entwicklung von Web-Anwendungen geht, stellt sich zunächst die Frage, was eine Web-Anwendung eigentlich ist. Wir bezeichnen als *Web-Anwendung* eine Client-Server-Anwendung, bei der zur Kommunikation zwischen Client und Server das *Hypertext Transfer Protocol (HTTP)* eingesetzt wird (s. Abschnitt 2.3.2) und der Client ein (HTML-)-Browser ist. Beispiele typischer *Browser* sind Internet-Explorer, Chrome, Firefox, Safari oder Opera. Das gebräuchlichste Format, in dem Text- und Hypertext-Informationen im Web gespeichert und übertragen werden, ist durch die *Hypertext Markup Language (HTML)* gegeben (s. Abschnitt 2.4). Soweit möglich passt der Browser die Darstellung eines *HTML-Dokuments* an die Größenverhältnisse des jeweiligen Bildschirms an. Häufig werden die Begriffe HTML-Dokument und *Web-Seite* synonym verwendet.

Web-Anwendungen sind also Client-Server-Anwendungen. *Client* und *Server* bezeichnen hier Software-Programme. Zwar ist es möglich, dass Client und Server auf demselben Rechner laufen, in der Regel werden sie jedoch auf unterschiedlichen Rechnern ausgeführt, die über ein Netzwerk miteinander verbunden sind. Jeder Rechner in einem Netzwerk wird als *Host* bezeichnet, d.h. Client und Server laufen jeweils auf einem Host. Mit Host wird in der Literatur bisweilen aber auch nur der Rechner bezeichnet, auf dem der Server läuft. Diese Begriffsverwendung kommt aus dem Bereich der Terminal-Host-Anwendungen.

Bei Web-Anwendungen ist der Client für die Oberfläche der *Benutzungsschnittstelle* zuständig, stellt Anfragen an den Server und nimmt dessen Antworten entgegen. Der Server übernimmt den gesamten Rest der Web-Anwendung, d.h. die restliche *Benutzungsschnittstelle*, den *Anwendungskern* und die *Datenhaltung*.

Web-Anwendungen haben im Vergleich zu Desktop-Anwendungen (auch GUI-Anwendungen genannt) besondere Spezifika, von denen einige nachfolgend genannt werden:

- *Gestaltungsmöglichkeiten*: Die Gestaltungsmöglichkeiten der Benutzungsoberfläche von Web-Anwendungen sind gewöhnlich durch HTML (s. Abschnitt 2.4) vorgegeben und wesentlich geringer als bei Desktop-Anwendungen. Sie können jedoch durch Verwendung von CSS und JavaScript (s. Kurseinheit 2) verbessert werden und dann Funktionen und Aussehen bieten, wie man es von einer Desktop-Anwendung gewohnt ist.
- *Fenster*: Bei Web-Anwendungen steht nur ein Fenster zur Verfügung. Mehrere übereinander liegende oder nebeneinander angeordnete Fenster wie bei Desktop-Anwendungen sind gar nicht oder nur mit Einschränkungen zu realisieren.[1]
- *Zulässige Programmabläufe*: Das Sicherstellen zulässiger Programmabläufe gestaltet sich bei Web-Anwendungen insofern schwieriger, als die Benutzer über die Angabe einer URL (s. Abschnitt 2.2) grundsätzlich jede beliebige Web-Seite ansteuern und damit letztlich auch an beliebige Positionen im Programmablauf springen können. Darüber hinaus ist es mittels des Zurück-Buttons des Browsers möglich, beliebige Rücksprünge durchzuführen.
- *Kommunikationsaufwand*: Bei Web-Anwendungen entsteht ein erhöhter Kommunikationsaufwand, da alle Anfragen und Antworten zwischen Client und Server übertragen werden.
- *Installationsaufwand und Wartung*: Die Web-Anwendung wird nur auf dem Server installiert und gewartet. Auf der Client-Plattform muss außer einem Browser keine weitere Software installiert werden. (Client und Server bezeichnen in diesem Kontext Hardware.)
- *Portabilität*: Bei Web-Anwendungen ist die Portabilität (Übertragbarkeit auf unterschiedliche Plattformen) des Anwendungsprogramms unproblematisch, da nur der Client (Browser) für verschiedene Plattformen existieren muss. Das ist praktisch immer gegeben.
- *Verfügbarkeit*: Da nur eine Internet-Verbindung und ein Browser benötigt werden, kann von überall dort, wo diese beiden Voraussetzungen erfüllt sind, zu jeder Zeit die Web-Anwendung genutzt werden. Dasselbe gilt für Web-Anwendungen, die über ein Intranet einem eingeschränkten Benutzerkreis zur Verfügung gestellt werden.

2.1.1. Thin Client und Rich Client

Ist der Client einer Web-Anwendung lediglich ein Browser, spricht man von einem *Thin Client*. Ein Thin Client stellt nur die Benutzungsoberfläche dar (hier meist aus HMTL-Dokumenten, s. Abschnitt 2.4), richtet Anfragen an den Server und nimmt dessen Antworten entgegen. Die Verarbeitung der Daten erfolgt bei Thin Clients ausschließlich auf dem Server. Im Gegensatz dazu realisiert ein *Rich Client* über die Benutzungsoberfläche hinaus weitere Teile der Web-Anwendung, meist z.B. die gesamte Benutzungsschnittstelle.

[1] Innerhalb eines Browserfensters ist es jedoch unter der Hinzunahme von JavaScript möglich, mehrere Fenster zu simulieren.

2.1.2. Code on Demand

Gegenüber Desktop-Anwendungen ist das Spektrum möglicher Benutzerinteraktionen bei Web-Anwendungen zunächst beschränkt. Um diese Beschränkungen teilweise aufzuheben, ermöglichen es die meisten Browser, Programme oder einzelne Funktionen vom Server zu laden und Client-seitig auszuführen. Die bekanntesten Varianten dieser *Code on Demand (COD)* genannten Technik sind *Java Applets* oder *JavaScript-Anweisungen*. Auf JavaScript wird in Kurseinheit 2 eingegangen, Java Applets werden im Rahmen des Kurses nicht weiter betrachtet.

2.1.3. Web-Server

Als *Web-Server* bezeichnet man Server, die im Zusammenhang mit Web-Anwendungen verwendet werden, also HTTP unterstützen. Einfache Web-Server liefern als Dateien abgelegte HTML-Dokumente an die Clients aus. Zu den bekanntesten Web-Servern dieser Art gehören der *Apache HTTPD Web Server* ([HTTPD]) und *Microsoft Internet Information Services* ([IIS]). Neben einfachen Dateien können Web-Server auch algorithmische Anweisungen enthalten, die in unterschiedlichen Programmiersprachen verfasst sein können. Dafür werden entweder modulare Erweiterungen oder aber spezielle Web-Server verwendet.

2.2. URI und URL

Bei einer Web-Anwendung fordern die Clients Daten vom Server an. Dazu müssen sie in der Lage sein, die angeforderten Daten auf dem Server zu adressieren. Dies geschieht über Uniform Resource Identifier, kurz URI. Eine URI ist ein weltweit gültiger und eindeutiger Bezeichner für eine Ressource. Eine Ressource ist alles, was über eine URI identifiziert werden kann. Benutzer nehmen Ressourcen z.B. als Web-Seiten wahr, die von einem Web-Server bereitgestellt werden. Genau genommen ist eine Web-Seite die visuelle Darstellung von einer auf einem Server liegenden Ressource. Ressourcen können in unterschiedlicher Form auftreten, als Datei oder als eine Programmkomponente wie z.B. ein Servlet (s. Kurseinheit 3).

Uniform Resource Locator (URL) und Uniform Resource Name (URN) sind die beiden Varianten einer URI. Eine URL identifiziert eine Ressource eindeutig über die Ortsangabe und wird z.B. bei dem Protokoll HTTP verwendet; sie wird daher häufig auch mit Internet-Adresse übersetzt. Eine URN identifiziert eine Ressource über den Namen und kommt bei HTTP nicht zum Einsatz.

Der Aufbau einer URI ist protokollabhängig, man spricht von dem Schema der URI:

```
URI = scheme : <scheme-specific-part>
```

Zwei URI-Beispiele zur Erläuterung von Schemas:

URI gemäß HTTP-Schema:

`http://www.fernuni-hagen.de` (s. Abschnitt 2.3.2)

URI gemäß mailto-Schema:

`mailto:Kurs1796@FernUni-Hagen.de?subject=Studientag`

2.3. Protokolle IP, TCP (UDP) und HTTP

Die Kommunikation zwischen Client und Server verläuft stets nach bestimmten Regeln, die in einem Protokoll definiert sind. Diese Protokolle zu kennen ist auch deshalb wichtig, weil ihre Eigenschaften (vor allem die von HTTP) die Möglichkeiten von Web-Anwendungen einschränken. Man unterscheidet die Protokolle IP, TCP (UDP) und HTTP, die jeweils einer von drei Schichten zugeordnet sind:

Schicht	**Protokoll**
Anwendungsschicht	HTTP
Transportschicht	TCP (UDP)
Vermittlungsschicht	IP

Tabelle 2.1.: Schichten und ihre Protokolle

Die in der Tabelle dargestellten Schichten sind Teil des ISO/OSI-Referenzmodells, das insgesamt sieben Schichten definiert und ein Referenzmodell für Netzwerkprotokolle darstellt. Diese Schichten sind:

1. Bitübertragung
2. Sicherungsschicht
3. Vermittlungsschicht
4. Transportschicht
5. Sitzungsschicht
6. Darstellungsschicht
7. Anwendungsschicht

Sendet ein Server Daten zu einem Client, so werden diese Daten entlang der Schichten bei dem Server von unten nach oben (7, 6, . . . , 1) transformiert und bei dem Client von oben nach unten (1, 2, . . . , 7).

2.3.1. IP und TCP (UDP)

Das *Internet Protocol (IP)*, das zur *Vermittlungsschicht* gehört, und das *Transmission Control Protocol (TCP)*, das zur *Transportschicht* gehört, sorgen gemeinsam dafür, dass Daten zuverlässig von einem Rechner zum anderen transportiert werden. Dazu wird jedem Rechner beim Verbindungsaufbau (gemeint ist die Verbindung mit dem Internet) dynamisch eine eindeutige Adresse, die sogenannte *IP-Adresse*, zugewiesen. Die dynamische Vergabe einer IP-Adresse sorgt dafür, dass im Durchschnitt weniger als eine IP-Adresse für einen Client benötigt wird, da nie alle möglichen Clients zur gleichen Zeit online sind. Es gibt aber auch Rechner, die eine feste IP-Adresse besitzen. Da es relativ unkomfortabel ist, mit den numerischen IP-Adressen zu arbeiten, werden über einen speziellen Dienst, das *Domain Name System (DNS)*, den IP-Adressen Namen zugeordnet. Im Fall der FernUniversität entspricht der (festen) IP-Nummer „132.176.114.181“ der Name „www.fernuni-hagen.de“. Zurzeit (Stand 2013) sind zwei IP-Versionen aktuell, IPv4 und IPv6. Der Unterschied dieser beiden Versionen besteht in dem darstellbaren Adressraum. IPv4 verwendet 4*8 Bits = 32 Bits, um eine Adresse darzustellen. 8 Bits stehen in der Dezimalschreibweise für das Zahlenintervall 0 bis 255. Eine 32 Bit-Adresse besteht daher aus 4 Zahlen im Intervall 0 bis 255, wobei die vier Zahlen jeweils durch einen Punkt getrennt sind. Somit sind mit IPv4 theoretisch insgesamt 4.294.967.296 Adressen darstellbar. Da sich das Internet stets vergrößert und immer mehr IP-Adresse benötigt wurden, verringerte sich die Anzahl der noch freien IPv4-Adressen. Besonders aus diesem Grund wurde der Nachfolger IPv6 entwickelt, der nicht mit 32 Bit, sondern mit 128 Bit arbeitet. IPv6-Adressen werden zur besseren Übersicht nicht mehr in Dezimalschreibweise notiert, sondern hexadezimal[2] XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX:XXXX (8 Blöcke, jeder Block enhält eine vierstellige Hexadezimalzahl, die das Intervall 0 bis 65535 darstellen kann). Diese Version des Internet Protokolls kann ca. $3,4 \cdot 10^{38}$ Adressen darstellen.

Für die unterschiedlichen unter einer IP-Adresse angebotenen Dienste eines Servers wie z.B. E-Mail, Streaming etc. reicht die IP-Adresse zur Adressierung alleine nicht aus, sondern es muss auch der gewünschte Dienst bei der Anfrage mit angeben werden. Dazu sind im Protokoll TCP sogenannte *Ports* vorgesehen, die über Zahlen zwischen 0 und 65536 identifiziert werden. Einige Ports werden standardmäßig für spezielle Dienste verwendet, dies ist z.B. E-Mail, FTP, Telnet oder Internet News der Fall. So ist für HTTP (siehe Abschnitt 2.3.2) beispielsweise Port 80 vorgesehen. Beim Verbindungsaufbau muss der Client auf seinem Host eine beliebige, freie Portnummer wählen, damit das TCP zwischen diesem Client-Port und dem Server-Port eine Verbindung aufbauen kann.

Das Protokoll IP ist paketorientiert, d.h. Daten werden paketweise übertragen. Dabei arbeitet es unzuverlässig, d.h. es überprüft nicht, ob alle Pakete korrekt beim Empfänger angekommen sind. Zudem ist es verbindungslos: Der Empfänger hat keine Informationen über den Sender, kann also bei Schwierigkeiten keinen Kontakt mit dem Sender aufnehmen. Das TCP behebt beide Probleme. Durch die Ports und die IP-Adressen kann es eine (beidseitige) Verbindung herstellen. Die Korrektheit der Daten wird überprüft und im Fehlerfall werden Pakete erneut angefordert. Es ist somit ein verbindungsorientiertes Protokoll.

[2] Also mit 16 Ziffern: 0,...,9,A,...,F

Bei dem User Datagram Protocol (UDP), das wie TCP auf der Verbindungsschicht agiert, handelt es sich um ein verbindungsloses Protokoll. Es wird keine Fehlerkontrolle durchgeführt und Pakete, die verloren gegangen sind, werden nicht erneut gesendet. Durch den geringeren Verwaltungsaufwand im Vergleich zu TCP wird Zeit gespart, was bei zeitkritischen Anwendungen (wie z.B. Video-Streaming oder Sprachübertragung) hilfreich sein kann, wenn die Vollständigkeit der Daten weniger wichtig ist als ihre Rechtzeitigkeit.

2.3.2. HTTP

Bei Web-Anwendungen findet die Kommunikation zwischen Client und Server auf der Anwendungsschicht statt. Eingesetzt wird das *Hypertext Transfer Protocol (HTTP)*, das sich auf TCP/IP abstützt. Es basiert auf dem Austausch von *Nachrichten (Messages)*: Der Client schickt eine *Anfrage (Request)* an den Server und dieser schickt eine entsprechende *Antwort (Response)* zurück. Dieses Verfahren bezeichnet man als eine *Pull-Technologie*, die für alle Client-Server-Anwendungen typisch ist: Der Client initiiert die Kommunikation und „zieht“ eine Antwort vom Server. Der Server kann dabei nur reagieren, d.h. ohne Aufforderung keine Informationen an den Client schicken.

Im Fall von HTTP ist eine URL nach dem *HTTP-Schema* aufgebaut. Ihre Syntax geben wir in Backus-Naur-Form (BNF) an, bei der eckige Klammern optionale Angaben umschließen.

```
1 http_URL    = "http" ":" "//" host [":" port] [abs_path ["?" query]]
2 host     = <IP-Adresse oder DNS-Name des Zielrechners>
3 port     = <Portnummer, standardm\"aßig 80>
4 abs_path = <Pfadangabe der Ressource>
5 query    = <Zeichenkette mit zusätzlichen Parametern>
```

Listing 2.1: Backus-Naur-Form einer URL

Für `host` wird die IP-Adresse oder der Name des Server-Rechners eingesetzt. Ist `port` nicht angegeben, wird als Port-Nummer standardmäßig 80 angenommen. `abs_path` enthält die Angabe, wo die angesprochene Ressource auf dem Server zu finden ist, vergleichbar mit dem Pfad zu einer Datei in einem Dateisystem. Häufig spiegelt `abs_path` tatsächlich die Pfadangabe des Dateisystems auf dem Server wider – was aber nicht der Fall sein muss. Der Query-String `query` enthält Parameterangaben, über die Daten vom Client zum Server gesendet werden können. Die Syntax für Parameterangaben ist

```
Parameter1=Wert1&Parameter2=Wert2&Parameter3=Wert3&...
```

Eine *HTTP-Nachricht*, also ein *Request* oder ein *Response*, ist ein einfacher ASCII-Text. Je nach Übertragungsmethode unterscheidet sich der Aufbau der Nachrichten. Die zwei wichtigsten Übertragungsmethoden sind GET und POST. Darüberhinaus gibt es noch zahlreiche weitere Übertragungsmethoden wie z.B. PUT, HEAD, TRACE und CONNECT, die wir aber hier nicht behandeln.

Bei der *GET-Übertragungsmethode* werden die Client-Daten (als Parameterangaben im Query-

String) in der URL übertragen. Wegen der Browser-abhängigen Längenbeschränkung der URL kann die GET-Übertragungsmethode nur dann eingesetzt werden, wenn der Client nur wenige oder keine Daten sendet und in erster Linie Daten vom Server anfordert.

Typische Beispiele für den Einsatz der GET-Übertragungsmethode sind Requests, bei denen die Benutzerin eine URL (inklusive der Parameterangaben) in der Adresszeile des Browsers eingibt oder einen Link verfolgt. Werden Parameterangaben übertragen, findet sich unter der URL als Ressource in der Regel eine Programmkomponente, ein Servlet (s. Kurseinheit 3), die die Parameterangaben verarbeitet. Die Nutzung der GET-Übertragungsmethode ist jedoch als sicherheitskritisch einzustufen, da die zu übertragenen Daten in der URL direkt ablesbar sind.

Beispiel 2.1:

Wir nehmen an, dass eine Benutzerin die URL `http://localhost:8080/kursauswahl?kursNr=01796&semester=aktuell` (vgl. `http_URL` in List. 2.1) in der Adresszeile des Browsers eingibt. Der Browser erzeugt daraus folgende Anfrage mit der GET-Übertragungsmethode:

```
1 GET /kursauswahl?kursNr=01796&semester=aktuell HTTP/1.1
2 User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:24.0) Gecko
    /20100101 Firefox/24.0
3 Accept: text/html,application/xml,application/xhtml+xml;q=0.9,
    */*;q=0.8
4 Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
5 Accept-Encoding: gzip,deflate
6 Host: localhost:8080
7 Connection: keep-alive
```

Listing 2.2: Request, der mit der GET-Übertragungsmethode abgeschickt wird

Bevor eine Anfrage abgesendet wird, stellt der Browser zunächst eine TCP/IP-Verbindung zum Host auf dem angegebenen Port her (im Beispiel zu `localhost` auf dem Port 8080). Die erste Zeile in List. 2.2 besteht aus dem Namen der Methode, dem absoluten Pfad (`abs_path`) `kursauswahl`, dem Query-String (`query`) mit den Parameterangaben `kursNr=01796&semester=aktuell` und der HTTP-Version `HTTP/1.1`.

Dann folgen – einer pro Zeile – die verschiedenen sogenannten *Header*, die beispielsweise den Browser (`User-Agent`), die akzeptierten Sprachen (`Accept-Language`), hier Deutsch und Englisch, usw. angeben.

Die *POST-Übertragungsmethode* wird insbesondere dann verwendet, wenn der Client umfangreiche Daten, etwa Dateien, an den Server schickt, die mit GET wegen der Browser-abhängigen URL-Längenbeschränkung nicht gesendet werden können. Da die Daten jetzt nicht mehr Teil der URL sind, können diese auch nicht in der Adresszeile des Browsers abgelesen werden. Dies stellt bei Passworteingaben einen weiteren Grund für die Wahl von POST als Übertragungsmethode dar.

Beispiel 2.2:

Der zu dem „GET-Request“ in Beispiel 2.1 analoge „POST-Request“ sieht wie folgt aus:

```
1  POST /kursauswahl HTTP/1.1
2  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; de-DE; rv
       :1.7.8) Gecko/20050511 Firefox/1.0.4
3  Accept: text/xml,application/xml,application/xhtml+xml,text/html;
       q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
4  Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
5  Accept-Encoding: gzip,deflate
6  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
7  Host: localhost:8080
8  Connection: close
9  Content-Length: 30
10
11 kursNr=01796&semester=aktuell
```

Listing 2.3: Request, der mit der POST-Übertragungsmethode abgeschickt wird

Der Unterschied zum „GET-Request“ besteht darin, dass die Parameter nicht Teil der URL sind, sondern, durch eine Leerzeile getrennt, im Anschluss an die Header gesendet werden.

Eine *Response des Servers* auf obigen „GET-Request“ bzw. „POST-Request“ könnte z.B. folgendermaßen aussehen:

```
1  HTTP/1.1 200 OK
2  Date: Mon, 07 Oct 2013 9:00:00 GMT
3  Server: Apache
4  X-Powered-By: PHP/5.4.20
5  Last-Modified: Mon, 05 Oct 2013 13:10:02 GMT
6  Connection: Keep-Alive
7  Transfer-Encoding: chunked
8  Content-Type: text/html
9
10 <html>
11    ...
12 </html>
```

Listing 2.4: Mögliche HTTP-Response des Servers

Die erste Zeile der Response enthält die HTTP-Version, hier `HTTP/1.1`, den Status-Code `200` und die diesen Code erläuternde Reason-Phrase `OK`. Der Status-Code ist in der Definition von HTTP festgelegt: `200` bedeutet hier, dass der Request beantwortet werden konnte und kein Fehler aufgetreten ist. Der bekannteste Fehler-Code ist wohl `404 Not Found` – Ressource nicht gefunden.

Ab der zweiten Zeile bis zur Leerzeile sind die Header der Antwort aufgeführt, pro Zeile ein Header. Content-Type gibt die Art der Daten an, die nach der Leerzeile gesendet werden. Transfer-Encoding beschreibt, wie die Daten übermittelt werden, damit sie sicher beim Client ankommen.

Chunked bedeutet hier „aufgeteilt“, die Daten werden somit in Chunks (Datenpakete) gesendet. Zusätzlich handelt sich um ein HTML-Dokument, wie man an dem `Content-Type` sowie dem öffnenden und schließenden `<html>`-Tag erkennt (s. Abschnitt 2.4).

2.3.3. HTTPS

Ein Nachteil von HTTP ist, dass alle Daten in einfacher Textform übertragen werden. Das macht ihr „Abhören“, etwa mit sogenannten Packet-Sniffern[3] , leicht und stellt in manchen Fällen ein nicht akzeptables Sicherheitsrisiko dar, etwa wenn Passwörter oder andere persönliche Daten übertragen werden.

Um eine sicherere Datenübertragung zu gewährleisten, können Daten auch vor dem Transport verschlüsselt und beim Empfänger wieder entschlüsselt werden. Dafür wird ein *Secure Socket Layer (SSL)* eingesetzt werden; das zugehörige Protokoll heißt *Hypertext Transfer Protocol Secure (HTTPS)*. In der URI macht sich der Einsatz eines SSL durch den Protokollnamen „https“ anstelle von „http“ bemerkbar. SSL ist zwischen HTTP und TCP angesiedelt und für die Entwickler von Web-Anwendungen nicht sichtbar und nicht relevant, sodass wir nicht weiter darauf eingehen. Folgendes Schema zeigt die Anordnung der Protokolle unter Einbeziehung von SSL:

Schicht	Protokoll
Anwendungsschicht	HTTP
Transportschicht	SSL
	TCP (UDP)
Vermittlungsschicht	IP

Tabelle 2.2.: Schichten und ihre Protokolle (mit SSL)

2.4. Dokumentenformate HTML und XML

Bisher wurde erläutert, wie Client und Server über Anfragen und Antworten auf Grundlage des Protokolls HTTP miteinander kommunizieren und wie die eigentlichen Inhalte in Anfragen und Antworten identifiziert werden. Über diese Inhalte selbst und insbesondere über das Format, in dem sie übertragen werden, wurde bisher kaum etwas gesagt. Das soll in diesem Abschnitt nachgeholt werden.

Für die Definition der Inhalts- und Medientypen sowie für die Codierung der Daten, also etwa die Umwandlung von binären Daten zur Repräsentation von Bildern, in ASCII-Zeichen, wird die *Multipurpose Internet Mail Extension (MIME)* eingesetzt.

[3] Unter einem *Sniffer* versteht man ein Programm, das den Datenverkeht eines Netzwerkes untersuchen kann (z.B. zur Diagnose, Analyse, aber auch zur Datenspionage). Packet ist englisch für Paket.

In der Response, die in List. 2.4 dargestellt ist, findet sich der Header `Content-Type`, mit dem der Server beschreibt, welches Format der Inhalt im Datenbereich der Antwort besitzt. In diesem Fall ist es der gebräuchliche Typ „*text/html*“, der Dokumente im HTML-Format bezeichnet.

2.4.1. HTML

HTML (vgl. [HTML]) steht für *Hypertext Markup Language* (auch *Hypertext-Auszeichnungssprache* genannt). Hypertexte sind im Fall von HTML formatierte Texte, also Texte mit ausgezeichneten Überschriften, Absätzen und anderen Formaten. Das Besondere an Hypertexten ist ihre Fähigkeit, über sogenannte *Links (Referenzen)* auf andere Texte oder Dateien bzw. auf andere Stellen in demselben Text zu verweisen. Zur Kennzeichnung der Formateigenschaften und der Referenzen werden *Markierungen (Markups)* eingesetzt, auch *Tags* (englisch für Etikett oder Kennzeichnung) genannt. Sie haben in der Regel eine Anfangs- und eine Endkennung, die den zu kennzeichnenden Text umfassen: `<sometag>...</sometag>`.

Jedes HTML-Dokument wird vom Tag `<html>` eingeschlossen und besteht aus zwei Elementen: der *Kopfteil* wird vom Tag `<head>`, der Rumpf vom Tag `<body>` umschlossen. Der Kopfteil enthält allgemeine Informationen über das Dokument, wie z.B. seinen Titel oder die Zeichenkodierung. Im Allgemeinen ist der Rumpf das wesentliche Element; er enthält den eigentlichen Text des Dokuments.

List. 2.5 zeigt ein einfaches HTML-Dokument. Im Kopfteil steht lediglich der Titel (`<title>`). Der Rumpf besteht aus einer Überschrift (`<h1>`) und einem Absatz (`<p>`). Die Elemente sind zur besseren Lesbarkeit ihrer Hierarchie entsprechend eingerückt.

Die im Beispiel verwendete Einrückung hat keine Auswirkung auf die Bildschirmdarstellung, sondern dient lediglich der besseren Lesbarkeit des Dokuments. Bei HTML-Tags wird auch nicht zwischen Groß- und Kleinschreibung unterschieden. Weiterhin gibt es einen HTML-Kommentar, der durch `<!--` und `-->` eingerahmt ist. Ein HTML-Kommentar wird zur Dokumentation des eigentlichen HTML-Textes eingesetzt. Er wird nicht vom Browser angezeigt.

```
1  <html>
2    <head>
3      <title>Ein kleines Beispiel</title>
4    </head>
5    <!-- es folgt der Rumpf -->
6    <body>
7      <h1>Überschrift der Ebene 1</h1>
8      <p>Hello World!</p>
9    </body>
10 </html>
```

Listing 2.5: Beispiel eines HTML-Dokuments

Zur *Textgestaltung* sind in HTML u.a. folgende Tags definiert:

Im Text können Sonderzeichen mit Platzhaltern, so genannten *Entities*, umschrieben werden.

Insbesondere im Fall des Kleiner-Zeichens („<“) ist dies zwingend notwendig, da es in der Syntax von HTML zur Kennzeichnung der Tags dient. Entities werden einleitend mit „&“ und abschließend mit „ ; „ gekennzeichnet. Deswegen muss auch das „&“ selbst durch ein Entity ersetzt werden. Die wichtigsten Entities sind in folgender Tabelle aufgeführt:

Links in HTML verwenden URIs zur Referenzierung. Beginnt die URI mit der Angabe des Protokolls (meistens „http://“) oder mit „/“, spricht man von einem *absoluten Link*. Ist dies nicht der Fall, liegt ein *relativer Link* vor, d.h. der Link wird relativ zum aktuellen Dokument interpretiert. Ein Verweis kann im referenzierenden Dokument über einen *Anker* `<a>` markiert werden, die referenzierte URI wird über ein zugehöriges Attribut `href` angegeben. Neben dem `<a>`-Tag gibt es noch weitere Tags, die als Verweis funktionieren. So kann mit dem `<img>`-Tag z.B. auf eine Bilddatei verwiesen werden.

Beispiel 2.3:

Der folgende Auszug aus einem HTML-Dokument enthält einen absoluten Verweis auf die Homepage der FernUniversität sowie zwei relative Verweise.

```
1 ...
2 <p>
3   Besuchen Sie die
4   <a href="http://www.fernuni-hagen.de">FernUniversit&auml;t</a>
5   im Internet<br/>
6   <a href="toc.html">Inhaltsverzeichnis</a><br>
7   <a href="glossar.html">Glossar</a>
8 </p>
9 ...
```

Listing 2.6: Verweise in einem HTML-Dokument

Angenommen, obiger Beispieltext hätte die URI

`http://host/some/path/beispiel.html`,

Element	Bedeutung
`h1, h2, ..., h6`	Überschriften verschiedener Ebenen
`p`	Einfacher Absatz
`b, i, u`	Zeichenformatierungsbefehle: fett, kursiv, unterstrichen[4]
`table, tr, td`	Tabellen; eine Tabelle besteht aus Zeilen (`tr`), die wiederum Zellen (`td`) enthalten
`hr`	Horizontale Linie (ohne End-Tag)
`br`	einfacher Zeilenumbruch (ohne End-Tag)

Tabelle 2.3.: HTML-Elemente zur Textgestaltung

Entity	Zeichen	Erklärung
`<`	<	„lt“ steht fuer „less than“
`>`	>	„gt“ steht fuer „greater than“
`&`	&	„amp“ steht fuer „ampersand“
`&quot;`	“	„quot“ steht für „quote“
`&auml;` oder `&Auml;`	ä Ä	„auml“ steht fuer a-Umlaut
`&uuml;` oder `&Uuml;`	ü Ü	„uuml“ steht fuer u-Umlaut
`&ouml;` oder `&Ouml;`	ö Ö	„ouml“ steht fuer o-Umlaut

Tabelle 2.4.: Entities und ihre Darstellung

so würden die relativen Verweise vom Browser übersetzt in die absoluten Verweise

`http://host/some/path/toc.html`

und

`http://host/some/path/glossar.html`.

Auch Bilder lassen sich, ähnlich wie Verweise, in ein HTML-Dokument integrieren. Sie werden über das Tag `<img>` eingebunden. Die Bilddaten selbst sind nicht im HTML-Dokument enthalten, stattdessen wird mit Hilfe des Attributs src über die URI auf das Bild verwiesen, z.B.

```
<p>Ein Bild: <img src="img/sample.gif"alt="Ein Beispielbild"></p>
```

Das Attribut `alt` sollte immer angegeben werden. Es beinhaltet den Text, der angezeigt wird, sollte das referenzierte Bild im angegebenen Pfad nicht gefunden werden.

Zur Erfassung von Benutzereingaben werden Formulare verwendet. Tabelle 2.5 zeigt die wichtigsten Elemente zur Erstellung von Formularen.

Alle Formularelemente außer dem `<form>`-Tag selbst müssen innerhalb eines `<form>`-Tags untergebracht werden. In List. 2.7 ist der HTML-Quelltext für ein Beispieldokument gegeben, das im Rumpf nur ein Formular enthält.

Tag	Beschreibung und wichtige Attribute
`form`	Wichtige Attribute bei der Definition des Formulars: `action`: URI der Ressource (hier eine Programmkomponente), an welche die Daten (zwecks Verarbeitung) gesendet werden `method`: HTTP-Übetragungsmethode, z.B. GET oder POST
`input`	Eingabefelder des Formulars; über Attribute wird der Typ der Eingabe gewählt. Wichtigste Attribute: `name`: Name des Feldes. Dieser Name dient als Parametername beim Senden des Formulars. `value`: Vorbelegung bei Eingabefeldern, Wert des Feldes bei Auswahlelementen und Beschriftung bei Buttons `type`: Typ des Feldes: `text`, `password`, `radio`, `hidden`, `check`, `submit` oder `reset`. Wenn der Typ `submit` ist, wird ein Button angezeigt, bei dessen Aktivierung die Daten an die im `form`-Tag definierte `action` übertragen werden. Mit `reset` wird ebenfalls ein Button erzeugt, bei dessen Aktivierung die Daten auf die Vorgabewerte zurückgestellt werden.
`textarea`	Mehrzeiliges Eingabefeld für Text. Der von diesem Tag eingerahmte Text wird als Inhalt des `textarea`-Elements dargestellt. `name`: Name des Feldes. Dieser Name dient als Parametername beim Senden des Formulars.
`select`, `option`	Auswahlliste; die wählbaren Optionen müssen über `option` angegeben werden. Das wichtigste Attribut von `select`: `name`: Name des Feldes. Dieser Name dient als Parametername beim Senden des Formulars.

Tabelle 2.5.: Wichtige Elemente zur Erstellung von Formularen

Beispiel 2.4:

```
1  <html>
2    <head>
3      <meta http-equiv="content-type" content="text/html; charset=
           iso-8859-1">
4    </head>
5    <body>
6      <form action="http://localhost:8085/BeispielFormular/Request"
           method="post">
7        <p>Name: <input name="name" type="text"></p>
8        <p>Geschlecht:
9          <input name="geschlecht" type="radio" value="w">weiblich
10         <input name="geschlecht" type="radio" value="m">m&auml;
               nnlich
11       </p>
12       <p>Nebenfach
13         <select name="nebenfach">
```

```
14            <option>BWL</option>
15            <option>Mathematik</option>
16            <option>Elektrotechnik</option>
17            <option>sonstige</option>
18          </select>
19        </p>
20        <p>
21          <textarea name="kommentar">Geben Sie zus&auml;tzliche
                Bemerkungen an</textarea>
22        </p>
23        <p>
24          <input name="reset" type="reset" value="zur&uuml;cksetzen"
                >
25          <input name="submit" type="submit" value="abschicken">
26        </p>
27      </form>
28    </body>
29  </html>
```

Listing 2.7: Quelltext eines HTML-Dokuments mit einem Formular

Im Kopfteil des HTML-Dokuments wird als Metaangabe der verwendete Zeichensatz angegeben (Zeile 3), hier `iso-8859-1`. Ansonsten enthält das Dokument nur ein Formular (Zeile 6). Die Angaben in Zeile 6 sagen, dass das Formular an die URL `http://localhost:8085/BeispielFormular/Request` gesendet und als Übertragungsmethode `POST` verwendet wird. Die beiden Radio-Buttons (Zeilen 9 und 10) haben denselben Namen, was bedeutet, dass jeweils nur einer von ihnen ausgewählt werden kann. Die select-Angabe (Zeile 13) erzeugt eine Auswahlliste, sie wird in Form einer Dropdown-Liste vom Browser angezeigt. Das Textfeld (Zeile 21) enthält einen vorgegebenen Text, der vom Benutzer überschrieben werden kann. Am Ende des Dokuments finden sich zwei Buttons, einer zum Löschen der Eingabe (Zeile 24) und der zweite zum Senden der Daten des Formulars (Zeile 25).

Abb. 2.1 zeigt einen Screenshot des Formulars, so wie es ein Browser anzeigt.

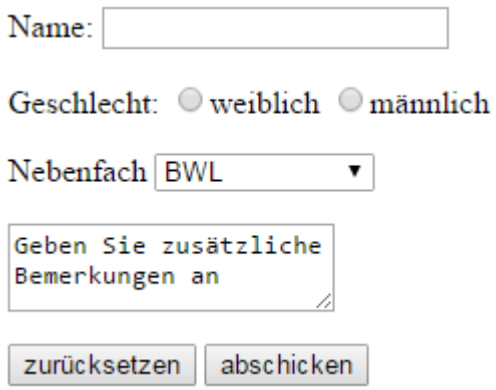

Abbildung 2.1.: Screenshot des Formulars

Löst die Benutzerin den „abschicken“-Button aus, stellt der Browser den zugehörigen

Request zusammen und sendet ihn an den Server. Die Benutzerin sieht davon natürlich nichts. Der Request könnte bei ausgefülltem Formular z.B. so aussehen:

```
1  POST /BeispielFormular/Request HTTP/1.1
2  Host: localhost:8085
3  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:24.0) Gecko
       /20100101 Firefox/24.0
4  Accept: text/xml,application/xml,application/xhtml+xml,text/html;q
       =0.9,text/plain;q=0.8,image/png,*/*;q=0.5
5  Accept-Language: de-de,de;q=0.8,en-us;q=0.5,en;q=0.3
6  Accept-Encoding: gzip,deflate
7  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8  Keep-Alive: 300
9  Connection: keep-alive
10 Referer: http://localhost:8084/BeispielFormular/index.jsp
11 Cookie: JSESSIONID=B4EDA3C1D02EF3D5F2D4DB57CC97149D
12 Content-Type: application/x-www-form-urlencoded
13 Transfer-Encoding: chunked
14
15 name=Mustermann&geschlecht=m&nebenfach=sonstige&kommentar=Geben+
       Sie+zus%E4tzliche+Bemerkungen+an&submit=abschicken
```

Listing 2.8: Beispiel-Request für das in List. 2.7 codierte HTML-Dokument

2.4.2. XHTML

In *XHTML*, einer neueren Version von HTML, wird die Syntax von HTML auf Basis von *XML* (s.u.) definiert. Im Ergebnis sehen sich XHTML-Dokumente und HTML-Dokumente sehr ähnlich. Die wichtigsten Änderungen sind ein anderer Dokumentenkopf und striktere Regeln für die Schreibweise der Tags. So müssen alle Elemente in XHTML klein geschrieben werden, in HTML wurde noch Großschreibung empfohlen.

Für diesen Kurs spielt die Version der Auszeichnungssprache, HTML oder XHTML, eine untergeordnete Rolle. Im Allgemeinen nützt es ohnehin nichts, die Spezifikation einer bestimmten XHTML-Version zu erfüllen, wenn dann die HTML-Browser nicht die gewünschte Ausgabe produzieren. Das ist häufig genug der Fall, und meistens hilft hier nur Ausprobieren, bis es mit allen gewünschten Browsern klappt. Um für die Zukunft gerüstet zu sein, empfehlen wir aber die Verwendung einer an XML bzw. XHTML orientierten Syntax.

2.4.3. XML

Die *Extensible Markup Language (XML)* ist eine Metasprache, die sich besonders zur Beschreibung hierarchischer (Dokument-)Strukturen eignet (vgl. z.B. [Be/Mi]). Mit XML können auch Markup Languages definiert werden. So ist XHTML beispielsweise eine Markup Language, die in XML definiert ist.

Zu den Vorteilen von XML gehören die Trennung von Struktur und Inhalt sowie die Plattformunabhängigkeit. So wird XML häufig eingesetzt, wenn Dokumente zwischen Programmen, die auf verschiedenen Plattformen laufen können, ausgetauscht werden müssen. Da die Sprache XML sowohl für Programme als auch für Menschen (relativ) leicht lesbar ist, wird sie auch für Konfigurationsdateien, gerade im Bereich von Web-Anwendungen, eingesetzt. Daher ist es wichtig, die Grundzüge dieser Sprache zu verstehen, selbst wenn sie später nicht als Dokumentenformat der Web-Anwendung eingesetzt wird.

Sie sind *baumartig* aufgebaut: Ein Dokument selbst wird als *Wurzelelement* betrachtet, in das sich alle (Teil-)Elemente hierarchisch einfügen.

Elemente werden in XML, wie in HTML, über *Tags* ausgezeichnet. Während in HTML Tags wie `<br>` (Zeilenumbruch) ohne End-Tag erlaubt sind, müssen in XML Tags immer geschlossen werden. Das Schließen eines Tags kann auf zweierlei Weise erfolgen. Entweder folgt auf ein öffnendes Tag ein entsprechendes schließendes Tag:

```
<sometag>...</sometag>
```

oder es wird ein in sich geschlossenes Tag notiert:

```
<sometag/>
```

Hierbei handelt es sich um eine Kurzschreibweise, die äquivalent zu `<sometag></sometag>` (ohne irgendetwas zwischen öffnendem und schließendem Tag) ist. So wird in XHTML der Zeilenumbruch durch das `<br/>`-Tag notiert.

Attribute werden wie in HTML angegeben. Allerdings muss ihnen immer ein Wert zugewiesen werden, der in Anführungsstrichen angegeben wird:

```
<sometag attribute1="value1"attribute2="value2"/>
```

Ein weiterer Unterschied zu HTML besteht darin, dass XML zwischen Groß- und Kleinschreibung unterscheidet.

Die Definition eines Dokumententyps erfolgt mit Hilfe der so genannten Document Type Definition (DTD). Definiert werden die in einem Dokumententyp erlaubten XML-Elemente mit ihren Hierarchien und Attributen. DTDs haben eine spezielle Nicht-XML-Syntax.

Jedes (konkrete) XML-Dokument sollte vor Angabe des eigentlichen Wurzelelements mit einem Prolog beginnen, in dem unter anderem die XML-Version und evtl. auch der verwendete Zeichensatz angegeben werden. Danach kann über die spezielle Marke `<!DOCTYPE ...>` der Dokumententyp spezifiziert werden, falls eine passende DTD existiert.

Hier der Anfang eines XML-Dokuments, das die Vorgaben einer DTD buch zur Beschreibung von Büchern befolgt. Die DTD buch ist unter buch.dtd abgelegt:

```
1 <?xml version="1.0" encoding="UTF-8" ?>
2
3 <!DOCTYPE buch SYSTEM "buch.dtd">
4 ...
```

Listing 2.9: XML-Prolog

Ein Dokument, das die Syntaxregeln von XML einhält, heißt *wohlgeformt (well-formed)*. Ein wohlgeformtes Dokument, dessen Struktur den Regeln einer angegebenen DTD entspricht, wird als *gültig (valid)* bezeichnet.

2.5. Statisch, dynamisch und dynamisch erzeugter Inhalt

Sind die Inhalte, die über HTTP vom Web-Server an den Client geschickt werden, auf dem Server abgelegte Dateien, spricht man von *statischen Inhalten*. Solche Dateien sind in der Regel HTML-Dokumente und Bilder.

Liegen die Inhalte nicht in Form von Dateien auf dem Server bereit, sondern werden als Antwort auf eine Anfrage von der Web-Anwendung erst erzeugt, werden sie als *dynamisch erzeugte Inhalte* bezeichnet.

Häufig liest man auch den Begriff *dynamischer Inhalt*. Manchmal wird er synonym zu dynamisch erzeugtem Inhalt benutzt, meistens jedoch ist damit HTML in Verbindung mit Code-on-Demand Techniken (vgl. Abschnitt 2.1.2) gemeint – die „Dynamik“ entsteht also beim Client. Eine Methode für die Nutzung dieser Technik ist die Skriptsprache JavaScript, die im weitesten Sinne als Programmiersprache angesehen werden kann. JavaSript kann in den HTML-Code integriert oder in separate Dateien ausgelagert werden und erweitert die Möglichkeiten von HTML z.B. um die Auswertung von Benutzereingaben, um dynamisches Nachladen von Seiteninhalten oder um die Veränderung der bereits geladenen Seite. Die Verbindung von HTML mit JavaScript wird häufig auch als DHTML, für dynamisches HTML, bezeichnet.

Dieser Kurs beschäftigt sich in erster Linie mit dynamisch erzeugten Inhalten. Die einfachste Art, Inhalte dynamisch zu erzeugen, ist die Verwendung des *Common Gateway Interface (CGI)*, das eine Schnittstelle zwischen dem Web-Server und weiteren Programmen bildet. Der Web-Server leitet die Anfrage als Eingabestrom an ein anderes Programm weiter und sendet dessen Ergebnis zurück an den Client. Die Programme, auch *CGI-Skripte* genannt, sind beliebige ausführbare Dateien, z.B. Perl-Skripte, können aber auch in C geschriebene Programme sein. Das Problem dieses CGI-Ansatzes ist der hohe Server-seitige Ressourcenverbrauch, da für jede Anfrage ein eigener Prozess gestartet werden muss. Das Prinzip, einen Request entsprechend aufbereitet an ein Programm weiterzuleiten und dessen Ergebnis als Response an den Client zurückzuschicken, ist aber allen Verfahren zur Erzeugung dynamischer Inhalte gemein.

2.6. Session

Moderne Web-Anwendungen sind komplexe Anwendungsprogramme, die meist temporäre Daten speichern, also sogenannte *Zustandsinformationen* besitzen. So muss z.B. während des Online-Shoppings nachgehalten werden, welche virtuellen Artikel die Benutzerin bisher in ihren virtuellen Einkaufswagen gelegt hat. Prinzipiell gibt es zwei Möglichkeiten, derartige Zustandsinformationen zu speichern: Entweder der Client speichert den Zustand, oder der Server. Aus sicherheitsrelevanten Überlegungen, und da die Anwendungslogik durch Software auf dem Server realisiert wird, liegt es näher, die entsprechenden Daten auf dem Server zu speichern. Dabei stellt sich allerdings ein neues Problem: Wie wird bei einem eingehenden Request das zugehörige Client-Programm eindeutig identifiziert, damit ihm „seine“ Zustandsinformationen zugeordnet werden können? Wir gehen in der weiteren Diskussion davon aus, dass zu jeder Zeit ein Client nur einer einzigen Benutzerin zugeordnet ist, sodass wir nicht zwischen Benutzerin und Client unterscheiden werden, es sei denn wir weisen ausdrücklich darauf hin.

Die Verwendung der IP-Adresse (vgl. Abschnitt 2.3) als Identifikator für den Client scheidet aus mehreren Gründen aus. Zum einen können sich über Router mehrere Rechner eines lokalen Netzes nach außen hin dieselbe IP-Adresse teilen. Zum anderen kann sich die dynamische IP-Adresse eines Clients zwischen zwei Anfragen ändern, etwa wenn der Client über eine Modem-Verbindung mit dem Internet verbunden ist und die Verbindung zwischenzeitlich getrennt wurde.

Zur Lösung dieses Problems wird das Konzept der *Session (Sitzung)*verwendet. Eine Session ist eine logische, dauerhafte Verbindung zwischen Client und Server. Dabei wird eine eindeutige Zahl oder Zeichenkette, die so genannte *Session-ID*, vom Server nach dem ersten Request des Client erzeugt und dem Client bei der ersten Response mit übermittelt. Bei jeder folgenden Anfrage schickt der Client seine Session-ID mit und kann so identifiziert werden. Wenn die Sitzung beendet wird, entweder indem die Benutzerin sich explizit abmeldet oder nachdem eine festgelegte Zeit seit der letzen Anfrage verstrichen ist, wird die Session-ID ungültig und die mit ihr verbundenen Zustandsinformationen werden gelöscht. Da HTTP keine automatisierte Unterstützung für das Versenden der Session-ID vorsieht, müssen sich die Entwickler einer Web-Anwendung selbst darum kümmern. Es gibt drei Möglichkeiten, die Session-ID in eine Anfrage einzubinden:

- URL-Rewriting
- Verwendung von versteckten Formularfeldern
- Cookies

2.6.1. URL-Rewriting

Beim URL-Rewriting wird die Session-ID in die URL eingesetzt, wobei dies auf unterschiedliche Weise geschehen kann. Die Session-ID kann z.B. als zusätzlicher Parameter vom Client in

den Query-String aufgenommen werden. Bei einem Link erhalten wir so:

```
<a href="http://host/path?sessionid=123456">...</a>
```

Die Session-ID kann aber auch vom Server in die URL, z.B. wie folgt, eingefügt werden:

```
<a href="http://host/path;jsessionid=123456">...</a>
```

Ein solcher Link wird als Bestandteil eines HTML-Dokuments vom Server an den Client gesendet. Klickt die Benutzerin diesen Link an, wird die URL (inklusive aller Parameter) und somit auch die Session-ID an den Server zurückgeschickt. Auf diese Weise zirkuliert die Session-ID zwischen Server und Client. URL-Rewriting kann außer bei Links auch bei Formularen (genauer: bei der URL, an die die zugehörigen Formulardaten gesendet werden sollen) eingesetzt werden.

2.6.2. Versteckte Formularfelder

Diese Technik, eine Session-ID zu übermitteln, kann nur bei HTML-Formularen eingesetzt werden. Dazu enthalten die Formulare ein *verstecktes Formularfeld* zur Aufnahme der Session-ID, wie in List. 2.10 demonstriert. Ein solches Formular wird (als Bestandteil eines HTML-Dokuments) vom Server an den Client gesendet. Beim Betätigen des Submit-Buttons wird der Inhalt aller Formularfelder und somit auch die Session-ID an den Server zurückgeschickt. Auf diese Weise zirkuliert die Session-ID zwischen Server und Client.

```
1 ...
2 <form action="someURL" method="post">
3   <input type="hidden" name="sessionid" value="12345" />
4   ...
5 </form>
6 ...
```

Listing 2.10: Session-ID in einem versteckten Formularfeld

Der Nachteil dieser Methode ist, dass für die Dauer einer Session ausschließlich mit Formularen gearbeitet werden muss und keine Links eingesetzt werden können.

2.6.3. Cookies

Die Verwendung von URL-Rewriting oder versteckten Formularfeldern hat den Nachteil, dass jede URL bzw. jedes Formular für den Austausch der Session-ID vorbereitet werden muss. Hier schaffen *Cookies* Abhilfe. Cookies stellen eine Erweiterung des HTTP dar und werden von allen gängigen Browsern und Servern unterstützt. Sie sind kleine Datenpakete, die beliebige Informationen, also auch die Session-ID, clientseitig speichern können. Bei der ersten Kontaktaufnahme des Client mit dem Server schickt dieser mit der Response das Cookie (als Header-Eintrag) mit der Session-ID an den Client. Bei jeder Anfrage des Client an den Server sorgt der Browser dafür, dass das Cookie (als Header-Eintrag) mit an den Server geschickt wird.

Allerdings werden Cookies häufig von Clients nicht akzeptiert, etwa weil Benutzer die Verwendung von Cookies unterbunden haben, um ihre Privatsphäre zu schützen. Cookies verbleiben nämlich auch nach Session-Ende auf dem Client-Rechner, dokumentieren frühere Internet-Zugriffe und können unerwünschte Bezüge zwischen verschiedenen Sessions herstellen. Wollen die Programmierer einer Web-Anwendung sicherstellen, dass auch in diesen Fällen ihre Software nutzbar ist, müssen sie auf URL-Rewriting oder versteckte Formularfelder zurückgreifen.

Kurseinheit 2

Sprachen, Technologien, Medien und Anwendungen

Diese zweite Kurseinheit gibt zunächst einen Überblick über aktuelle und verbreitete Sprachen aus dem Web-Umfeld. Dazu zählen die Skriptsprachen JavaScript und PHP sowie die Auszeichnungssprache CSS. Anschließend werden das Web-Framework ASP.NET, die JavaScript-Bibliothek jQuery und das Konzept Ajax kurz erläutert.

Im heutigen Internet nehmen Medien verschiedener Art einen hohen Stellenwert ein. Grafiken, Video und Audio sind in unterschiedlicher Quantität auf fast jeder Web-Seite vorzufinden. Die Kurseinheit beleuchtet die wichtigsten Grafik- und Videoformate.

Im letzten Kapitel dieser Kurseinheit wird auf ein zentrales Programm zur Nutzung des Internets eingegangen: den Web-Browser. Ob Internet Explorer, Firefox, Chrome, Opera oder Safari, mit der grundlegenden Funktionsweise eines Web-Browsers sollten Entwickler von Web-Anwendungen vertraut sein.

Diese Seite bleibt aus technischen Gründen frei!

3. Sprachen

Für die Entwicklung von Web-Anwendungen werden nicht, wie bei Desktop-Anwendungen, nur einzelne klassische Programmiersprachen wie Java, C++ oder C# verwendet. Vielmehr wird eine Komposition aus verschiedenen Sprachtypen verwendet, um dadurch unterschiedliche Funktionalitäten für die jeweils entsprechenden Bereiche zu bieten. Darunter fallen die sogenannten Auszeichnungssprachen HTML, XHTML und die erweiterbare Auszeichnungssprache XML (XML wird oft auch als Meta-Auszeichnungssprache angesehen). Aktuell wird HTML in der Version 5 von immer mehr Webbrowsern unterstützt, obwohl die Entwicklung dieser Version noch nicht abgeschlossen ist. Die Arbeit an dieser Spezifikation wird seit 2008 durchgeführt und ist mittlerweile abgeschlossen. Zur Definition von Stilvorlagen für HTML-Dokumente wird die Sprache CSS (Cascading Style Sheets) verwendet. Eine Unterkategorie der Programmiersprachen bilden die sogenannten Skriptsprachen. Hierzu zählen die Sprache JavaScript für Software auf einem Client-Rechner, die bereits im Rahmen von „Code on Demand"-Technik erwähnt wurde, und PHP als Skriptsprache für den Server. Beide Sprachen zählen zu den am häufigsten verwendeten Sprachen im Web-Bereich.

Dieses Kapitel soll einen Überblick verschaffen und hat keineswegs eine vollständige Beschreibung der erwähnten Sprachen zum Ziel. Zu Einführungen in die jeweilige Sprachverwendung und zu Sprachreferenzen findet man leicht geeignete Literatur. Einige relevante Literaturhinweise werden in dieser Kurseinheit gegeben.

3.1. CSS

Mit der Auszeichnungssprache HTML wird die Struktur einer Internetseite durch bestimmte, vorgegebene Elemente (Tags) beschrieben. Darunter fallen auch Elemente, mit denen die Benutzer der Internetseite direkt interagieren können, z.B. ein Textfeld oder eine Schaltfläche innerhalb eines Formular-Tags. Die Formatierung von Texten und die Gestaltung von HTML-Elementen spielen bei einem reinen HTML-Dokument eine untergeordnete Rolle. Die Möglichkeiten eine Internetseite mit reinem HTML zu gestalten sind daher sehr beschränkt.

CSS ist die Abkürzung für Cascading Style Sheets (geschachtelte Formatvorlage) und wurde als CSS 1.0 im Jahre 1996 mit der HTML-Version 4 eingeführt. Das W3C (World Wide Web Consortium) definiert CSS mit dem folgenden Satz: „Cascading Style Sheets (CSS) is a simple mechanism for adding style (e.g. fonts, colors, spacing) to Web documents" Quelle: [CSS]. Aktuell ist die Version 3 (CSS3) in Bearbeitung (seit dem Jahr 2000).

HTML bietet verschiedene Möglichkeiten, Formatmerkmale, die mit CSS definiert wurden, in

ein Dokument einzubinden. Zum einen können die für HTML-Elemente zu verwendenden Formatmerkmale in einem speziellen Tag innerhalb der HTML-Seite definiert werden. Es kann jedoch auch eine separate Datei mit der Endung `.css` erstellt werden, die in das HTML-Dokument mit Hilfe einer Referenz eingebunden wird. Eine dritte Möglichkeit bietet die Einbindung von Formatmerkmalen direkt innerhalb eines HTML-Elementes. Allerdings sollte auch bei der Entwicklung von Web-Seiten möglichst eine Aufteilung von „Inhalt“ und „Gestaltung“ beibehalten werden. CSS unterstützt diese Aufteilung.

Einen ersten Eindruck von den Möglichkeiten von CSS zur Gestaltung von Internetseiten liefert ein kleines Beispiel. Es besteht im ersten Schritt aus einer HTML-Seite. Im zweiten Schritt werden alle drei genannten Varianten der Einbindung von CSS exemplarisch dargestellt.

Die durch den folgenden HTML-Code erzeugte Seite stellt einen Textabsatz mit dem Tag `<p>`, einen allgemeinen Block mit dem Tag `<div>` und eine Schaltfläche mit dem Tag `<input type="button">` dar.

```
1  <html>
2     <head>
3        <title>Erste CSS-Seite</title>
4     </head>
5     <body>
6        <p>Willkommen zur ersten CSS-Seite.</p>
7        <div></div>
8        <input type="button" value="Hier dr&uuml;cken" />
9     </body>
10 </html>
```

Listing 3.1: Einfache HTML-Seite ohne CSS

Das Ergebnis sieht in einem Webbrowser folgendermaßen aus:

Abbildung 3.1.: Einfache HTML-Seite ohne CSS

Nun gestalten wir jedes dieser drei HTML-Elemente mit entsprechenden CSS-Regeln. Die erste Variante sieht vor, alle CSS-Regeln direkt innerhalb der HTML-Tags zu schreiben. Die CSS-Regeln werden als Zeichenkette dem Attribut `style` zugeordnet. Jede Regel ist ein Paar aus der Eigenschaft und ihrem Wert. Regeln werden durch Semikolons voneinander getrennt.

Es resultiert der folgende Code:

```
1  <html>
2     <head>
3        <title>Erste CSS-Seite</title>
4     </head>
5     <body>
6        <p style="width: 300px; height: 30px; background-color:rgb(100, 100,
             100); color: rgb(255, 255, 0)">
7           Willkommen zur ersten CSS-Seite.
8        </p>
9        <div style="width: 400px; height: 100px; border-style: dashed; border
             -width: 10px; margin-bottom: 20px"></div>
10       <input type="button" value="Hier dr&uuml;cken" style="background-
             color: rgb(100, 255, 100); border-radius: 15px; width: 300px;
             height: 50px; text-align: center; font-size: 20px;" />
11    </body>
12 </html>
```

Listing 3.2: Einfache HTML-Seite mit CSS innerhalb der HTML-Elemente

Für jedes HTML-Element wurden innerhalb der `style`-Attribute mehrere CSS-Regeln definiert, die das jeweilige Element gestalten.

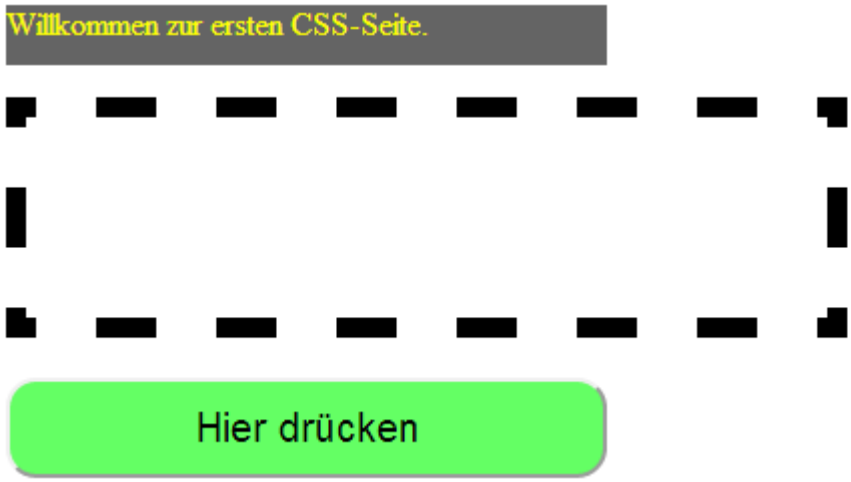

Abbildung 3.2.: Einfache HTML-Seite mit CSS innerhalb der HTML-Elemente (Ausgabe)

Bei den beiden weiteren Möglichkeiten, CSS in eine HTML-Seite einzubinden, werden die CSS-Regeln nicht direkt an ein HTML-Element gebunden, sondern außerhalb definiert. Dies geschieht entweder durch die Einbindung einer CSS-Datei oder durch die Angabe eines Style-Tags im `head`-Bereich der HTML-Seite. Beide Varianten verwenden einen Mechanismus, um CSS-Regeln an HTML-Elemente zu binden: sogenannte Selektoren. Ein Selektor ist ein Wort, das entweder den Namen eines Standard-HTML-Elementes wiedergibt (z.B. `h1`, `p`, `div`) oder das ein einzelnes HTML-Element oder eine Gruppe von HTML-Elementen referenziert.

Damit ein HTML-Element eindeutig identifizierbar ist, wird das Attribut `id` verwendet. Diesem Attribut id wird eine eindeutige Zeichenkette zugeordnet, die z.B. die Funktion des HTML-

Elementes wiedergibt, z.B.: `<input type="button"id="berechnen"/>` oder `<p id="ueberschrift"></p>`. Um auf eine Gruppe von HTML-Elementen mit einem CSS-Selektor zugreifen zu können, wird das Attribut class verwendet, an das wie bei dem Attribut id eine Zeichenkette übergeben wird, die die Gruppe identifiziert. Beinhaltet eine HTML-Seite z.B. mehrere Textabsatzelemente `<p>`, die die Funktion einer Überschrift übernehmen, so sollte ihnen der gleiche Wert für das `class`-Attribut übergeben werden, damit die entsprechende CSS-Regel für alle diese Elemente gilt: `<p class="ueberschrift"></p>`. Tabelle 3.1 listet einige mögliche Selektoren auf.

Selektor(en)	**Beschreibung**
`p, a, div, h1, body`	Diese Selektoren referenzieren HTML-Elemente. Wird z.B. der `p`-Selektor gewählt, werden die definierten CSS-Regeln für alle `p`-Elemente in der HTML-Seite gesetzt.
`#eindeutigerName`	Dieser Selektor verweist auf ein HTML-Element mit der ID eindeutigerName. Er wird mit `id="eindeutigerName"` gesetzt.
`.gruppenName`	Dieser Selektor verweist auf eine Gruppe von HTML-Elementen mit dem `class`-Attribut `gruppenName`. Er wird mit `class="gruppenName"` gesetzt.

Tabelle 3.1.: Selektoren in CSS

Um alle drei Typen von Selektoren in unserem Beispiel unterzubringen, definieren wir eine Menge von CSS-Regeln, in Codeausschnitt 3.3.

```
1  p {
2     width: 300px;
3     height: 30px;
4     background-color: rgb(100, 100, 100);
5     color: rgb(255, 255, 0);
6  }
7
8  .einRahmen {
9     width: 400px;
10    height: 100px;
11    border-style: dashed;
12    border-width: 10px;
13    margin-bottom: 20px;
14 }
15
16 #bestaetigen {
17    width: 300px;
18    height: 50px;
19    background-color: rgb(100, 255, 100);
20    border-radius: 15px;
21    text-align: center;
22    font-size: 20px;
23 }
```

Listing 3.3: Definition mehrerer CSS-Regeln

Um auf CSS-Regeln dieser Menge aus einem HTML-Dokument heraus zugreifen zu können, müssen diese entweder Teil des HTML-Dokumentes sein oder als Bestandteil einer externen CSS-Datei eingebunden werden. Dazu kann innerhalb eines HTML-Dokumentes im Kopf das Tag `<style type="text/css"> ... </style>` definiert werden. Der Inhalt dieses Tags ergibt sich aus Codeausschnitt 3.3.

Sollen alle CSS-Regeln in eine Datei ausgelagert werden, muss der Codeausschnitt 3.3 in einer CSS-Datei gespeichert werden. Im Kopf der HTML-Seite verweist dann eine Referenz auf diese Datei. Dies wird mit dem `<link>`-Tag realisiert:

```
<link href="datei.css"type="text/css"rel="stylesheet"/>
```

CSS bietet weitere und weitreichende Möglichkeiten zur Gestaltung einer Internetseite. CSS in der Version 3 ist in Module aufgeteilt. Die wichtigsten Module sind: Selektoren, das Box-Modell, Hintergründe und Rahmen, Texteffekte, 2D- und 3D-Transformationen, Animationen, Mehr-Spalten-Layout und die Benutzungsoberfläche.

3.2. PHP

PHP ([P]HP [H]ypertext [P]reprocessor) ist eine plattformunabhängige, serverseitig zu verwendende Skriptsprache, die frei verfügbar ist und im Jahr 1995 von der „PHP Group“ veröffentlicht wurde. PHP ist aktuell in der Version 5.6.10 (Stand 11. Juni 2015) verfügbar und ermöglicht die Erzeugung von dynamisch generierten Internetseiten, bei denen der darzustellende Inhalt durch ein PHP-Programm auf dem Server erzeugt und anschließend an den Client gesendet wird. Die Syntax von PHP ist durch die Sprachen C, C++, Java und Perl beeinflusst.

PHP verfügt außerdem über vielfältige Methoden, um auf eine Datenbank zugreifen zu können. Die Kombination aus PHP als Skriptsprache und MySQL als relationales Datenbanksystem ist am weitesten verbreitet. MySQL steht als freie Software zur Verfügung, wird aber auch unter einer kommerziellen Lizenz angeboten.

Um in die Skriptsprache PHP einzuführen, wird ein kleines Beispielprogramm vorgestellt. Danach gehen wir auf ausgewählte Sprachelemente von PHP ein.

3.2.1. Einleitendes Beispiel

Im folgenden kleinen Beispielprogramm sollen zwei Zahlen von der Benutzerin eingegeben und, durch Betätigung der entsprechenden Schaltfläche, entweder addiert oder multipliziert werden. Das Ergebnis der Berechnung soll anschließend auf einer neuen Seite dargestellt werden.

Wir benötigen zwei PHP-Dateien: eine Datei für die Eingabe (eingabe.php) und eine Datei für die Ausgabe (ausgabe.php). Der Code für beide Seiten ist in Codeausschnitt 3.4 und Codeausschnitt 3.5 dargestellt.

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
    w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>PHP Beispielanwendung</title>
</head>
<body>
   <form action="ausgabe.php" method="post">
      Zahl 1: <input type="text" name="zahl1" />
      Zahl 2: <input type="text" name="zahl2" />

      Operation:
      <input type="radio" name="operation" value="addition" />Addition
      <input type="radio" name="operation" value="multiplikation" />
         Multiplikation<br/>
      <input type="submit" value="Ergebnis berechnen" />
   </form>
</body>
</html>
```

Listing 3.4: PHP-Beispielprogramm: eingabe.php

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

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
    w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
   <title>PHP Beispielanwendung</title>
</head>
<body>
   <h1>Hier ist das Ergebnis:</h1>
   <?php
         $zahl1 = $_POST["zahl1"];
         $zahl2 = $_POST["zahl2"];

         $operation = $_POST["operation"];

      $ergebnis = 0;

      if ($operation == "addition") {
         $ergebnis = $zahl1 + $zahl2;
      }
      elseif ($operation == "multiplikation") {
            $ergebnis = $zahl1 * $zahl2;
      }
      ?>
      <p>
         Das Ergebnis ist: <?php echo $ergebnis; ?>
```

```
28          </p>
29  </body>
30  </html>
```

Listing 3.5: PHP-Beispielprogramm: ausgabe.php

Ein PHP-Programm kann aus normalem HTML/XHTML-Code bestehen und muss nicht zwingend PHP-Elemente enthalten. Auch eine Vermischung von HTML/XHTML und PHP in einem PHP-Programm ist erlaubt, wie im Codeausschnitt 3.5 zu sehen ist.

Das Programm bietet auf der Startseite (eingabe.php) zwei Textfelder zur Eingabe jeweils einer Zahl und zwei Optionsfelder zur Auswahl der Rechenoperation an. Mit Hilfe einer Schaltfläche können die eingegebenen Werte an den Server gesendet und das Ergebnis berechnet und ausgegeben werden. PHP nutzt die Formularelemente, die Teil des HTML-Standards sind. Das `action`-Attribut des Formular-Tags gibt die PHP-Datei an, die für die Bearbeitung des `submit` (Absendung) verantwortlich sein soll sowie die Methode der Datenübertragung (hier POST). Codeausschnitt 3.5 enthält eine Mischung aus HTML/XHTML-Code und PHP-Code. PHP-Code muss immer innerhalb der PHP-Tags `<?php ... ?>` (oder einfacher `<? ... ?>`) stehen. HTML-Code, der innerhalb eines PHP-Codeblocks auftritt, führt zu einem Ausführungsfehler.

In unserem Beispielprogramm (ausgabe.php) werden in den Zeilen 12, 13 und 15 die Werte der Anfrage, also die in der vorigen Seite im Formular angegebenen Werte, in Variablen gespeichert. In Zeile 17 wird eine neue Variable initialisiert, die später das Ergebnis der Berechnung enthalten soll. Variablen beginnen in PHP immer mit einem Dollar-Zeichen ($). Ab Zeile 19 wird die Abfrage gestartet, welche Rechenoperation angewandt werden soll. Auf der Startseite (eingabe.php) sind zwei Optionsfelder (Radio-Button) definiert, die als Attribute den gleichen Namen, aber unterschiedliche Werte enthalten. Optionsfelder mit gleichem Namen gehören einer Gruppe an und schließen sich stets gegenseitig aus (es können nicht zwei oder mehrere Optionsfelder einer Gruppe gleichzeitig ausgewählt sein). Die Variable `$operation` holt sich mit Hilfe von `$_POST[...]` den Request-Parameter mit dem Namen operation, der den Wert des von der Benutzerin aktivierten Optionsfeldes enthält. Innerhalb der `if`-Abfrage wird geprüft, ob die Addition oder die Multiplikation ausgewählt wurde. Je nach Auswahl werden die eingegebenen Zahlen addiert (Zeile 20) oder multipliziert (Zeile 23).

Da HTML-Code nicht direkt in einen PHP-Codeblock integriert werden kann, bietet PHP Möglichkeiten zur Ausgabe von Texten an. Dazu gehört das Sprachkonstrukt `echo`. Hierbei handelt es sich nicht um eine Funktion, es müssen somit keine Klammern angegeben werden. Mit `echo` kann ein Variablenwert, ein Wert eines ausgewerteten Ausdrucks oder eine Zeichenkette ausgegeben werden:

```
1  echo $ergebnis;
2  echo "Das ist eine Zeichenkette.";
3  echo $zahl1 + $zahl2;
```

Listing 3.6: Der echo-Befehl in PHP

3.2.2. Einbettung von PHP-Code in HTML

PHP-Programmcode kann auf einfache Art und Weise in eine HTML-Seite integriert werden. So kann eine PHP-Anweisung oder ein PHP-Codeblock direkt in eine Seite integriert werden, es können aber auch PHP-Dateien ausgelagert und bei Bedarf eingebunden werden. Um PHP direkt in HTML einzubetten, wird eine Markierung durch die Tags `<?php ... ?>` verwendet. Sie leitet eine einzelne Anweisung oder einen Block von Anweisungen ein. Dabei gibt es keine Beschränkungen für eine solche Markierung, sie kann an jeder Stelle einer HTML-Seite eingefügt werden.

Der HTML-Code in Codeausschnitt 3.7 besteht teilweise aus Elementen einer HTML-Seite und generiert diese Seite teilweise durch Ausführung von eingebettetem PHP-Code.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
       w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4
5  <html xmlns="http://www.w3.org/1999/xhtml">
6  <head>
7     <title>HTML-Seite mit PHP-Block</title>
8  </head>
9  <body>
10    <?php
11       echo "<h1>Das ist der mit PHP erzeugte HTML-Code</h1>";
12       echo "<p>Dieser Abschnitt wird ebenso mit PHP erzeugt</p>";
13    ?>
14
15    <h1>Das ist der direkt in HTML geschriebene HTML-Code</h1>
16    <p>Dieser Abschnitt ist direkt durch HTML definiert</p>
17 </body>
18 </html>
```

Listing 3.7: HTML-Code mit PHP-Block

Eine Datei mit dem Inhalt aus Codeausschnitt 3.7 muss als PHP-Datei gespeichert werden (Endung `.php`), da die PHP-Anweisungen ansonsten nicht ausgeführt werden. Das Resultat ist eine HTML-Seite mit zwei Überschriften-Tags und zwei Textabschnitt-Tags. Die Benutzer der PHP-Seite können zu keinem Zeitpunkt Einblick in den PHP-Code nehmen, sie sehen lediglich den HTML-Code, der vom PHP-Interpreter aus der Datei erzeugt wurde.

Die Einbindung einer externen PHP-Datei wurde bereits im einführenden Beispiel vorgestellt. In dem Beispiel wird dem `action`-Attribut des `form`-Tags der Name einer PHP-Datei übergeben, die für die Verarbeitung der Formulardaten zuständig ist. Diese Datei besteht wieder aus einem Gemisch aus HTML und PHP-Code.

Im Kontext von JavaScript und PHP-Funktionen enthält eine PHP-Datei aber meist nur PHP-Anweisungen und keinen HTML-Code.

3.2.3. Kommentare in PHP

Je komplexer ein Programm wird, desto mehr sollte es von den Entwicklern strukturiert und modularisiert werden. Kommentare im Programmcode bieten die Möglichkeit, die implementierten Funktionalitäten zu beschreiben, eine bessere Zusammenarbeit mit anderen Entwicklern umzusetzen und bieten eine Gedankenstütze, wenn erst nach längerer Zeit wieder am Programmcode gearbeitet wird.

PHP bietet zwei Arten von Kommentaren: einzeilige und mehrzeilige Kommentare. Ein einzeiliger Kommentar beginnt mit den Zeichen `//` und endet am Ende der Zeile. Alle Zeichen dieser Zeile nach diesen Kommentarzeichen werden vom PHP-Interpreter nicht beachtet und natürlich auch nicht verarbeitet. Ein mehrzeiliger Kommentar beginnt mit den Zeichen `/*` und endet mit den Zeichen `*/`. Alle Zeichen zwischen diesen beiden Markierungen werden vom PHP-Interpreter nicht beachtet. Üblicherweise beschreiben einzeilige Kommentare einzelne PHP-Anweisungen, während mehrzeilige Kommentare dazu verwendet werden, größere Programmblöcke zu beschreiben. Codeausschnitt 3.8 ist ein kleines Beispiel für Kommentare in PHP.

```
1  <html>
2  <body>
3     <?php
4        echo "Wir kommentieren PHP-Code"; // Ein Kommentar
5
6        /*
7           Dies ist ein Kommentar, der
8           über mehrere Zeilen geht
9        */
10
11       // Anweisungen innerhalb eines Kommentars werden
12       // nicht ausgeführt: echo "Test";
13    ?>
14 </body>
15 </html>
```

Listing 3.8: Kommentare in PHP

3.2.4. Variablen, Datentypen und Operatoren

Wie die meisten Programmiersprachen bietet auch PHP Variablen an, stellt mehrere Datentypen zur Verfügung und kann komplexe Ausdrücke durch die Verknüpfung von Variablen und Konstanten mit Hilfe von Operatoren verarbeiten. Der Datentyp einer Variablen (oder Konstanten) definiert den Wertebereich und somit die Werte, die von der Variablen aufgenommen werden können. PHP stellt die Tabelle 3.2 dargestellten Datentypen zur Verfügung.

PHP wird als Skriptsprache bezeichnet. Wie bei vielen anderen Skriptsprachen, müssen Variablen nicht deklariert werden. Eine Variable kann somit einfach nur durch ihre Angabe im Programmcode eingeführt werden. In Codeausschnitt 3.5 werden z.B. die Variablen `$zahl1`

Beschreibung	Bezeichnung	Beispiele
Boolescher Wert	Boolean	true, false
Ganze Zahl	Integer	30, 0, 43892, -25, -1
Zahl mit Nachkommastellen	Float	3.542, -26.3452, 1.0
Zeichenkette	String	„Guten Tag“, „c“
Feld (ein- oder mehrdimensional)	Array	(1,3,6,7,8), („a“, „b“, „c“)
Objekt	Object	z.B. ein Objekt einer PHP-Klasse

Tabelle 3.2.: Datentypen in PHP

und `$zahl2` eingeführt und ohne eine vorherige Deklaration und somit ohne explizite Angabe ihres Datentyps mit Werten belegt.

Variablen können mit Werten belegt werden: `$zahl1 = 20`. Innerhalb eines PHP-Programms kann eine Variable ihren Datentyp wechseln, indem sie mit einem Wert eines anderen Datentyps belegt wird. Zwei aufeinanderfolgende Anweisungen:

```
1    $zahl1 = 20;
2    $zahl1 = "Hallo";
```

sind somit möglich und werden vom PHP-Interpreter nicht als Fehler bewertet.

Im Folgenden werden einige beispielhafte Wertebelegungen für Variablen aufgeführt. Bei dem Datentyp Array muss das Schlüsselwort `array` angegeben werden. Die Einträge eines Arrays werden innerhalb von runden Klammern, durch Kommata getrennt, aufgelistet.

```
1    $meinText = "Herzlich Willkommen zum Kurs 1796";
2    $eineZahl = 25.275;
3    $zahlen = array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
4    $woerter = array("wort1", "wort2", "wort3", "wort4");
5    $wahrheitswert = false;
```

PHP stellt nicht nur Operatoren für mathematische Berechnungen (arithmetische Operatoren) zur Verfügung, sondern verschiedene Operatorklassen: Arithmetische Operatoren, Zuweisungsoperatoren, Bit-Operatoren, Vergleichsoperatoren, Inkrement- und Dekrementoperatoren, logische Operatoren, Zeichenketten-Operatoren, Array-Operatoren und Typ-Operatoren. Wir beschränken uns in dieser Kurseinheit auf arithmetische und Zeichenketten-Operatoren. Es stehen insgesamt fünf arithmetische Operatoren zur Verfügung, die in Tabelle 3.3 erläutert werden.

3.2.5. Verarbeitung von HTML-Formularen

Ein einfacher Weg, Daten der Besucher einer Internetseite aufzunehmen und auf dem Server zu verarbeiten, ist das HTML-Formular. Im einführenden Beispiel wird ein Formular verwendet, damit die Benutzerin Operationen mit zwei Zahlen durchführen kann. Hierzu werden zwei Textfelder, zwei Optionsfelder und eine Submit-Schaltfläche erzeugt. Wird eine Submit-Schaltfläche

Operator	Beschreibung
+	Addition `$ergebnis = 10 + 20;`
-	Subtraktion `$ergebnis = 20 - 10;`
*	Multiplikation `$ergebnis = 10 * 20;`
/	Division `$ergebnis = 20 / 10;`
%	Modulo: Rest einer ganzzahligen Division `$ergebnis = 7 % 3;`

Tabelle 3.3.: Arithmetische Operatoren in PHP

innerhalb eines Formulars betätigt, wird die durch das Attribut `action` des Formular-Tags definierte Aktion mit der im Attribut `method` definierten Methode ausgeführt. Das `action`-Attribut spezifiziert die Adresse, an die die Daten des Formulars bei einem `submit` gesendet werden. Diese Adresse zeigt meist auf eine Datei (die z.B. ein PHP-Programm enthält), die für die Verarbeitung der übergebenen Daten verantwortlich ist.

Das Attribut method kann zwei mögliche Werte haben: `post` und `get`. Es hat den Initialwert get. Der Wert dieses Attributs gibt an, wie die Daten des Formulars an den Server gesendet werden.

Bei `get` werden die Daten vom Webbrowser, getrennt durch ein „?“, an die Adresse, die an das `action`-Attribut übergeben wurde, angehängt. Dies kann folgendermaßen aussehen:

```
http://www.einehomepage.de/auswertung.php?name=Otto&passwort=12345
```

Diese Methode ist für sensible Daten zu vermeiden, da alle eingegebenen Daten im Klartext in der Adresszeile zu sehen sind. Außerdem ist die Länge einer Adresszeile, abhängig vom Webbrowser, beschränkt. Möchte man sehr viele Daten in einem Schritt an den Server senden, ist die Methode `get` deshalb nicht geeignet.

Die zweite Möglichkeit, Daten eines Formulars an einen Server zu senden, bietet die Methode `post`. Sie nutzt nicht das Anhängen von Parametern an die Adresse, sondern überträgt die Daten innerhalb der Nachricht, die an den Server gesendet wird. Die übertragenen Daten sind somit für die Benutzer der Internetseite nicht ohne weiteres einsehbar. Zudem ist deshalb der Platz innerhalb der Nachricht für die mit `post` gesendeten Formulardaten nicht beschränkt. Die `post`-Methode sollte deshalb immer dann verwendet werden, wenn viele oder große Daten (z.B. bei einer Dateiübertragung) oder wenn sensible Daten übertragen werden.

3.2.6. Verzweigung und Schleifen

Die Sprache PHP umfasst, wie fast jede Programmiersprache, Sprachelemente für Verzweigungen und Schleifen. Diese Sprachelemente werden auch Kontrollstrukturen genannt, da sie den Kontrollfluss des Programms während der Ausführung bestimmen.

Um eine Verzweigung zu programmieren, stellt PHP zwei Möglichkeiten zur Verfügung: `if ... elseif ... else` und `switch`:

```
1    if (Bedingung) {
2       bedingter Code
3    }
4    [elseif(andereBedingung) {
5       bedingter Code
6    }]
7    [else{
8       auszuführender Code, wenn keine Bedingung zutrifft
9    }]
```

Listing 3.9: if...elseif...else

```
1  switch(Variable) {
2     case Wert1:
3        Auszuführender Code, wenn Variable=Wert1
4        [break;]
5     case Wert2:
6        Auszuführender Code, wenn Variable=Wert2
7        [break;]
8     [weitere Fälle von case]
9     [default:
10       Auszuführender Code, wenn kein Fall von case zutrifft]
11    }
```

Listing 3.10: Die switch-Verzweigung

Die `if`-Anweisung enthält zwei Klammerpaare. Zwischen runden Klammern steht die Bedingung, die erfüllt sein muss, damit der Code zwischen den geschweiften Klammern ausgeführt wird. Bei der Bedingung muss es sich um einen Ausdruck handeln, der entweder zu *wahr* oder zu *falsch* ausgewertet werden kann. Optional ist es möglich mit `elseif` weitere Bedingungen abzufragen. Sollte die erste Bedingung nicht zutreffen, wird die nächste Bedingung geprüft und ggf. der entsprechende Codeblock ausgeführt. An das Ende (und nur an das Ende) einer `if`-Anweisung kann ein `else`-Block angefügt werden, der immer dann ausgeführt wird, wenn keine Bedingung zu *wahr* ausgewertet wurde.

Die `switch`-Anweisung enthält, wie auch die `if`-Anweisung, zwei Klammerpaare. Zwischen den runden Klammern steht eine Variable oder ein Ausdruck, dessen Wert mit den Werten der in den geschweiften Klammern angegebenen Fälle (cases) verglichen wird. Stimmen beide für einen Fall überein, wird der gesamte dahinter stehende Code ausgeführt. Überraschenderweise wird aber auch der Code, der zu den nachfolgenden Fällen gehört ausgeführt, ohne dass die weiteren `case`-Ausdrücke ausgewertet werden. Um dies zu vermeiden, sollte am Ende eines jeden `case`-Programmblocks die Anweisung `break` eingefügt werden. Diese führt dazu, dass die gesamte `switch`-Anweisung verlassen wird. `default` gibt einen Fall an, der immer dann ausgeführt wird, wenn kein Fall positiv ausgewertet wurde.

PHP kennt auch mehrere Kontrollstrukturen für Schleifen. Der erste Schleifentyp ist die sogenannte `for`-Schleife. Die `for`-Schleife kann nur dann eingesetzt werden, wenn die Anzahl der Schleifendurchläufe vor Schleifenbeginn bekannt ist. Den generellen Aufbau einer `for`-Schleife und ein Beispiel für die Anwendung zeigt Codeausschnitt 3.11. In dem mit `echo` auszugebenen

Ausdruck wird der Konkatenationsoperator „.“ (Punkt) benutzt, um eine Zeichenkette und eine Variable zu einer neuen Zeichenkette zusammenzufügen.

```
for (Initialisierung; Bedingung; Veränderung der Zählervariable) {
   // Programmanweisungen
}

<?php
   for ($i = 1; $i <= 10; $i++) {
      echo "i hat den Wert " . $i;
   }
?>
```

Listing 3.11: Die for-Schleife in PHP

Eine `for`-Schleife besteht aus dem Schleifenkopf und dem Schleifenkörper. Der Schleifenkörper steht zwischen geschweiften Klammern, die den auszuführenden Programmcode begrenzen. Der Schleifenkopf besteht aus drei Ausdrücken, die durch Semikola voneinander getrennt werden. Der erste Ausdruck enthält die Initialisierung einer Zählervariablen `$i` (die Bezeichnung dieser Variable ist frei wählbar). In unserem Beispiel wird diese Variable mit dem Wert 1 initialisiert. Der zweite Ausdruck enthält die Bedingung, die zu Beginn jedes Schleifendurchlaufs geprüft wird. Ist die Bedingung *wahr*, wird der Programmblock des Schleifenkörpers erneut ausgeführt. Ist die Bedingung nicht *wahr*, wird die Schleife beendet. Der dritte Ausdruck enthält die Veränderung der Zählervariable. Bei dem doppelten Pluszeichen handelt es sich um den sogenannten Inkrementoperator, der als Abkürzung für `$i = $i + 1` steht. Im Beispiel wird die `echo`-Anweisung insgesamt also zehnmal ausgeführt.

Der zweite Schleifentyp, den wir betrachten, ist die `while`-Schleife. Eine `while`-Schleife wird allgemein dann verwendet, wenn eine unbekannte Anzahl von Wiederholungen durchgeführt werden soll. Z.B. werden `while`-Schleifen häufig für Datenbankabfragen genutzt, bei denen die Anzahl der abgefragten Datensätze nicht bekannt ist. Codeausschnitt 3.12 zeigt den generellen Aufbau einer `while`-Schleife und ein Beispiel für ihre Anwendung.

```
while (Bedingung) {
   // Programmanweisungen
}

<?php
   $i = 1;
   while ($i <= 10) {
      echo "i hat den Wert " . $i;
      $i = $i + 1;
   }
?>
```

Listing 3.12: Die while-Schleife in PHP

Wie zu sehen ist, hat die `while`-Schleife einen einfacheren Aufbau als die `for`-Schleife. Der Schleifenkopf besteht hier lediglich aus der Bedingung, die immer geprüft wird, bevor der

Programmcode im Schleifenkörper ausgeführt wird. Die Zählervariable dieses Beispiels muss vor Beginn der Schleife initialisiert werden.

Schleifen stellen für die algorithmische Lösung von Problemen in allen Bereichen der Softwareentwicklung eine wesentliche Kontrollstruktur dar. Neben der Anwendung in Programmier- oder Skriptsprachen für Berechnungen oder Datenbankabfragen können Schleifen im Zusammenhang mit HTML auch dazu genutzt werden, eine Reihe von HTML-Elementen zu erstellen. So kann z.B. die Darstellung einer Bildergalerie oder die Darstellung einer Tabelle mit einer Schleife realisiert werden. Codeausschnitt 3.13 zeigt ein PHP-Programm, das mit Hilfe einer Schleife eine HTML-Tabelle erstellt.

```
1  <html>
2     <head></head>
3     <body>
4        <table>
5           <?php
6              for ($i = 1; $i <= 10; $i++) {
7                    echo "<tr><td>Testeintrag " . $i . "</td></tr>";
8              }
9           ?>
10       </table>
11    </body>
12 </html>
```

Listing 3.13: Erstellung und Ausgabe einer HTML-Tabelle mit einer Schleife in PHP

3.2.7. Funktionen

Je größer ein PHP-Programm wird, desto wahrscheinlicher werden Programmteile mehrfach genutzt. Um zu vermeiden, dass gleiche Programmteile mehrfach im gesamten Programm vorkommen, ist es ratsam diese zentral zur Verfügung zu stellen. Auch in Hinsicht auf eine Modularisierung ist es stets sinnvoll, Anweisungen logisch zusammenzufassen und verschiedene Aufgaben voneinander zu trennen. PHP bietet für diesen Zweck die Möglichkeit, Funktionen zu definieren.

Eine Funktion ist, wie in anderen Programmiersprachen auch, ein Block von Anweisungen, dem für seine Ausführung Parameter übergeben werden können und der einen Wert an den Aufrufer der Funktion zurückgeben kann. Eine Funktion ist aufgeteilt in den Funktionskopf und den Funktionsrumpf. Abbildung 3.3 zeigt den allgemeinen Aufbau einer Funktion.

Bei den Parametern im Funktionskopf handelt es sich um eine Liste von Variablen, die die Werte aufnehmen, die bei einem Funktionsaufruf angegeben werden. Der Rückgabewert kann durch einen beliebigen Ausdruck definiert sein und ist wie die Angabe der Parameter optional.

Codeausschnitt 3.14 zeigt exemplarisch die Definition einer Funktion addiere, die zwei Zahlen als Eingabeparameter übergeben bekommt, ein Ergebnis berechnet und dieses Ergebnis als Rückgabewert zurückgibt. Der Rückgabewert ersetzt quasi in der aufrufenden Codezeile

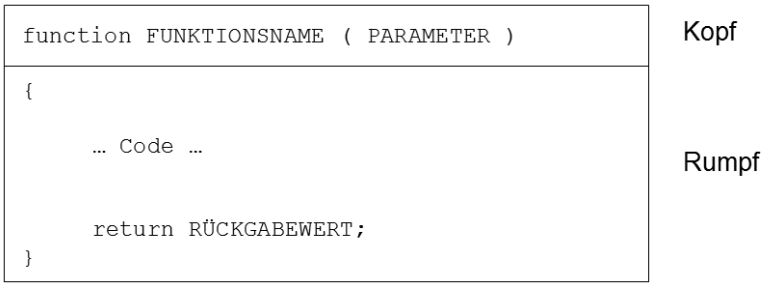

Abbildung 3.3.: Aufbau einer Funktion in PHP

den Funktionsnamen inklusive der angegebenen Parameterliste. Im Beispiel wird die Funktion `addiere` zweimal, mit unterschiedlichen Parameterwerten aufgerufen und das Ergebnis jeweils in einer Variablen gespeichert. Beide Additionsergebnisse werden im Anschluss mit dem `echo`-Befehl auf der HTML-Seite ausgegeben.

```
1  <html>
2     <head></head>
3     <body>
4        <?php
5           function addiere ($zahl1, $zahl2)
6           {
7              $ergebnis = $zahl1 + $zahl2;
8              return $ergebnis;
9           }
10
11          $addition1 = addiere(10, 20);
12          $addition2 = addiere(5, 6);
13
14          echo "Ergebnis der ersten Addition: " . $addition1;
15          echo "Ergebnis der zweiten Addition: " . $addition2;
16       ?>
17    </body>
18 </html>
```

Listing 3.14: Definition und Aufruf einer Funktion in PHP

3.2.8. Klassen und Objekte

Mit der Einführung von PHP 5 ergaben sich besonders im Bereich der objektorientieren Programmierung wesentliche Neuerungen. Objektorientierung bietet Möglichkeiten für einen verbesserten Aufbau eines Programms und erleichtert seine Wartung. Die Grundlagen objektorientierter Konzepte und die Nutzung von Klassen und Objekten werden im Kurs 1793 – „Software Engineering I“ vermittelt. Aus diesem Grund gehen wir in diesem Abschnitt direkt auf die Umsetzung dieser Konstrukte in PHP ein. Grundlage dieses Abschnitts ist das Buch „Einstieg in PHP 5 und MySQL 5“ von Thomas Theis, Kapitel E.

Die Verwendung von Klassen und Objekten in PHP werden wir anhand eines kleinen Beispiels veranschaulichen. In diesem Beispiel wird ein Klasse Fahrzeug mit den Klassenattributen *Bezeichnung* und *Geschwindigkeit* erzeugt. Die Klasse Fahrzeug verfügt über eine Methode, mit der die Geschwindigkeit erhöht werden kann, über eine Methode, die die Bezeichnung des Fahrzeugs setzt und über eine Methode, die die Bezeichnung und die Geschwindigkeit des Fahrzeugs ausgibt.

Codeausschnitt 3.15 zeigt den Code der Klasse.

```
1  <?php
2     class Fahrzeug {
3        private $bezeichnung;
4        private $geschwindigkeit;
5
6        function beschleunigen($wert) {
7           $this->geschwindigkeit += wert;
8        }
9
10       function setzeBezeichnung($bezeichnung) {
11          $this->bezeichnung = $bezeichnung;
12       }
13
14       function ausgabe() {
15          echo "Die Geschwindigkeit von " . $this->bezeichnung . " beträgt "
                 . $this->geschwindigkeit . " km/h.";
16       }
17    }
18 ?>
```

Listing 3.15: Definition der Klasse Fahrzeug in PHP

In der Klasse Fahrzeug werden zwei Variable `$bezeichnung` und `$geschwindigkeit` mit der Sichtbarkeit `private` deklariert, d.h. sie sind außerhalb der Klasse Fahrzeug nicht sichtbar. Zwei Funktionen erhalten einen Parameter, der innerhalb der Funktion verarbeitet wird (`$wert` wird zu `$geschwindigkeit` addiert und `$bezeichnung` wird an das Klassenattribut bezeichnung übergeben). Die Funktion ausgabe gibt die Bezeichnung und die Geschwindigkeit des Fahrzeugs aus. `$this` ist eine Referenz auf das Objekt, zu welchem die Methode gehört, in der es verwendet wird. Zu beachten ist, dass bei Angabe einer Variable nach `$this` kein Dollarzeichen vor deren Name steht.

Neben dem Schlüsselwort `private` gibt es zwei weitere Schlüsselwörter, mit denen man die Sichtbarkeit von Attributen einer Klasse bestimmen kann: `protected` und `public`. Während private die Sichtbarkeit von Attributen auf die Klassendefinition beschränkt und somit nur Methoden bzw. Funktionen der Klasse auf diese Attribute direkt zugreifen können, sind mit `public` deklarierte Attribute auch außerhalb sichtbar. Mit `protected` deklarierte Attribute sind innerhalb der Klassendefinition, aber auch innerhalb von abgeleiteten Klassen dieser Klasse sichtbar.

Codeausschnitt 3.16 zeigt, wie die erstellte Fahrzeugklasse genutzt werden kann, wie Objekte dieser Klasse erzeugt werden und wie unter Verwendung dieser Objekte Methoden aufgerufen

werden.

```
<?php
   class Fahrzeug {
      ...
   }

   $audi = new Fahrzeug();
   $vw = new Fahrzeug();

   $audi->setzeBezeichnung("Mein Audi");
   $audi->beschleunigen(50);

   $vw->setzeBezeichnung("Mein VW-Auto");
   $vw->beschleunigen(70);

   $audi->ausgabe();
   $vw->ausgabe();
?>
```

Listing 3.16: Objekterzeugung in PHP

Nachdem die Klasse Fahrzeug definiert wurde, werden zwei Objekte dieser Klasse erzeugt: `audi` und `vw`. Die Erzeugung eines Objekts einer Klasse geschieht mit dem Operator `new`, dem der Klassenname und ein Paar runde Klammern folgen. Der restliche Programmcode ruft die unterschiedlichen Methoden der Klasse auf den erzeugten Objekten auf. Zuerst erhält das Objekt `audi` eine Bezeichnung und seine Geschwindigkeit wird erhöht, anschließend werden diese beiden Methoden auf dem Objekt `vw` aufgerufen. Letztendlich wird die Methode `ausgabe` auf beiden Objekten aufgerufen, die jeweils die Bezeichnung und die Geschwindigkeit ausgibt.

Eine einfachere Möglichkeit, Werte an die Attribute eines Objekts zu übergeben, erlaubt der sogenannte Konstruktor. Der Konstruktor einer Klasse ist eine Methode, die immer dann aufgerufen wird, wenn ein Objekt der Klasse erzeugt wird. Fügt man einer Konstruktor-Methode Parameter hinzu, so können einem Objekt direkt bei seiner Erzeugung Werte übergeben werden. Ein Konstruktor hat dabei den Aufbau: `public function __construct(PARAMETERLISTE)`. Zu beachten ist der doppelte Unterstrich vor dem Schlüsselwort `construct`. Codeausschnitt 3.17 zeigt die Erzeugung unserer beiden Fahrzeug-Objekte mit Hilfe von Konstruktoren.

```
<?php
   class Fahrzeug { ...
      public function __construct($bez, $wert) {
         $this->bezeichnung = $bez; // Setzen des Attributs
         $this->beschleunigen($wert); // Aufrufen einer Methode
      }
   }
   $audi = new Fahrzeug("Mein Audi", 50);
   $vw = new Fahrzeug("mein VW-Auto", 70);
   $audi->ausgabe();
   $vw->ausgabe();
?>
```

Listing 3.17: Objekterzeugung mit dem Konstruktor in PHP

PHP stellt für die objektorientierte Programmierung weitere Techniken zur Verfügung. Wie auch zu allen anderen Abschnitten gibt es auch zu diesem Abschnitt Übungsaufgaben im Übungssystem, mit denen Sie die Handhabung von PHP üben und vertiefen können.

3.3. JavaScript

Entwickelt man Internetseiten mit HTML und erweitert die Gestaltungsmöglichkeiten mit CSS, so kann man bereits mit diesen beiden Methoden ansprechende Web-Auftritte erstellen. Durch verschiedenartige HTML-Elemente, wie Schaltflächen oder Textfelder, kann die Benutzerin Daten eingeben und an einen Server senden, der diese Daten (z.B. den eingegebenen Namen in einem Textfeld) dann verarbeitet und ein Ergebnis an den Client zurücksendet. Mit PHP haben wir eine Server-seitige Programmiersprache kennengelernt, die einen derartigen Ablauf unterstützt. Doch nicht immer ist es sinnvoll, und auch nicht immer möglich, alle Verarbeitungsschritte von einem Server erledigen zu lassen. 1995 wurde der Grundstein für die Skriptsprache JavaScript (abgekürzt JS) gelegt, mit der es möglich ist, Programmcode auf dem Client durch den Webbrowser ausführen zu lassen. JavaScript erlaubt die Auswertung von Benutzerinteraktionen und die Veränderung sowie Generierung von Inhalten einer HTML-Seite und erweitert somit die bereits erwähnten Möglichkeiten von HTML und CSS.

In diesem Abschnitt möchten wir die Sprache JavaScript kurz vorstellen und auf ihre Integration in HTML eingehen. Generell gibt es zwei Varianten, JavaScript in HTML-Seiten einzubinden. Zum einen kann JavaScript-Code direkt in eine HTML-Seite eingefügt werden, zum anderen kann der Code in eine JavaScript-Datei mit der Endung .js ausgelagert werden. In der HTML-Seite muss dann der Dateiname mit Hilfe eines `<script>`-Tags eingebunden werden. Wie zu sehen ist, ähnelt die Einbindung von JavaScript sehr der Einbindung von CSS in eine Internetseite.

Um einen Einstieg zu geben, betrachten wir ein einfaches Beispiel: Eine HTML-Seite mit einem JavaScript-Block, in dem zwei Zahlen addiert werden.

```
1  <html>
2     <head>
3        <script>
4           var zahl1 = 10;
5           var zahl2 = 20;
6           var ergebnis = zahl1 + zahl2;
7           alert("Das Ergebnis lautet: " + ergebnis);
8        </script>
9     </head>
10    <body>
11       <p>Erste Anwendung von JavaScript</p>
12    </body>
```

```
13 </html>
```

Listing 3.18: HTML-Seite mit JavaScript-Code zur Addition zweier Zahlen

Um JavaScript direkt in eine HTML-Datei einzubinden, wird das Tag `<script>` benutzt. Da es sich hier um XHTML handelt, gibt es zu dem öffnenden `<script>`-Tag auch ein schließendes Tag. Eine Variable wird in JavaScript mit dem Schlüsselwort var deklariert. Sie kann optional direkt bei der Deklaration mit einem Wert belegt werden, wie es in den ersten beiden Zeilen zu sehen ist. Das Schlüsselwort `var` sollte pro Variablennamen nur einmal verwendet werden, ansonsten würde eine weitere Variable mit demselben Namen generiert. In der dritten Codezeile wird die Addition der beiden Werte 10 und 20 in einer Ergebnisvariablen gespeichert. Eine Möglichkeit der Ausgabe mit JavaScript ist der `alert`-Befehl, der ein Dialogfenster öffnet, das die angegebenen Zeichenketten als Information enthält. Die Zeichenkette kann dabei eine Konkatenation aus mehreren Zeichenketten, Variablen und/oder Ausdrücken sein, die einzelnen Elemente werden mit dem Konkatenationszeichen + zusammengefügt.

Da JavaScript in einigen Punkten der PHP-Skriptsprache ähnelt, werden im Folgenden Variablen, Operatoren und Strukturen nur kurz angesprochen und die Unterschiede hervorgehoben.

Arithmetische Operatoren werden wie in PHP genutzt um mathematische Berechnungen durchzuführen. Die zur Verfügung stehenden Operationen sind: Addition (+), Subtraktion (-), Multiplikation (*), Division (/) und der Modulo-Operator zur Berechnung des Rests einer Division (%).

Wie in PHP gibt es in JavaScript zwei Kontrollstrukturen für Verzweigungen, die `if`-Anweisung und die `switch`-Anweisung. Wie in Codeausschnitt 3.19 zu sehen ist, entspricht die Syntax beider Kontrollstrukturen der in PHP.

```
1  if (zahl == 10) {
2     // Anweisungsblock, wenn erste Bedingung wahr
3  }
4  else if (zahl == 20) {
5     // Anweisungsblock, wenn zweite Bedingung wahr
6  }
7  else {
8     // Anweisungsblock, falls keine Bedingung wahr
9  }
10
11 switch (zahl) {
12    case 10:
13       // Anweisungsblock 1
14       break;
15    case 20:
16       // Anweisungsblock 2
17       break;
18    default:
19       // Anweisungsblock 3
20 }
```

Listing 3.19: Verzweigungen in JavaScript

Auch Schleifen haben den syntaktisch gleichen Aufbau wie Schleifen in PHP (Codeausschnitt 3.20).

```
1 for (zahl = 0; zahl < 20; zahl = zahl + 1) {
2    // Anweisungsblock
3 }
4
5 while (zahl < 20) {
6    // Anweisungsblock
7    zahl = zahl + 1;
8 }
```

Listing 3.20: Schleifen in JavaScript

Eine der besonderen Fähigkeiten von JavaScript ist die Möglichkeit HTML-Code zu manipulieren. Betrachten wir den normalen Verlauf einer Anfrage eines Clients an einen Server: Zunächst sendet der Client eine Anfrage an den Server. Dieser sendet eine HTML-Seite zurück, die entweder als statische Seite auf dem Server liegt oder, z.B. durch ein PHP-Programm, dynamisch erzeugt wurde. Auf dem Client-Rechner wird die erhaltene HTML-Seite anschließend durch einen Webbrowser angezeigt. Auf diese HTML-Seite kann nur noch lokal zugegriffen werden. Da JavaScript eine Client-seitige Skriptsprache ist, ist ein Zugriff auf den HTML Code mit Hilfe eines JavaScript-Programms möglich.

Um die im Folgenden vorgestellten beiden Varianten der Veränderung von HTML-Seiten besser zu verstehen[1], werden vorerst einige Objekte erläutert, die von JavaScript zur Verfügung gestellt werden und die Methoden zur Verfügung stellen, die bei der Arbeit mit JavaScript oft genutzt werden.

Das Objekt `screen` beinhaltet Informationen über den Bildschirm des Seitenbesuchers, z.B. die Höhe ohne die Windows-Taskleiste (`screen.availHeight`), die Breite ohne die Windows Taskleiste, die sich auch am linken oder rechten Bildschirmrand befinden kann (`screen.availWidth`), die Farbtiefe (`screen.colorDepth`), die gesamte Höhe des Bildschirms (`screen.height`) und die gesamte Breite des Bildschirms (`screen.width`).

Das Objekt `window` ist sehr viel umfangreicher und stellt neben einer Vielzahl von Eigenschaften mehrere Methoden zur Verfügung. Es repräsentiert ein geöffnetes Fenster innerhalb eines Webbrowsers. Tabelle 3.4 zeigt einige Attribute und Methoden des `window`-Objektes mit den entsprechenden Beschreibungen.

Das dritte Objekt, das erläutert werden soll, heißt `document`. Das `document`-Objekt repräsentiert das HTML-Dokument, das in den Webbrowser geladen wurde, und stellt Eigenschaften und Methoden zur Verfügung, um auf die Elemente eines HTML-Dokuments mit Hilfe von JavaScript zuzugreifen. Tabelle 3.5 erläutert einige wichtige Eigenschaften und Methoden.

Die erste Variante, eine HTML-Seite mit Hilfe von JavaScript zu verändern, besteht im Hin-

[1] Die innerhalb von JavaScript zur Verfügung gestellten Objekte beinhalten eine Vielzahl von Methoden, den HTML-Code einer Seite zu verändern. Wir beschränken uns auf einige Methoden und Eigenschaften, die häufig genutzt werden.

`window.innerHeight`	stellt die innere Höhe des Fensters dar (ohne Fensterelemente, wie Werkzeugleiste oder Scrollbalken)
`window.innerWidth`	stellt die innere Breite des Fensters dar (ohne Fensterelemente, wie Werkzeugleiste oder Scrollbalken)
`window.screenX`	stellt den Abstand des Fensters zum linken Bildschirmrand dar
`window.screenY`	stellt den Abstand des Fensters zum oberen Bildschirmrand dar
`window.close()`	schließt das aktuelle Fenster
`window.moveTo(100, 150)`	weist dem aktuellen Fenster die X-Position 100 und die Y-Position 150 zu
`window.print()`	druckt den Inhalt des aktuellen Fensters

Tabelle 3.4.: Anwendung des window-Objektes

`document.getElementById("meinElement")`	gibt das HTML-Element mit dem `id`-Attributwert `meinElement` zurück
`document.getElementsByName("anderesElement")`	gibt eine Menge von HTML-Element mit dem `name`-Attributwert `anderesElement` zurück
`document.readyState`	gibt den aktuellen Ladestatus des Dokuments zurück (`uninitialized`, `loading`, `interactive`, `complete`)
`document.title`	gibt den Titel des Dokuments zurück
`document.write("Das ist ein Satz")`	schreibt eine Zeichenkette in das HTML-Dokument; dies kann auch HTML-Code oder JavaScript sein
`document.writeln("Test")`	wie `document.write(...)`, jedoch wird nach der Ausgabe noch ein Zeilenumbruch hinzugefügt

Tabelle 3.5.: Anwendung des document-Objektes

zufügen von Code zu einer bestehenden Seite. Dies kann mit der vom Objekt `document` zur Verfügung gestellten Methode `document.write(...)` geschehen. So kann eine einfache Zeichenkette in das HTML-Dokument eingefügt werden, diese kann jedoch auch HTML- oder JavaScript-Code enthalten. Damit ist es möglich, neue HTML-Elemente zum bestehenden Dokument hinzuzufügen oder zusätzlichen JavaScript-Code einzupflegen. Codeausschnitt 3.21 zeigt, wie der `body`-Bereich einer HTML-Seite mit Hilfe von JavaScript erzeugt wird.

```
1  <html>
2     <head>
3        <script>
4           var datum = "01.01.2014";
5           document.writeln("<h1>Das ist eine Überschrift</h1>");
6           document.writeln("<p>Heute ist der " + datum + "</p>");
7           document.writeln("<a href='http://www.fernuni-hagen.de'>Hier geht
               es zur FernUniversit&auml;t in Hagen </a>");
8        </script>
9     </head>
10    <body>
11    </body>
12 </html>
```

Listing 3.21: Veränderung eines HTML-Dokuments mit document.writeln(...)

Die zweite Variante, eine HTML-Seite zu verändern, wird ebenfalls vom `document`-Objekt zur Verfügung gestellt. Die Methode `getElementById(...)` gibt die Referenz des HTML-Elements mit der angegebenen ID zurück. Ein HTML-Element wiederum verfügt in JavaScript über eine Menge an Eigenschaften und Methoden. Die Eigenschaft `innerHTML` ist eine davon. `innerHTML` kann dazu genutzt werden, den Inhalt eines HTML-Elements auszulesen oder diesen zu setzen. Codeausschnitt 3.22 zeigt, wie bei einem Mausklick der Inhalt eines HTML-Elements ausgelesen, angezeigt und danach geändert wird.

```
1  <html>
2        <head>
3        <script>
4              function inhaltHolen() {
5                 var element;
6                 var inhalt;
7                 element = document.getElementById("inhalt");
8                 alert(element.innerHTML);
9                 element.innerHTML = "Der Inhalt wurde ge&auml;ndert."; }
10       </script>
11       </head>
12       <body>
13          <h2 onclick="inhaltHolen()">Inhalt anzeigen</h2>
14          <p id="inhalt">Das ist der Inhalt, der ausgelesen und ver&auml;
               ndert werden soll</p>
15       </body>
16 </html>
```

Listing 3.22: Veränderung eines HTML-Dokuments mit der Eigenschaft innerHTML

Dieses Beispiel zeigt die Verwendung der Eigenschaft `innerHTML` und verwendet gleichzeitig zwei noch nicht angesprochene Möglichkeiten von JavaScript. Zum einen wird innerhalb des JavaScript-Elements eine Funktion `inhaltHolen` definiert. Funktionen werden wie in PHP ohne Rückgabewert oder Funktionsparametertypen definiert. Innerhalb dieser Funktion werden zwei Variablen element und inhalt deklariert. In der Variable `element` wird die Referenz auf das HTML-Element mit der ID `inhalt` hinterlegt (dies ist das Textabsatzelement `<p>`). Die Methode `alert(...)` erzeugt ein Dialogfenster mit dem Inhalt der Zeichenkette, die an die Methode übergeben wird. Im Beispiel wird dadurch der Inhalt des `<p>`-Elementes ausgegeben. In der darauffolgenden Zeile wird die Eigenschaft innerHTML in anderer Richtung genutzt, um den Inhalt des `<p>`-Elementes neu zu setzen. Der vorherige Inhalt wird damit überschrieben. Um die Funktion `inhaltHolen` aufzurufen, wird das Element `<h2>` um ein Attribut `onclick` erweitert.

JavaScript ist in der Lage auf Nachrichten unterschiedlicher Art zu reagieren. Interaktionen mit HTML-Elementen rufen Ereignisse hervor, die mit JavaScript abgefangen werden können. Dies kann z.B. ein Mausklick auf ein Element sein (`onclick`) oder eine Mausbewegung (`onmousemove`). Die Menge aller möglichen Ereignisse, die mit JavaScript verarbeitet werden kann, wird dabei in die Mengen: Mausereignisse, Tastaturereignisse, Objektereignisse und Formularereignisse unterteilt.

Diese Seite bleibt aus technischen Gründen frei!

4. Technologien[1]

4.1. ASP.NET und Java EE

ASP.NET (Active Server Pages .NET) ist ein freies Web-Framework zur Erstellung von dynamisch erzeugten Internetseiten in Verbindung mit HTML, CSS und JavaScript. Es handelt sich nicht um eine Programmiersprache oder Skriptsprache. Vielmehr ist es eine Server-seitige Programmiermethodik, die von Microsoft entwickelt wurde und mit .NET kompatiblen Programmiersprachen wie z.B. C#, VB.NET oder F# verwendet werden kann. Das .NET-Framework ist ebenfalls eine Entwicklung von Microsoft und stellt neben einer Laufzeitumgebung zur Programmausführung eine Sammlung von Klassenbibliotheken, Progammierschnittstellen und Dienstprogrammen zur Verfügung. ASP.NET unterstützt drei Ansätze für die Erstellung von dynamisch erstellten Internetseiten. ASP.NET Web Forms benutzt Steuerelemente und ein Ereignismodell, um eine komponentenbasierte Entwicklung zu unterstützen. Mit ASP.NET MVC wird das MVC-Muster (Model – View – Controller) und mit ASP.NET Web Pages die Entwicklung von Internetseiten, die eine Komposition aus Programmcode und HTML- Struktur enthalten, unterstützt.

Bei der Java Enterprise Edition (Java EE) handelt es sich wie bei ASP.NET nicht um eine Programmier- oder Skriptsprache. Auch handelt es sich nicht um ein Framework wie ASP.NET. Vielmehr stellt die Java Enterprise Edition eine Sammlung von Spezifikationen für eine Softwarearchitektur dar, die insbesondere für die Entwicklung von Web-basierten Geschäftsanwendungen geeignet ist. Diese Architektur unterteilt sich grob gesehen in vier Schichten: die Client-Schicht, die Web-Schicht, die Business- Schicht und die Persistenzschicht. Version 7 der Java Enterprise Edition wurde im Mai 2013 in der finalen Version veröffentlicht und umfasst insgesamt 47 Spezifikationsdokumente, z.B. für die Spezifikationen „Enterprise Java Beans“, „Standard Tag Library for JSP“, „Java EE Application Deployment“ und die „Java Servlet“-Spezifikation. All diese Spezifikationen beschreiben diverse Funktionalitäten, die mit Hilfe eines Anwendungs-Servers (ein Server-seitiges Programm, das die Java EE-Spezifikationen implementiert) zur Verfügung stehen. Diese umfassen z.B. Sicherheits-Features, Transaktionsmanagement, Kommunikation zwischen Komponenten und Persistenzdienste. Eine ausführliche Betrachtung der Java Enterprise Edition ist Hauptbestandteil dieses Kurses und wird in späteren Kapiteln wieder aufgegriffen.

Es gibt einige wichtige Unterschiede zwischen ASP.NET und Java EE. Java EE ist im Unter-

[1] Der deutsche Begriff Technologie entspricht nicht dem englischen technology und passt eigentlich nirgendwo und überall. Mir ist nicht richtig wohl bei der Verwendung in diesem Kontext, aber der Begriff hat sich etabliert und es gibt keine vernünftige Alternative, die so flexibel einsetzbar wäre.

schied zum .NET Framework und somit auch zu ASP.NET herstellerunabhängig und läuft auf fast allen Betriebssystemen, ist aber sprachabhängig, d.h. es kann lediglich mit der Programmiersprache Java entwickelt werden. ASP.NET ist an Windows angepasst, stellt ein proprietäres Framework dar und ist sprachunabhängig, d.h. .NET Programme können in mehr als 25 Sprachen erstellt werden (z.B. C++, JScript, COBOL, Eiffel, Perl, Python). Aktuell wird ASP.NET für ca. 18% der Internetseiten verwendet, die eine Server-seitige Programmiersprache nutzen. Java als Server-seitige Programmiersprache ist aktuell lediglich auf 2,7% solcher Seiten zu finden (Stand 24.01.2014). (Quellen: [Asp], [Lor], [Jun], [Schi/Schm])

4.2. Ajax

AJAX steht für *Asynchronous JavaScript and XML* und stellt einen neuen Weg dar, vorhandene Internet-Standards zu nutzen. Es handelt sich somit um keine Programmier- oder Skriptsprache wie JavaScript oder PHP. Wie das Akronym AJAX jedoch verrät, handelt es sich um eine Technologie, die etwas mit JavaScript und XML zu tun hat.

Das klassische Modell einer Web-Anwendung sieht vor, dass eine Benutzerin mit Hilfe eines Webbrowsers eine Anfrage an einen Server sendet, um eine gewünschte Internetseite angezeigt zu bekommen. Der Server verarbeitet diese Anfrage und sendet eine HTML-Seite an den Client zurück. Interagiert die Benutzerin mit dieser Internetseite, so muss die Seite nach jeder Interaktion, die z.B. aus dem Absenden eines Formulars oder einem Klick auf einen Link bestehen kann, komplett neu geladen werden. Es wird somit jedes Mal eine komplette HTML-Seite vom Server an den Client zurückgesendet. Im Rahmen dieses klassischen Modells muss die Benutzerin auch stets darauf warten, dass ihre Anfrage komplett bearbeitet wurde. Andere Interaktionen können währenddessen nicht durchgeführt werden. So können Fehlerüberprüfungen eines Formulars nicht direkt geschehen und die Befüllung eines Warenkorbes in einem Online Shop zieht regelmäßig Verzögerungen mit sich, worunter auch die Benutzungsfreundlichkeit solcher Internetseiten leidet.

Mit AJAX erfolgte ein Paradigmenwechsel: Es ist nun möglich, nur Teile einer Internetseite zu aktualisieren, ohne die kompletten Seiten nachzuladen. Um diese Funktionalität umzusetzen basiert AJAX auf entsprechenden Internet-Standards wie dem XMLHttpRequest-Objekt, JavaScript, CSS und XML.

Das XMLHttpRequest-Objekt stellt den Kern von AJAX dar. Jeder moderne Webbrowser besitzt ein solches Objekt. Mit ihm können Daten mit einem Server ausgetauscht werden, und zwar so, dass auch Teile einer Internetseite nachgeladen werden können. Der Nachladeprozess bleibt der Benutzerin, bis auf die Anzeige des Ergebnisses, verborgen.

Auf das XMLHttpRequest-Objekt kann mit der Skriptsprache JavaScript zugegriffen werden. Wir werden ein einfaches Beispielprogramm betrachten, das mit JavaScript ein XMLHttpRequest Objekt erzeugt, mit diesem Objekt eine asynchrone Anfrage (Request) an einen Server sendet und die erhaltene Antwort (Response) auf der Internetseite anzeigt, ohne dass diese komplett neu geladen werden muss.

```
1  <html>
2     <head>
3        <script type="text/javascript">
4           var xmlhttp;
5           xmlhttp = new XMLHttpRequest();
6           xmlhttp.open("POST", "test.txt", true);
7           xmlhttp.send();
8
9           xmlhttp.onreadystatechange=function() {
10             if (xmlhttp.readystate == 4 && xmlhttp.status == 200) {
11                document.getElementById("meinText").innerHtml=xmlhttp.
                     responseText;
12             }
13          }
14       </script>
15    </head
16    <body>
17       <p id= "meinText"></p>
18    </body>
19 </html>
```

Listing 4.1: Ausführung eines XMLHttpRequests

Codeausschnitt 4.1 zeigt den kompletten Code, den wir für unser Beispiel benötigen. Die Seite besteht zum größten Teil aus einem JavaScript-Block. Im Rumpf, also innerhalb des `body`-Tags wird ein `<p>`-Tag mit der ID `meinText` definiert. Den Inhalt dieses Elementes möchten wir mit Hilfe von AJAX verändern. Die ID wird benötigt, damit aus JavaScript heraus auf dieses Element zugegriffen werden kann. Der JavaScript-Block besteht im Wesentlichen aus drei Teilen. In Zeile 4 erstellen wir eine Variable, die das XMLHttpRequest-Objekt aufnehmen soll. In Zeile 5 wird dieses Objekt erzeugt. In Zeile 6 wird zum ersten Mal eine Methode des Objektes benutzt: open. Die `open`-Methode erwartet drei Parameter: die Übertragungsmethode, eine URL und die Angabe, ob die Anfrage asynchron oder synchron gestellt werden soll. Der Methodenkopf hat folgenden Aufbau: `open(method, url, async)`. Als Übertragungsmethode wird POST benutzt. Die URL zeigt in diesem Beispiel auf eine Textdatei, die auf dem Server liegt. Durch den Parameter `async` wird deutlich, dass AJAX auch dazu genutzt werden kann synchrone Anfragen zu erzeugen (wenn async auf false gesetzt wird), in unserem Beispiel möchten wir jedoch eine asynchrone Anfrage erzeugen. Die erzeugte Anfrage wird in der Zeile 7 durch die Methode `send()` des XMLHttpRequest-Objektes an den Server gesendet. Da es sich hier um einen asynchronen Aufruf handelt, wird die Anfrage nun im Hintergrund weiter bearbeitet. JavaScript Programmcode, der auf den Aufruf der `send()`-Methode folgt, wird unmittelbar anschließend ausgeführt. Dies bringt uns zu der Frage, wann und wo wir mit der Antwort unserer Anfrage rechnen können. Die Frage nach dem „wann“ lässt sich pauschal nicht beantworten, da dies immer davon abhängt, welche Verarbeitungsschritte bei der gestellten Anfrage durchgeführt werden müssen (z.B. eine einfache Formularauswertung oder ein aufwendiger Algorithmus in einer PHP-Datei). Die Frage nach dem „wo“ beantworten die nächsten Zeilen unseres Beispielprogramms. Hier wird eine Funktion (ohne Namen) definiert, die dann aufgerufen wird, wenn das Ereignis `onreadystatechange` eintritt. Innerhalb dieser Funktion werden zwei Abfragen durchgeführt. Das Attribut `readystate` des XMLHttpRequest-Objektes gibt Auskunft über

dessen Zustand. Hat das entsprechende Zustandsattribut den Wert „4“, so ist die Anfrage bearbeitet worden und die Antwort steht zur Verfügung. Weiterhin wird geprüft, ob das Statusattribut den Wert 200 (OK) besitzt. Wenn z.B eine Datei oder PHP-Datei, die über die URL der Anfrage festgelegt wurde, nicht gefunden werden konnte, erhält das Attribut `status` den Wert 404 (Seite nicht gefunden . Wurde die Bedingung der Abfrage mit wahr ausgewertet, wird mit Hilfe von JavaScript auf das im Rumpf definierte Textabsatz-Element mit der ID `meinText` zugegriffen. Dazu wird die von JavaScript angebotene Variable `document` genutzt, um die Methode `getElementById()` mit der ID des HTML-Elementes aufzurufen. Diese Methode gibt eine Referenz auf das Element zurück. Anschließend wird der HTML-Code, der sich zwischen dem öffnenden und dem schließenden `p`-Tag befindet, durch den Inhalt der Antwort unserer Anfrage überschrieben. In unserem Beispiel wäre dies der Inhalt der angegebenen Textdatei.

AJAX eröffnet eine ganze Reihe von neuen Möglichkeiten, Internetseiten zu gestalten und eine andere Perspektive auf die Interaktion der Benutzer mit dem Internet einzunehmen. Da es gerade bei größeren Projekten sehr aufwendig werden kann, „manuell“ mit dem XMLHttpRequest-Objekt zu arbeiten, kann man als Entwicklerin auf eine ganze Reihe von Frameworks zurückgreifen, die die Arbeit mit AJAX vereinfachen. jQuery, das im nächsten Abschnitt vorgestellt wird, ist ein solches auf JavaScript basierendes Framework.

Für weitere Aspekte von AJAX verweisen wir auf geeignete Literatur, die auch dem Literaturverzeichnis dieser Kurseinheit entnommen werden kann.

4.3. jQuery

Für die Entwicklung einer Internetseite mit erhöhter Funktionalität und Benutzungsfreundlichkeit hat sich JavaScript als Skriptsprache etabliert. Im vorangehenden Abschnitt wurde mit AJAX eine Technologie vorgestellt, die ein dynamisches Nachladen von Teilen einer Internetseite ermöglicht. jQuery stellt die meistbenutzte JavaScript-Bibliothek dar. Nun ist selbstverständlich die Frage berechtigt, warum eine zusätzliche Bibliothek benötigt wird und ob ein Einsatz einer solchen überhaupt sinnvoll ist. Da jQuery auf JavaScript basiert, wären alle Funktionalitäten die jQuery bietet, doch auch direkt mit JavaScript implementierbar. Der Grund für den Einsatz einer Bibliothek ist meist die Vereinfachung; mit jQuery soll die Benutzung von JavaScript vereinfacht werden, indem es Methoden anbietet JavaScript-Funktionalitäten auf einfache Art und Weise zu nutzen. So sind einige Dinge mit reinem JavaScript nur mühsam umzusetzen, und ein eigener Versuch würde Zeit kosten. Zudem wäre der eigene Programmcode unübersichtlicher und schwerer zu warten. Soll z.B. ein HTML- Objekt animiert werden, könnte die Implementierung einer geeigneten Funktion 30 Programmzeilen benötigen. Eine Bibliothek wie jQuery hat diese Funktionalität bereits implementiert und bietet einfach zu handhabende Funktionen für die Animation des genannten HTML-Objekts an.

Neben der Animation stellt jQuery Methoden zur Verfügung, um Elemente im (X)HTML-Dokumentenbaum mit Hilfe von Selektoren aufzufinden, diese zu manipulieren und Inhalte mit AJAX einzufügen. Darüber hinaus ist mit jQuery ein Event-Handling realisierbar, d.h. Ereignisse (Tastatureingaben, Mausklicks, usw.) können identifiziert und verarbeitet werden. Neben

diesen Methoden für Grundfunktionalitäten kann auch auf das jQuery UI Framework zugegriffen werden, das die Entwicklung von komplexen Benutzungsschnittstellen ermöglicht. Die Bibliothek kann auch durch Entwickler erweitert werden, um Funktionalitäten zu implementieren, die nicht bereits Teil der Bibliothek sind.

Um jQuery nutzen zu können, muss kein zusätzliches Programm installiert werden. Lediglich eine JavaScript-Datei wird benötigt, die von der jQuery-Homepage heruntergeladen werden kann[2]. Der Link „Download the compressed, production jQuery 2.0.3“ öffnet die Datei „jQuery 2.0.3.js“. Der Inhalt dieser Datei ist komprimiert, z.B. wurden Variablennamen minimiert und Zeilenumbrüche und nicht notwendige Leerzeichen entfernt. Zum Vergleich ist in der unkomprimierten Version der JavaScript-Code von jQuery gut lesbar. Um die Performanz bzw. die Ladezeit einer Internetseite zu verbessern, sollte stets die komprimierte Version genutzt werden.

Eine Alternative zum Download der jQuery-Bibliothek ist ihre Einbindung mit Hilfe eines Content Delivery Networks (CDN). Sowohl Google als auch Microsoft bieten ein solches Netzwerk an und stellen die jQuery-Bibliothek zur Verfügung. Die Entwickler von jQuery bieten diese Möglichkeit auch selber an. jQuery kann so bequem durch einen Link in die HTML-Seite eingebunden werden:

```
1 <head>
2    <script src="http://code.jquery.com/jquery-1.11.2.min.js"></script>
3 </head>
```

Listing 4.2: Einbindung von jQuery im Kopfbereich einer HTML-Seite

4.3.1. Syntax

In diesem Abschnitt wird auf die Syntax von jQuery eingegangen. Eine der Hauptfunktionalitäten von jQuery ist die Selektion eines oder mehrerer HTML-Elemente und die Ausführung einer oder mehrerer Funktionen auf diesen Elementen.

Die Basissyntax für diese Funktion sieht folgendermaßen aus:

```
$(selektor).aktion()
```

Das Dollarzeichen $ signalisiert, dass jQuery benutzt werden soll. `selektor` ist ein Ausdruck, der HTML-Elemente „anfragt“ (Anfrage = engl. query) bzw. findet. Bei der Auswertung dieses Ausdrucks kann das Ergebnis der Anfrage nur ein oder auch mehrere HTML-Elemente sein. Die Aktion, die sich hinter der Funktion `aktion()` befindet, wird auf alle durch den `selektor`-Ausdruck gefundenen HTML-Elemente angewandt.

Betrachten wir einige Beispiele:

Wie in Tabelle 4.1 zu sehen ist, benutzt jQuery die Syntax von CSS-Selektoren. Wie in CSS ist es somit auch möglich, innerhalb des Ausdrucks eine Hierarchie zu erstellen. Der Ausdruck

[2] jQuery-Downloadseite: http://jquery.com/download/

`$(this).hide()`	selektiert das aktuelle Element und versteckt es.
`$("div").hide()`	versteckt alle `div`-Elemente in der aktuellen HTML-Seite
`$("#meinElement").hide()`	versteckt nur das HTML-Element mit der ID `meinElement`
`$(".rahmen").hide()`	versteckt alle Elemente die für das Attribut `class` den Wert `rahmen` haben.

Tabelle 4.1.: Beispiele zur Syntax von jQuery

`$("#meinElement.rahmen")` sucht nach allen Elementen mit dem Wert `rahmen` für das `class`-Attribut, die sich in dem Element mit der ID `meinElement` befinden.

Um jQuery in einer Internetseite korrekt anwenden zu können, muss man verstehen, wie eine Internetseite aufgebaut wird bzw. in welcher Reihenfolge die verschiedenen Teile der Seite geladen und/oder ausgeführt werden. Der gesamte Weg von einer Client-Anfrage bis zur Darstellung einer Internetseite im Webbrowser wird in Kapitel 6 näher betrachtet.

Nehmen wir an, wir haben eine HTML-Seite erstellt, die im Rumpf über ein Textabsatzelement verfügt mit der ID `text`. Im Kopf wird die jQuery-Bibliothek mit Hilfe der entsprechenden Referenz eingebunden sowie ein JavaSript-Block mit einer jQuery-Anweisung definiert, die das `p`-Element mit der ID text versteckt.

Codeausschnitt 4.3 zeigt die gesamte HTML-Seite.

```
1  <html>
2     <head>
3        <script src="http://code.jquery.com/jquery-1.11.2.min.js">
4        </script>
5        <script type="text/javascript">
6           $("#text").hide();
7        </script>
8     </head>
9     <body>
10       <p id="text">Das ist ein Text</p>
11    </body>
12 </html>
```

Listing 4.3: Anwendung einer jquery-Methode in JavaScript

Wird diese Seite im Webbrowser angezeigt, ist der Text „Das ist ein Text“ auf der Seite sichtbar, obwohl das Element eigentlich durch die jQuery-Anweisung versteckt werden sollte. Dies liegt daran, dass der JavaScript-Block bereits ausgeführt wird, bevor der Rest der Seite komplett geladen wurde. Der JavaScript-Code hat zu diesem Zeitpunkt noch keine Möglichkeit auf die noch nicht geladenen HTML-Elemente zuzugreifen. Das `p`-Element ist somit noch gar nicht vorhanden, wenn die `hide`-Funktion der jQuery-Anweisung dieses verstecken will. Um zu vermeiden, dass JavaScript-Code ausgeführt wird, solange noch nicht die gesamte HTML-Seite geladen wurde, wird die jQuery-methode `ready()` verwendet (siehe Codeausschnitt 4.4).

```
<html>
   <head>
      <script src="http://code.jquery.com/jquery-1.10.1.min.js">
      </script>
      <script type="text/javascript">
           $(document).ready(function() {
               $("#text").hide();
           });
      </script>
   </head>
   <body>
      <p id="text">Das ist ein Text</p>
   </body>
</html>
```

Listing 4.4: Anwendung einer jQuery-Methode nach dem Ladevorgang

Die `ready`-Funktion wird auf das Objekt `document` angewendet. Das Objekt `document` bezieht sich auf den Inhalt, der in einem Webbrowser-Fenster angezeigt wird. Die Funktion `ready` erwartet als Parameter die Angabe einer JavaScript-Funktion, die den JavaScript-Programmcode enthält, der nach dem Laden der HTML-Seite ausgeführt werden soll. Wird diese Seite nun in einem Webbrowser angezeigt, so ist das `p`-Element nicht mehr sichtbar, da es unmittelbar, nachdem es geladen wurde, durch die jQuery-Methode `hide()` versteckt wird.

4.3.2. Selektoren

Da Selektoren in jQuery zu den wichtigsten Bestandteilen gehören, betrachten wir zunächst verschiedene Typen von Selektoren. Vier Typen von Selektoren wurden in Abschnitt 4.3.1 bereits vorgestellt. Eine ausführlichere Übersicht bietet Tabelle 4.2 [übernommen aus http://www.w3schools.com/jquery/jquery_selectors.asp].

Für eine Gesamtübersicht über alle in jQuery zur Verfügung stehenden Selektoren verweisen wir auf die API-Dokumentation, zu finden auf der jQuery-Homepage: http://api.jquery.com/category/selectors/.

4.3.3. Ereignisse

Unter einem Ereignis (Event) versteht man eine Aktion, die eine Benutzerin einer Webseite ausführen kann und die durch die Programmlogik identifiziert werden kann. Eine Aktion kann unterschiedlicher Natur sein. Z.B. kann eine Benutzerin einen Klick mit der Maus auf ein bestimmtes HTML-Element durchführen, sie kann die Tastatur benutzen, um eine Eingabe zu tätigen oder sie kann durch die Webseite blättern (*scrollen* aus dem engl. *to scroll = blättern, Text rollen*).

jQuery (und auch reines JavaScript) ist in der Lage, solche Ereignisse abzufangen und darauf

`$("*")`	selektiert alle Elemente
`$(this)`	selektiert das aktuelle HTML-Element
`$("p.intro")`	selektiert alle <p>-Elemente mit der Klasse `intro`
`$("p:first")`	selektiert das erste <p>-Element
`$("ul li:first")`	selektiert das erste -Element des ersten -Elements
`$("ul li:first-child")`	selektiert das erste -Element von jedem -Element
`$("[href]")`	selektiert alle Element, die ein `href`-Attribut haben
`$("a[target='_blank']")`	selektiert alle <a>-Elemente mit einem `target`-Attribut, das den Wert `_blank` hat
`$("a[target!='_blank']")`	selektiert alle <a>-Elemente mit einem `target`-Attribut, das nicht den Wert `_blank` hat
`$(":button")`	selektiert alle <button>-Elemente und alle <input>-Elemente mit `type="button"`
`$("tr:even")`	selektiert alle geraden <tr>-Elemente
`$("tr:odd")`	selektiert alle ungeraden <tr>-Elemente

Tabelle 4.2.: Selektoren in jQuery

geeignet zu reagieren. Tabelle 4.3 zeigt einige dieser Ereignisse.

Mausereignisse	**Tastaturereignisse**	**Formularereignisse**	**Fensterereignisse**
`click`	`keypress`	`submit`	`resize`
`dblclick`	`keydown`	`change`	`scroll`
`mouseenter`	`keyup`	`focus`	`load`
`mouseleave`		`blur`	`unload`

Tabelle 4.3.: Ereignisse in jQuery

Die Ereignisse werden kurz beschrieben:

- `click`: Wird ausgelöst, wenn die Benutzerin mit der Maus auf ein HTML-Element klickt.
- `dblclick`: Wird ausgelöst, wenn die Benutzerin mit der Maus einen Doppelklick auf ein HTML-Element ausführt.
- `mouseenter`: Wird ausgelöst, wenn die Benutzerin den Mauszeiger in ein HTML-Element hineinbewegt.
- `mouseleave`: Wird ausgelöst, wenn die Benutzerin den Mauszeiger aus einem HTML-Element herausbewegt.
- `keypress`: Wird ausgelöst, wenn die Benutzerin eine Taste auf der Tastatur betätigt (Drücken und Loslassen)
- `keydown`: Wird ausgelöst, wenn die Benutzerin eine Taste auf der Tastatur herunterdrückt.

- `keyup`: Wird ausgelöst, wenn die Benutzerin eine Taste auf der Tastatur loslässt.
- `submit`: Wird ausgelöst, wenn ein Formular der Benutzerin abgesendet wird.
- `change`: Wird ausgelöst, wenn sich der Wert eines Elements in einem Formular ändert.
- `focus`: Wird ausgelöst, wenn ein Elment in einem Formular den Fokus erhält. Ein Element erhält den Fokus, wenn die Benutzerin mit der Maus in das Element klickt, um z.B. einen Text in ein Textfeld einzugeben.
- `blur`: Wird ausgelöst, wenn ein Element in einem Formular den Fokus verliert.
- `resize`: Wird ausgelöst, wenn die Größe des Webbrowser-Fensters verändert wird (wird auf dem `window`-Objekt aufgerufen).
- `scroll`: Wird ausgelöst, wenn die Benutzerin innerhalb des Elements zu einer anderen Position scrollt.
- `load`: Wird ausgelöst, wenn das Fenster geladen wurde (wird auf dem window-Objekt aufgerufen und erst nach dem `document.ready()`-Ereignis ausgelöst.
- `unload`: Wird ausgelöst, wenn das Fenster geschlossen wird, neu geladen wird oder zu einer anderen Internetseite gewechselt wird.

Um ein Ereignis mit jQuery abzufangen und zu verarbeiten gibt es mehrere Möglichkeiten. Wir beschränken uns hier auf eine einfache Variante: Es werden mit Hilfe eines JQuery-Selektors ein oder mehrere HTML-Elemente ausgewählt. Auf dieser Menge wird anschließend die Funktion des entsprechenden Ereignisses aufgerufen. Diese Ereignisfunktion benötigt als Parameter eine Funktion, die den Programmcode enthält, der beim Eintreten des Ereignisses ausgeführt werden soll. Codeausschnitt 4.5 zeigt eine HTML-Seite, in der jedes `<p>`-Element, auf das mit der Maus geklickt wird, versteckt wird.

```
1  <html>
2     <head>
3        <script src="http://code.jquery.com/jquery-1.10.1.min.js">
4           </script>
5        <script type="text/javascript">
6              $(document).ready(function() {
7                 $("p").click(function() {
8                    $(this).hide();
9                 });
10             }
11       </script>
12    </head>
13    <body>
14       <p>Das ist ein Text</p>
15          <p>Das ist noch ein Text</p>
16          <p>Und noch ein dritter Text</p>
17    </body>
18 </html>
```

Listing 4.5: Nutzung der click-Methode in jQuery

Weiterführende Informationen sind auf der Homepage von jQuery im *Learning Center* unter http://learn.jquery.com/ oder in [Bo/Vo] zu finden.

5. Medien

5.1. Bildformate

Bereits die erste Version von HTML definierte ein Element (Tag) zum Einfügen von Grafiken. Dabei sollte es jedoch nie die Aufgabe von HTML sein, Formate für Bilddateien anzugeben oder Formate zu definieren, die von HTML-Webbrowsern angezeigt werden können. Die Referenzen auf die entsprechenden Bilddateien werden durch HTML gesetzt, die Darstellung der referenzierten Bilddateien wird jedoch durch den Webbrowser gehandhabt.

Der Einsatz von Grafiken auf Internetseiten wurde maßgeblich durch die verfügbare Datenrate geprägt. Zu Zeiten des Telefonmodems (Ende der 1990er Jahre) betrug die maximale Datenrate in privaten Haushalten 56 kbit/s (Kilobit pro Sekunde), heutzutage sind mit einer DSL-Leitung (Digital Subscriber Line) Datenraten von bis zu 100 Mbit/s zu erreichen.

Zum besseren Vergleich dieser beiden Werte:

56 kbit/s = 56.000 bit/s
100 Mbit/s = 100.000.000 bit/s

Gerade in der Anfangszeit des Internet suchte man nach Möglichkeiten die verfügbaren Datenraten bestmöglich auszunutzen. Doch um der Frage nachzugehen, warum solche Methoden überhaupt notwendig waren, betrachten wir näher, um was es sich bei einer digitalen Grafik bzw. einem digitalen Bild handelt.

Digitale Grafiken lassen sich grob in zwei Kategorien einteilen: pixelbasierte Grafiken und vektorbasierte Grafiken. Für die Bildbearbeitung und die Bildverarbeitung[1] ist der Unterschied dieser zwei Kategorien essentiell. Da auch im Internet beide Grafiktypen genutzt werden, werden diese nun näher beschrieben.

Bei pixelbasierten Grafiken besteht ein Bild aus einer Menge von Bildpunkten, die Pixel genannt werden. Jedes Pixel kann einen Farbwert annehmen und besitzt eine bestimmte Position im Bild. Damit ein Computer ein pixelbasiertes Bild auf dem Monitor anzeigen kann, muss er jedes Pixel des Bildes aus der Bilddatei auslesen.

[1] Unter Bildbearbeitung versteht man die Veränderung und Manipulation von Bildern, wie z.B. die Farbanpassung oder das Schärfen von Fotos. Bei der Bildverarbeitung wird im Allgemeinen versucht Informationen aus den Bilddaten zu extrahieren. Dies geschieht meist mit entsprechenden Algorithmen, die auf die Bilddaten zugreifen. So ist es z.B. möglich in einem Foto Gesichter zu erkennen, mit einer Fahrzeugkamera Straßenschilder automatisiert zu deuten oder im medizinischen Bereich automatische Diagnosen von Bildern von Muttermalen zu erstellen.

Die Farbe, die ein Pixel annehmen kann, ist ein Wert aus einem fest definierten Intervall. Dieses Intervall bestimmt die sogenannte Farbtiefe des Bildes, die angibt, wie viele unterschiedliche Farben ein Bildpixel annehmen kann. Bei einem schwarz-weiß-Bild kann jedes Pixel nur entweder schwarz oder weiß sein. Bei Fotos, die mit einer Digitalkamera erstellt wurden, kann jedes Pixel normalerweise ca. 16 Millionen Farben annehmen. Darüber hinaus gibt es auch höhere Farbtiefen für hochwertige Fotografien (ca. 68 Milliarden Farben) oder den Druckbereich (ca. 281 Billionen). Bei Fotos, bei denen jedes Pixel ca. 16 Millionen Farben annehmen kann, spricht man bereits von „True Color“, da solche Fotos bei menschlichen Betrachtern einen natürlichen Eindruck erwecken.

Um eine Grafik auf einem Computer zu speichern, wird Speicherplatz benötigt. Jedes Pixel verbraucht eine bestimmte Menge an Speicherplatz. Die Farbtiefe einer Grafik wird in Bit angegeben und setzt sich bei digitalen Grafiken üblicherweise aus drei Farbkanälen zusammen: Rot, Grün und Blau. Bei einer 24 Bit Farbtiefe stehen für jeden Farbkanal 8 Bit zur Verfügung. Eine 8 Bit Binärzahl kann eine Zahl zwischen 0 und 255 aufnehmen. Somit gilt für die Kombination alle drei Farbkanäle: 256*256*256 = 16.777.216 Farbwerte. Für die Speicherung eines Pixels werden bei einer 24 Bit Grafik 24 Bit Speicherplatz eingenommen. Der gesamte notwendige Speicherplatz eines Bildes setzt sich somit aus der Farbtiefe und der Ausdehnung (Breite und Höhe) des Bildes zusammen, sowie aus einigen zusätzlichen Informationen, die abhängig vom verwendeten Grafikformat sind. Auf diese zusätzlichen Informationen wird in diesem Kurs nicht weiter eingegangen.

Nehmen wir eine Internetseite an, die ein Bild darstellen soll, das 1000 Pixel breit und 600 Pixel hoch ist. Jedes Pixel soll eine von 256 möglichen Farben annehmen können (eine Farbtiefe von 8 Bit) und benötigt somit 8 Bit Speicherplatz. Das gesamte Bild benötigt demnach 1000*600*8 Bit = 4.800.000 Bit. Ein Telefonmodem würde für das Herunterladen dieses Bildes bzw. den Aufruf der Internetseite ca. 86 Sekunden benötigen (4.800.000 bit / 56.000 bit/s = 85.71 s), eine Zeitspanne, die keiner Nutzerin der Seite zumutbar ist. Die Übertragungszeit würde sich verdreifachen, wenn das Bild nicht eine Farbtiefe von nur 8 Bit hätte, sondern eine Farbtiefe von 24 Bit (über 4 Minuten Ladezeit). Bei den möglichen Datenraten heutiger Internetleitungen stellt sich natürlich die Frage, inwiefern solche Dateigrößen ein Problem darstellen, da unsere Grafik bei einer 100 Mbit-Leitung in einem Bruchteil einer Sekunde heruntergeladen werden könnte. Wechselt man jedoch die Perspektive und betrachtet nicht nur die Nutzer, sondern auch die Server, die solche Daten zur Verfügung stellen, wird die Problematik sichtbar. Internet-Plattformen wie *YouTube* oder *flickr* müssen nicht nur ein Bild für eine Nutzerin zur Verfügung stellen, sondern Millionen von Bildern oder Videos für Millionen von Nutzern. Es entstehen somit enorme Datenmengen, die von der Gesamtheit der Nutzer angefragt werden und die von den Servern dieser Plattformen gesendet werden müssen. Auf eine Optimierung der Dateigrößen kann deshalb auch in der heutigen Zeit nicht verzichtet werden.

Bevor auf die Methoden eingegangen wird, wie pixelbasierte Grafiken bezüglich der Dateigröße optimiert werden können, wenden wir uns der zweiten Kategorie von Grafiken zu, den vektorbasierten Grafiken. Im Unterschied zu pixelbasierten Grafiken bestehen vektorbasierte Grafiken nicht aus einzelnen Pixeln, die gespeichert werden müssen, sondern auf Beschreibungen des Inhalts. Stellen wir uns eine Grafik mit den Abmessungen 500*500 Pixel vor, die aus einem

weißen Hintergrund und einem schwarzen Kreis besteht. In einer pixelbasierten Grafikdatei müssten alle 250.000 Pixel gespeichert werden. Die Pixel, die den Hintergrund darstellen, erhalten den Farbwert für „weiß“, die Pixel, die den Kreis darstellen sollen, erhalten den Farbwert für „schwarz“. Bei einer vektorbasierten Grafik würden keine Pixel gespeichert, sondern Befehle, die den Inhalt beschreiben. Diese könnten z.B. wir folgt aussehen:

```
hintergrund-farbe(255, 255, 255);
kreis(250, 250, 100, 0, 0, 0);
```

Die erste Zeile würde den Hintergrund der gesamten Grafik beschreiben und diesem den Farbwert (255, 255, 255) für die Farben Rot, Grün, Blau übergeben. Die zweite Zeile definiert einen Kreis, der seinen Mittelpunkt im Zentrum des Bildes hat (250, 250), über einen Radius von 100 Pixeln verfügt und als Farbwert schwarz (0, 0, 0) übergeben bekommt. Der gesamte Inhalt der Grafik wird mit nur zwei Textzeilen beschrieben (die Ausdehnung der Grafik müsste selbstverständlich in den Metadaten der Datei hinterlegt sein). Der Speicherverbrauch einer vektorbasierten Grafik ist demnach sehr viel geringer als der Speicherverbrauch einer pixelbasierten Grafik.

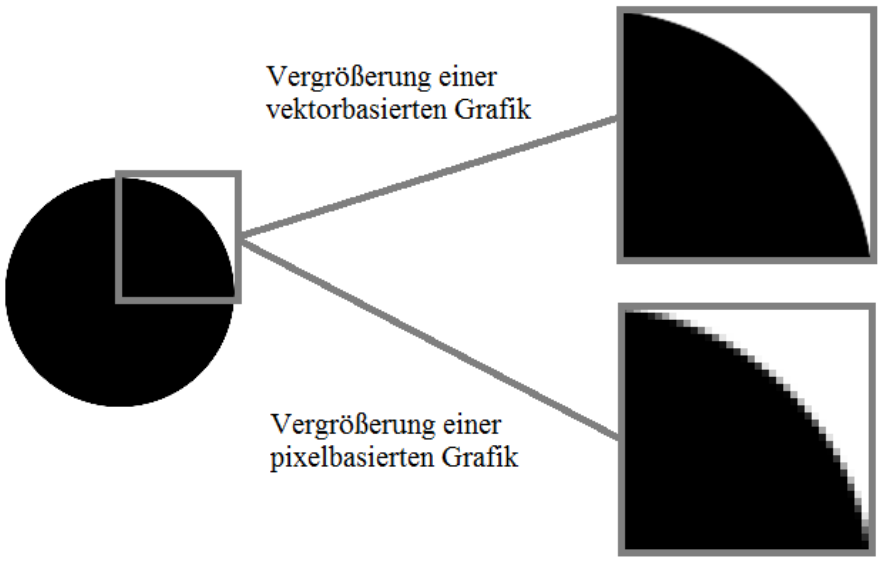

Abbildung 5.1.: Vergrößerung einer Grafik (vektorbasiert und pixelbasiert)

Ein weiterer großer Unterschied zwischen pixelorientierten und vektorbasierten Grafiken wird sichtbar, wenn beide Grafiktypen vergrößert werden, also eine Zoom-Funktion angewandt wird. Die Bildpunkte einer pixelorientierten Grafik bleiben zu jedem Zeitpunkt und zu jeder Zoomstufe erhalten. Sie können lediglich von einem Programm, das die Vergrößerung vornimmt, vergrößert werden. Die Auflösung des Bildausschnittes und somit die Qualität der sichtbaren Grafik nimmt somit ab. Bei einer vektorbasierten Grafik werden alle Pixel basierend auf der Inhaltsbeschreibung der Grafik stets neu berechnet. Die Qualität des ausgewählten und vergrößerten Bildausschnittes bleibt somit zu jedem Zeitpunkt erhalten (siehe Abbildung 5.1).

Als Vorteil von Pixelgrafiken wird die mögliche Anwendung von Filtern genannt. Ein Filter

ist eine Operation, die auf ein Bild angewendet wird und die Pixel des Bildes verändert. So ist es z.B. möglich ein Foto zu schärfen oder zu verzerren. Dies ist bei einer Vektorgrafik nicht möglich, da hier auf keine Pixelinformationen zugegriffen werden kann, sondern lediglich auf die sprachähnlichen Inhaltsbeschreibungen.

Wir haben bereits einige Vor- und Nachteile beider Grafiktypen kennengelernt und fassen diese und weitere Unterscheidungsmerkmale in Tabelle 5.1 zusammen.

	Pixelbasierte Grafiken	**Vektorbasierte Grafiken**
Bildformate	TIF, JPG, BMP, PNG, GIF, ...	EPS, AI, CDR, WMF, ...
Vorteile	programmunabhängiges Dateiformat, Reichtum an Farbabstufungen, Anwendung von Filtern möglich	Skalierung ohne Qualitätsverlust, geringe Dateigröße, nachträgliche Bearbeitung der enthaltenen Formen möglich
Geeignet für	komplexe Bilder, Digitalfotos, gescannte Bilder	Diagramme, Logos, geometrische Figuren und Schriften, technische Zeichnungen

Tabelle 5.1.: Vergleich von Pixelgrafiken und Vektorgrafiken

Welche Methoden gibt es nun, um die Bildgröße von pixelbasierten Grafiken zu verringern und sie für das Internet tauglich zu machen? Die Reduzierung digitaler Daten nennt man Kompression oder Komprimierung. Bei der Komprimierung von Grafiken werden Inhalte zusammengefasst. Meist handelt es sich dabei um Inhalte, die redundant sind, überflüssig sind oder denen kein hoher Stellenwert zugesprochen wird. Insgesamt unterscheidet man zwischen verlustbehafteter und verlustfreier Komprimierung. Bei der verlustbehafteten Komprimierung geht Information verloren. Ein komprimiertes Bild wird im Vergleich zu seinem Originalzustand bei verlustbehafteter Komprimierung weniger Informationen beinhalten, z.B. können Farbinformationen zu einer Menge von Pixeln fehlen. Dadurch leidet die Qualität des Bildes, die immer abhängig ist von der Rate der Kompression (wie stark wird komprimiert). Die verlustfreie Kompression lässt nur Informationen weg, die wirklich überflüssig sind oder kompakter dargestellt werden können. Betrachten wir unser Bildbeispiel mit einem weißen Hintergrund und einem schwarzen Kreis, so ist es möglich dieses Bild ohne Qualitätsverlust zu komprimieren. Dies geschieht, indem nicht jedes Pixel gespeichert wird, sondern nur die Positionen von Pixeln, die einen Farbwechsel definieren. In unserem Beispiel wären dies gerade die Pixel, die den Kreisrand beschreiben, z.B. alle Pixel, deren linker Nachbar eine andere Farbe aufweist. Der Speicherplatz für die Pixel, die sich innerhalb des weißen Hintergrunds oder innerhalb des schwarzen Kreises befinden, entfällt.

Es gibt eine ganze Reihe von Bildformaten für pixelbasierte Grafiken. Drei davon, die sich zur Nutzung im Internet etabliert haben sind JPEG, GIF und PNG. Alle drei Grafikformate lassen sich über das HTML-Tag `<img>` in eine Internetseite einbinden. Anzumerken ist jedoch, dass HTML keine zulässigen Grafikformate definiert und auch nicht definiert, wie Webbrowser mit unterschiedlichen Grafikformaten umzugehen haben. Im HTML-Code wird lediglich eine Referenz auf die entsprechende Grafikdatei gesetzt, für die Darstellung ist der Webbrowser verantwortlich.

In Tabelle 5.2 werden diese drei Bildformate kurz beschrieben. Quelle: [Lub]

	GIF	**JPEG**	**PNG**
Abkürzung für	Graphics Interchange Format	Joint Photographic Experts Group	Portable Network Graphics
Dateiendungen	.gif	.jpg	.png
max. Farben	256	16,7 Millionen	16,7 Millionen
Komprimierung	ja	ja	ja
Qualitätsverlust bei Komprimierung	ja	ja	nein
Animation erstellbar	ja	nein	nein
Geeignet für:	Bilder mit wenigen Farben	Fotos bzw. Bilder mit weichen Farbübergängen	Bilder und Fotos
Transparenz	ja	nein	ja

Tabelle 5.2.: Vergleich von GIF, JPEG und PNG

5.2. Videoformate

Ein Video ist, sehr vereinfacht betrachtet, eine Aneinanderreihung von einzelnen Bildern. In einem fest definierten Zeitintervall wird eine feste Anzahl von Bildern nacheinander angezeigt. Das menschliche Gehirn nimmt eine Reihe von Bildern als bewegte Szene dar, wenn mindestens ca. 16-18 Bilder pro Sekunde angezeigt werden. In Europa haben DVDs zum Beispiel eine Bildfrequenz (Anzahl Bilder, die pro Sekunde angezeigt werden) von 25Hz und sind somit dafür konzipiert 25 Einzelbilder pro Sekunde anzuzeigen.

Wie bereits im vorigen Abschnitt beschrieben wurde, hat jedes Bild einen gewissen Speicherbedarf. Dieser wächst mit der Größe und der Farbtiefe des Bildes, kann jedoch durch Komprimierung verringert werden. Ein Video, das 5 Minuten lang ist, beinhaltet 5 (Minuten) * 60 (Sekunden) * 25 Bilder = 7.500 Bilder. Hätte das Video eine Größe von 1000 * 600 Pixeln mit einer Farbtiefe von 24 Bit pro Pixel, so würde das Video einen Speicherplatz von ca. 12,6 GB (GigaByte) benötigen. Nun könnte eine Besucherin einer Internetseite, die dieses Video zur Verfügung stellt, solange warten, bis das gesamte Video heruntergeladen wurde und es dann lokal auf ihrem Computer betrachten. Um lange Wartezeiten zu vermeiden, hat es sich jedoch etabliert Videos zu „streamen“ (Streaming = kontinuierliche Übertragung von Daten). Das bedeutet, dass immer neue Teile des Videos geladen werden, während die Besucherin dieses bereits abspielt. Für unser Rechenbeispiel würde dies immer noch bedeuten, dass ca. 43 MB pro Sekunde übertragen werden müssten. Um solche Datenmengen zu vermeiden und Besuchern ein unterbrechungsfreies Video schnell liefern zu können, wurden auch für Videos verschiedene Komprimierungsverfahren entwickelt.

Videos werden zum einen durch ihr Format (wie es bei Grafiken auch der Fall ist) und zusätzlich durch einen sogenannten Videocodec beschrieben. Das Videoformat legt die Spezifikation des Videos fest und gibt Informationen über die Auflösung, die Bildfrequenz, die Farbtiefe und die Tonspur. Unter dem Videocodec versteht man ein Verfahren, das für die digitale Kodierung und Dekodierung der Daten im Video zuständig ist. Es handelt sich um ein Paar von Algorithmen, ein Algorithmus für die Kodierung und der andere für die Dekodierung. Der Codec übernimmt zusätzlich die Komprimierung und Dekomprimierung des Videos.

In Tabelle 5.3 werden die im Internet gängigsten Videoformate und Videocodecs vorgestellt. Dabei ist zu beachten, dass es viele Kombinationsmöglichkeiten von Formaten und Codecs gibt. Das Videoformat kann oft als Container verstanden werden, der verschiedene Codecs aufnehmen kann.

Format	Dateiendung	Codec
FLV (Flash Video)	.flv	H.264
WebM	.webm	VP9
MPEG-4	.mp4	H.264
OGV	.ogv, .ogg	Theora

Tabelle 5.3.: Videoformate

Vor der Einführung von HTML 5 war es nicht möglich ein Video direkt in eine Internetseite einzubinden. Dies gelang nur durch die Einbindung eines Plug-Ins für den Webbrowser, das Videos in Internetseiten einbettet. Die bekanntesten Plug-Ins dieser Art sind der Adobe Flash Player und das QuickTime-Plug-In. Seitdem jedoch der HTML Standard in Version 5 vorgestellt wurde, ist auf Internetseiten, die diesen Standard nutzen, ein installierter Flash Player nicht mehr notwendig. HTML 5-kompatible Webbrowser können die in dieser Version neu eingeführten HTML-Tags verstehen und mit dem Tag `<video>` auf einfache Art und Weise eine Videodatei in eine HTML-Seite einbinden. Codeausschnitt 5.1 zeigt, wie ein Video mit dem Namen `einVideo` in eine HTML-Seite eingebunden werden kann.

```
1  <html>
2     <head>
3
4     </head>
5     <body>
6     <video controls>
7        <source src="einVideo.webm" type="video/webm">
8        <source src="einVideo.mp4" type="video/mp4">
9        Ihr Webbrowser unterstützt die verfügbaren
10             Videoformate nicht oder ist nicht HTML5 kompatibel.
11    </video>
12    </body>
13 </html>
```

Listing 5.1: Einbindung eines Videos in eine HTML 5-Seite

In diesem Beispiel wird das HTML 5-Tag `<video>` genutzt. Wird das Schlüsselwort `controls`

hinzugefügt, wird im Video-Element eine Steuerungsleiste zur Steuerung des Videos angezeigt (Zeitleiste, Start, Stop, Lautstärke). Wird innerhalb des Video-Elements mehr als eine Quelle angegeben, wie es in unserem Beispiel der Fall ist, so wird nacheinander geprüft, ob die jeweilige Quelle vorhanden ist bzw. vom Web-browser unterstützt wird. Zuerst wird also das .webm-Video geprüft und, wenn dies fehl schlägt, anschließend das .mp4-Video. Ist eine Prüfung erfolgreich, werden die nachfolgenden Elemente ignoriert. Ist keine der Prüfungen erfolgreich, wird der Text selbst innerhalb des Video-Elements angezeigt. (Quelle: [Schmidt])

Diese Seite bleibt aus technischen Gründen frei!

6. Der Webbrowser

Der Webbrowser (oder auch Internetbrowser, Browser) ist ein Computerprogramm, das zur Anzeige von Dokumenten und Daten, speziell zur Anzeige von Internetseiten konzipiert wurde. Der Webbrowser hat sich als zentrales Programm für den Zugang zum World Wide Web etabliert. In den letzten Jahren ist auch die Nutzung von Internetbrowsern für mobile Geräte kontinuierlich gestiegen. In diesem Kapitel werden einige Informationen zu den aktuell am meist verbreiteten Browsern sowie deren gemeinsame, grundlegende Funktionsweise erläutert.

6.1. Verbreitung moderner Webbrowser

Aktuell gibt es fünf führende Webbrowser für Desktop-Computersysteme: Google Chrome, Microsoft Internet Explorer, Mozilla Firefox, Apple Safari und Opera der Firma Opera Software ASA. Tabelle 6.1 zeigt deren Verbreitung über die letzten zwei Jahre und das prozentuale Wachstum an (Oktober 2012 bis Oktober 2014).

Webbrowser	Nutzung Okt 2012	Nutzung Okt 2014	Wachstum
Chrome	35,93%	51,07%	+42%
Internet Explorer	33,29%	21,61%	-35%
Safari	5,19%	4,95%	-4%
Firefox	23,15%	18,36%	-21%
Opera	1,65%	1,37%	-17%
Andere	0,79%	2,64%	+234%

Tabelle 6.1.: Verbreitung von Webbrowsern (Quelle: gs.statcounter.com)

Die Wachstumsrate berechnet sich folgendermaßen:

$$w = \frac{aktuellerWert - vorherigerWert}{vorherigerWert}$$

Gibt es z.B. insgesamt 10000 PCs, so nutzen 0.79% (79 PCs) einen anderen Webbrowser im Oktober 2012 und 2.64% (264 PCs) einen anderen Webbrowser im Oktober 2014. Die Wachstumsrate beschreibt die Differenz dieser beiden Werte: 79 PCs * 2.34 = 185 PCs = 264 PCs - 79 PCs.

Webbrowser	Nutzung Okt 2012	Nutzung Okt 2014	Wachstum
Android	25,84%	21,03%	-19%
iPhone	20,87%	22,85%	+9%
Opera	18,92%	9,08%	-52%
UC Browser	7,65%	8,97%	+17%
Nokia	10,20%	3,11%	-70%
Chrome	0.52%	28,1%	+5287%
BlackBerry	4,02%	1,11%	-72%

Tabelle 6.2.: Verbreitung von mobilen Webbrowsern (Quelle: de.statistica.com)

Seit der Einführung mobiler Geräte mit Internet-Zugriff, wie Smartphones und Tablets, sind auch Webbrowser für diese Geräte relevant. Tabelle 6.2 zeigt eine Übersicht der mobilen Webbrowser (bzw. Plattformen) über die letzten beiden Jahre (Oktober 2012 bis Oktober 2014).

6.2. Grundlegende Funktionsweise

Moderne Webbrowser sind schon lange keine einfachen Programme mehr, in die ausschließlich eine Adresse einzutippen ist, eine Anfrage an einen Server gestartet wird und die zurückgegebene Antwort als HTML-Seite dargestellt wird. Die Implementierung eines Webbrowsers umfasst oft mehrere Millionen Codezeilen und resultiert in einem Programm, das neben der Anzeige der angefragten Internetseite Sicherheits-Features anbietet, favorisierte Internetseiten strukturiert, durch zusätzliche Programme erweiterbar ist und für Entwickler eine ganze Reihe von Werkzeugen mitbringt, um die entsprechende Seite auf unterschiedliche Aspekte und mit Hilfe unterschiedlicher Sichten (HTML-Code, JavaScript, CSS-Eigenschaften, Ladezeiten, Ressourcen, usw.) zu untersuchen.

In diesem Abschnitt möchten wir die grundlegende Funktionsweise eines Webbrowsers beleuchten und soweit abstrahieren, dass sie möglichst unabhängig von einer konkreten Implementierung ist, d.h. eine differenzierte Betrachtung konkreter Webbrowser wie Chrome oder Internet Explorer findet nicht statt.

Die Hauptkomponenten eines Webbrowsers sind:

- Die Benutzungsschnittstelle
- Das Browser-Modul
- Das Rendering-Modul
- Das Netzwerk-Modul
- Das UI-Backend (UI = User Interface - Benutzungsschnittstelle)
- Ein JavaScript-Interpreter

- Ein XML-Parser
- Ein Datenspeicher zur persistenten Speicherung von Daten

Abbildung 6.1 stellt diese Komponenten schematisch dar.

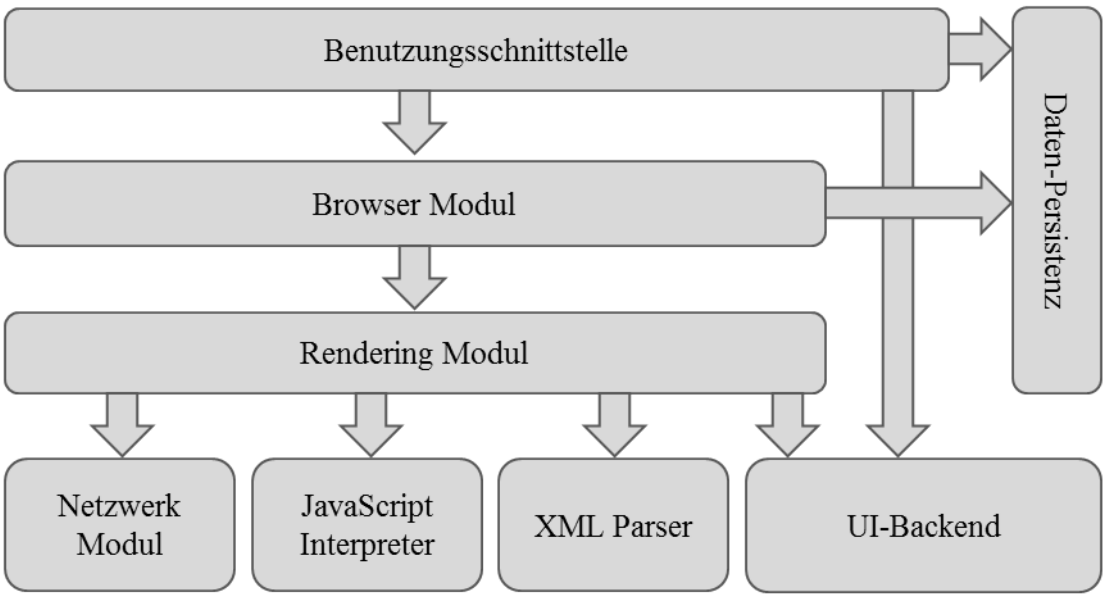

Abbildung 6.1.: Architektur eines Webbrowsers (Quelle: [Gross])

Die Benutzungsschnittstelle wird maßgeblich durch die sichtbaren Elemente des Webbrowsers definiert, mit denen eine Benutzerin interagiert. Diese Elemente umfassen z.B. die Adresszeile, in die eine URL eingetragen werden kann, sowie das Programmmenü, Werkzeugleisten und zusätzliche Schaltflächen wie die Zurück- oder die Vorwärts-Schaltfläche. Der Inhalt einer HTML-Seite ist nicht Teil der Benutzungsschnittstelle eines Webbrowsers. Werden Aktionen durch eine Benutzerin ausgeführt, kann die Benutzungsschnittstelle auf diese reagieren und die entsprechenden Ereignisse an das Browser-Modul, das als Vermittler zwischen der Benutzungsschnittstelle und dem Rendering-Modul dient, weitergeben.

Wird eine Internetseite von einer Benutzerin über die Benutzungsschnittstelle in das Adressfeld des Webbrowsers eingegeben und somit eine Anfrage erzeugt, ist es die Aufgabe des Netzwerk-Moduls entsprechende Netzwerkaufrufe, wie eine HTTP-Anfrage, durchzuführen.

Wurde eine HTML-Seite von einem Server an den Webbrowser zurückgesendet, so ist das Rendering Modul dafür verantwortlich die Internetseite in dem entsprechenden Bereich des Webbrowsers darzustellen[1]. Im ersten Schritt wird das erhaltene HTML-Dokument geparst. Ein Parser (engl. to parse = analysieren) ist ein Programm, das eine Eingabe in ein zur Weiterverarbeitung brauchbares Format umwandelt. Da es sich bei einem HTML-Code generell gesehen nur um eine lange Zeichenkette handelt, muss diese geparst und in eine geeignete Struktur umgewandelt werden. Diese Struktur wird Document Object Model (DOM) genannt. JavaScript

[1] Firefox nutzt z.B. das Rendering-Modul *Gecko*, Safari und Chrome verwenden beide *WebKit*.

und jQuery verwenden geeignete Befehle (Selektoren), um auf einzelne HTML-Elemente innerhalb des DOM mit einer ID oder einem entsprechenden `class`-Attribut zuzugreifen. Durch die Hinzunahme von CSS-Anweisungen innerhalb von HTML-Elementen oder innerhalb externer CSS-Dateien wird das DOM in eine weitere Struktur, die Rendering-Struktur, umgewandelt. Die Rendering-Struktur umfasst nicht nur abstrakte Element-Informationen, sondern setzt sich aus Rechtecken mit visuellen Informationen zusammen, die prinzipiell direkt auf der Darstellungsfläche des Webbrowsers ausgegeben werden könnten. „Prinzipiell“, da im Zeichenprozess zusätzlich auf das Modul „UI-Backend“ zugegriffen wird, das Informationen zu Standard-HTML-Elementen bereitstellt. So wird z.B. die Darstellung von Schaltflächen oder Texteingabefeldern durch das Betriebssystem beeinflusst, diese Standard-Elemente haben meist das gleiche Aussehen wie die entsprechenden Elemente des Betriebssystems. Das Rendering-Modul versucht die Inhalte einer Internetseite so schnell wie möglich darzustellen und wartet dabei nicht, bis das gesamte Dokument geparst worden ist.

Enthält die HTML-Seite zusätzlich JavaScript-Code, so muss dieser ebenfalls geparst und entsprechend interpretiert werden, wofür der JavaScript-Interpreter zuständig ist. Ein Interpreter ist ein Programm, das in JavaScript definierte Befehle in Befehle umwandelt, die vom Computersystem direkt verstanden und ausgeführt werden können. Dabei wird im Unterschied zu einem Compiler das Programm nicht im Vorhinein komplett umgewandelt, sondern Schritt für Schritt, während es ausgeführt wird.

Das Persistenz-Modul ist dafür zuständig, Daten, die vom Webbrowser gespeichert und zu einem späteren Zeitpunkt wieder benötigt werden, lokal auf dem Computersystem zu speichern. Zu diesen Daten gehören Cookies, in denen Informationen gespeichert werden (z.B. ob die Benutzerin sich im aufgerufenen Online-System bereits authentifiziert hat) oder auch Ressourcen wie Grafiken, die lokal abgelegt werden, um ein späteres, erneutes Herunterladen aus dem Internet zu vermeiden. (Quellen: [Gross], [Gar/Iri])

Dieser Abschnitt konnte Ihnen nur eine grobe Vorstellung davon geben, wie ein Webbrowser intern funktioniert. Selbstverständlich ist die Menge an Schritten, die wirklich ausgeführt werden, um eine Internetseite abzurufen und darzustellen, weitaus größer. Bei weiterem Interesse an dieser Thematik findet man entsprechende Quellen im Literaturverzeichnis.

Kurseinheit 3

Java EE: Servlets und JavaServer Pages

Die vorliegende dritte Kurseinheit beginnt in Kapitel 7 mit einem Überblick über die *Java Platform, Enterprise Edition (Java EE)*, die eine Standardarchitektur für Java-basierte Anwendungen spezifiziert. Der Kurs bezieht sich auf Version 7 der Java EE Spezifikation und setzt diese für die Entwicklung von Web-Anwendungen ein. Die Verzeichnisstruktur der Web-Schicht von Java EE schließt das Kapitel ab.

Der Rest dieser Kurseinheit ist den Konzepten *Servlet* und *JavaServer Pages (JSP)* gewidmet. Kapitel 8 beschreibt neben den Aufgaben eines Servlet einen typischen Request-Ablauf bei Verwendung von Servlets. Kapitel 9 beginnt mit einer Motivation des JSP-Konzepts und einem typischen Request-Ablauf mit einem Servlet und einer JSP-Seite. Anschließend geht es um Techniken zur Behandlung von dynamischen Inhalten in einer JSP durch *Scripting-Elemente*, die *Expression Language* und die *JavaServer Pages Standard Tag Library*. Kapitel 8 und 9 werden jeweils abgerundet durch ein ausführliches Beispiel.

Kapitel 10 schließt die Kurseinheit mit Hinweisen zur Verwendung von Servlets und JSP-Seiten im Sinne der Grundsätze des Software Engineerings ab.

Diese Seite bleibt aus technischen Gründen frei!

7. Einstieg

7.1. Java EE im Überblick

Die *Java Platform, Enterprise Edition (Java EE)* ist die Spezifikation einer Standardarchitektur für Java-basierte Anwendungsprogramme, die von der Firma Sun Microsystems herausgegeben wurde und seit der Übernahme von Sun Microsystems durch die Firma Oracle im Jahr 2010 von dieser weiterentwickelt wird. In diesem Kurs werden wir Java EE nur für die Entwicklung von Web-Anwendungen einsetzen. Dabei sei erwähnt, dass viele Technologien von Java EE auch für Java Desktop-Anwendungen genutzt werden können und nicht zwangsweise auf ein Web-Umfeld angewiesen sind.

Unter dem Begriff „Spezifikation“ (aus dem mittellateinischen: specificatio = Auflistung, Verzeichnis) versteht man im Allgemeinen die „Gesamtheit von Vorgaben, nach denen etwas produziert wird “[1]. Im Software Engineering ist eine Spezifikation mit einer Anforderungsdefinition gleichzusetzen. Sie gibt Informationen darüber, was ein Modul, eine Software oder eine Architektur für Anforderungen (Eigenschaften) erfüllen muss, ohne konkrete Auskünfte darüber zu geben, *wie* diese Anforderungen in einer späteren Phase des Softwareentwicklungsprozesses umgesetzt und implementiert werden.

Die Spezifikation Java EE umfasst 39 Dokumente (einzelne Dokumente besitzen mehr als 500 Seiten), die in sechs Kategorien eingeteilt sind[2]. Im Unterschied zu Anforderungsdokumenten, die keine konkrete Implementierung adressieren, ist die Java EE-Spezifikation auf die Programmiersprache Java ausgerichtet und enthält sehr viele Code-Vorgaben und Code-Beispiele. Eine Code-Vorgabe kann z.B. ein Code-Ausschnitt sein, der eine Java-Schnittstelle definiert, die mit ihrer (in der Spezifiation noch offenen) Implementierung eine bestimmte Funktionalität zur Verfügung stellen soll.

Die Architektur von Java EE orientiert sich an einer Schichtenarchitektur. Grundlegend werden drei Schichten unterschieden:

- der Client-Computer
- der Java EE-Server
- der Datenbank-Server

[1] Quelle www.duden.de, Stichwort: Spezifikation

[2] Diese Kategorien sind: Java EE Platform, Web Application Technologies, Enterprise Application Technologies, Web Services Technologies, Management and Security Technologies und Java EE-bezogene Spezifikationen in Java SE, siehe: http://www.oracle.com/technetwork/java/javaee/tech/index.html

Bei einem Web-Anwendungsprogramm verwenden Benutzer einen Web-Browser, der auf ihrem Rechner installiert ist (Client). Der Web-Browser stellt über das Internet Anfragen (Requests) an den Java EE-Server und erhält Antworten (Responses), dies können z.B. HTML-Seiten sein. Bei Bedarf greift der Java EE-Server auf einen Datenbank-Server zu, um dort gespeicherte Daten zu laden, diese ggf. zu bearbeiten und in seine Antwort an den Client einfließen zu lassen.

Der Java EE-Server ist in zwei weitere Schichten unterteilt: die Web-Schicht und die Schicht der *Geschäftslogik*. An dieser Stelle sollen die Begriffe „Geschäftslogik“ (engl. business logic) und „Geschäftsanwendung“ (engl. business application) erläutert werden, die im Zusammenhang mit Java EE immer wieder auftauchen werden. Java EE wurde als Softwarearchitektur für Geschäftsanwendungen konzipiert. Eine Geschäftsanwendung bezeichnet grob ein Anwendungsprogramm, das ein Unternehmen in seinen Aufgaben unterstützt und im Allgemeinen zu groß und zu komplex für sehr kleine Unternehmen oder Einzelpersonen ist. Die Schicht der Geschäftslogik (engl. business tier) innerhalb des Java EE-Servers beinhaltet die Software-Komponenten, die die Geschäftslogik umsetzen. Eine solche Komponente kann z.B. die Anforderung realisieren, dass vor dem Verkauf eines Produkts die Daten der Kreditkarte des Kunden geprüft werden. Die Geschäftslogik wird im weiteren Kurs auch Anwendungslogik genannt. Die Web-Schicht (engl. web tier) des Java EE-Servers enthält alle Software-Komponenten, die Anfragen von einem Client entgegennehmen, Antworten in Form von Web-Seiten erzeugen und an den Client zurücksenden. Diese Komponenten werden auch Web-Komponenten genannt. Unter einer Software-Komponente versteht man dabei eine funktional abgeschlossene Softwareeinheit, die aus Java-Klassen und -dateien besteht und Teil eines Java EE-Anwendungsprogramms ist. Unterschiedliche Komponenten können miteinander kommunizieren.

Wir wissen nun, dass es sich bei der Java EE-Spezifikation um eine Sammlung von Dokumenten und Vorgaben handelt, jedoch nicht, wie ein Java EE-Anwendungsprogramm ausgeführt wird. Hier kommt der Begriff des Application Server (Anwendungs-Server) ins Spiel. Ein *Application Server* bezeichnet in der Java EE-Welt keinen physischen Rechner, sondern eine Software, die auf einem Server installiert ist und eine Java EE-Spezifikation implementiert. Sie umfasst somit alle durch die Spezifikation gegebenen Anforderungen und ist in der Lage ein entwickeltes Java EE-Anwendungsprogramm auszuführen.

Die Java EE-Spezifikation definiert vier Bereiche, die *Container* genannt werden. Zwei dieser Container sind Teil des Application Server, die anderen beiden Container befinden sich auf dem Client-Rechner. Jeder dieser vier Container basiert auf der Java Standard Edition (Java SE) und stellt eine Laufzeitumgebung für verschiedene Komponenten bereit. Abbildung 7.1 zeigt die Struktur dieser vier Container. Die jeweils in einem Container dargestellten Komponenten werden durch den Container verwaltet.

Wir skizzieren kurz die vier Container und deren Komponenten, genauere Erklärungen folgen später:

- Der *Web Container* ist für die Web-Komponenten *Java Servlet* , die *JavaServer Faces Technologie (JSF)* und die *JavaServer Pages Technologie (JSP)* zuständig. Wie auch bei den anderen Containern sind die Komponenten von den später real verfügbaren Klassen, Objekten und Seiten zu unterscheiden. So verwaltet der Web Container die JSP-Technolgie,

während der Ausführung eines Java-EE Anwendungsprogramms kann es jedoch mehrere auf JSP basierende Internetseiten geben, die durch den Web Container verwaltet werden. Es können auch mehrere Klassen als *Servlet*[3] definiert und beliebig viele Objekte dieser Klassen erzeugt werden. All diese Elemente (aufbauend auf der entsprechenden Technologie) dienen der Bearbeitung von Anfragen, die von einem Client gesendet werden, und der Erzeugung der jeweiligen Antworten in Form von Web-Seiten. Benötigen diese Elemente Funktionalitäten aus der Geschäftslogik, so können Sie auf Elemente des *EJB Containers* zugreifen.

- Der *EJB Container* (EJB steht für *Enterprise Java Beans*) ist für die EJB Technologie zuständig und verwaltet alle auf dieser Technologie basierenden Elemente. Dies sind Klassen, die als Enterprise Java Bean[4] definiert werden, und die von diesen Klassen erzeugten Objekte. EJB-Klassen sind vergleichbar mit Kontrollklassen (s. [SE 1]) und implementieren die Anwendungslogik eines Java EE-Anwendungsprogramms.

- Der *Applet Container* ist für die Verwaltung von *Applets* zuständig. Applets bieten für Web-Anwendungen graphische Oberflächen und können via HTTP mit den Elementen des Web Containers kommunizieren. In ihrer Funktionalität entsprechen sie daher GUI-Anwendungen[5]. Der Applet Container befindet sich auf dem Client-Rechner und besteht aus einem Browser mit zugehörigem Java-Plugin.

- Ein *Application Client* bezeichnet eine klassische Java GUI-Anwendung auf dem Client-Rechner. Der *Application Client Container* umfasst eine Menge von Java-Klassen, Bibliotheken und weitere Dateien, die benötigt werden um eine Java-Anwendung auszuführen. Auch diese Anwendungsprogramme können mit Elementen des Web Containers, des EJB Containers oder direkt mit einer Datenbank kommunizieren.

Für die genaue Beschreibung der einzelnen Bestandteile wird in der Java EE-Spezifikation häufig auf weitere Spezifikationen wie die *Servlet-Spezifikation* [Serv/Spec], die *JSP-Spezifikation* [JSP/Spec], die *Java SE-Spezifikation* [JSE/Spec] oder die *EJB-Spezifikation* [EJB/Spec] verwiesen.

Die Behandlung von persistenten Anwendungsobjekten spezifiziert der *Java Persistence* genannte Teil der Java EE Spezifikation [JPA/Spec]. Ein persistentes Anwendungsobjekt kann z.B. ein Objekt einer Klasse sein, das einen konkreten Kunden im System darstellt. Solche Objekte werden auch *Entities* genannt und sind vergleichbar mit Instanzen von Entitätsklassen (s. [SE 1]). Entities werden wie EJB-Objekte durch den EJB Container verwaltet.

Die wichtigsten Elemente innerhalb des Web Containers sind Servlets und JavaServer Pages. Ein Servlet (s. Kapitel 8) ist eine spezielle Java-Klasse, welche die Abarbeitung einer von einem Client gestellten Anfrage steuert. Fordert die Anfrage Ergebnisse an, die nur mit Wissen der Anwendungslogik geliefert werden können, ruft das Servlet dazu die entsprechenden EJB-Objekte

[3] Eine erste Beschreibung, worum es sich bei einem Servlet handelt, erfolgt etwas später in diesem Abschnitt.

[4] Eine Java Bean ist eine Java-Klasse mit bestimmten Eigenschaften. Es muss ein öffentlicher Standardkonstruktor vorhanden sein, die Klasse muss serialisierbar sein, und es müssen öffentliche Zugriffsmethoden für die Attribute der Klasse vorhanden sein (Getter- und Setter-Methoden).

[5] GUI = Graphical User Interface (graphische Benutzungsoberfläche)

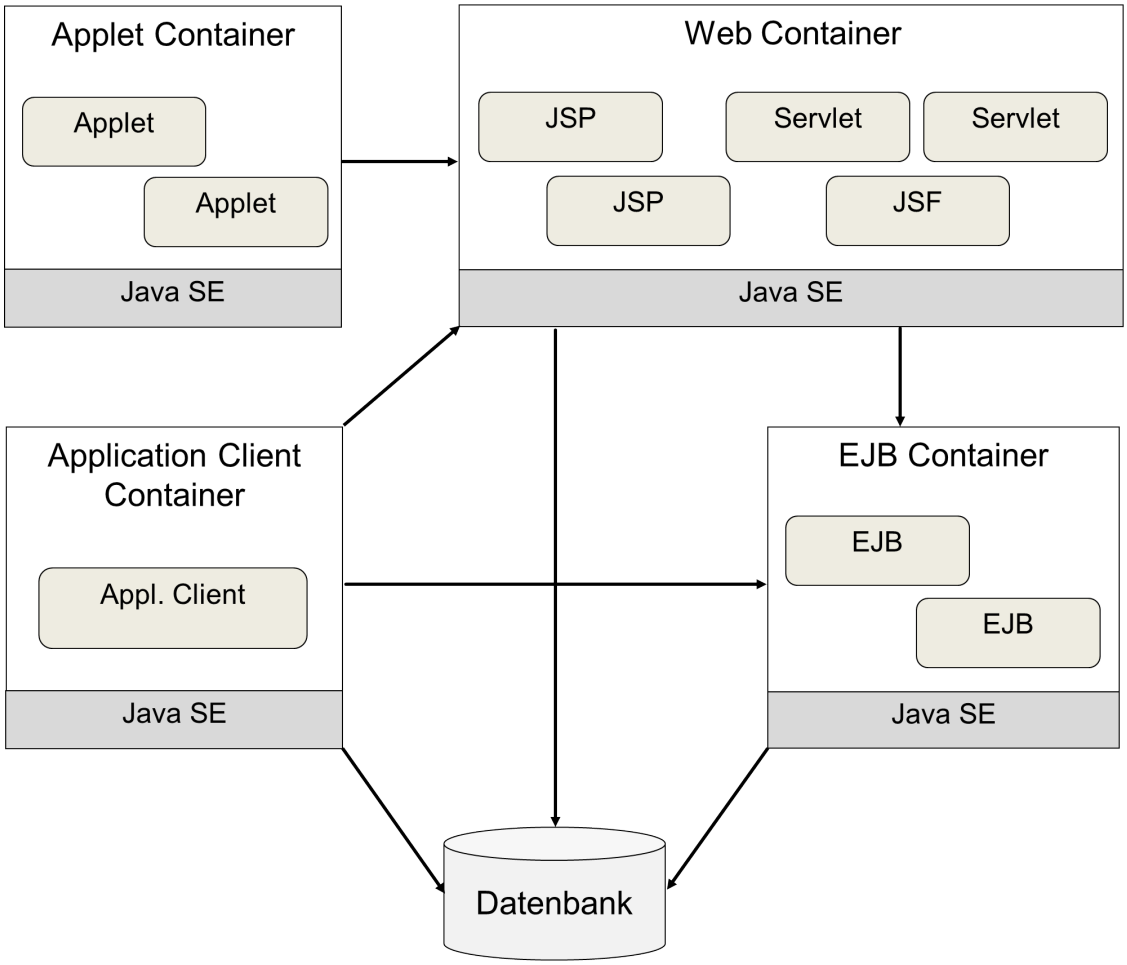

Abbildung 7.1.: Container in Java EE

(auch EJBs genannt) auf. Abschließend erstellt in der Regel eine JSP-Seite (s. Kapitel 9) aus dem dynamisch erzeugten Inhalt und dem statischen Inhalt ein Ausgabedokument, das dann der Web Container an den Client sendet. Der statische Anteil wird im Format des Ausgabedokuments (HTML, XML usw.) angegeben, zur Einbettung des dynamisch erzeugten Inhalts werden verschiedene Techniken angeboten.

Da die Java EE-Spezifikation frei verfügbar ist, gibt es mittlerweile mehrere Implementierungen, sprich Application Server, die teils kommerziell, teils als Open Source angeboten werden. Verschiedene Application Server können in den Punkten, die die Java EE-Spezifikation nicht behandelt oder offen lässt, voneinander abweichen. Dies betrifft insbesondere die Konfiguration der Server. Oracle bietet im Open-Source-Projekt *Glassfish* [GFish] eine Referenzimplementierung unter dem Namen *Oracle GlassFish Server Open Source Edition* (Version 4.0) an, die auch Teil des *Java EE 7 SDK* ist. SDK ist die Abkürzung für Software Development Kit. Ein *Software Development Kit* umfasst eine Menge von Werkzeugen, mit denen ein Anwendungsprogramm entwickelt werden kann, und beinhaltet oftmals auch Dokumentationen, die die Handhabung dieser Werkzeuge und Anwendungen beschreiben. EJB Container und Web Container können

von unterschiedlichen Herstellern stammen. So entwickelt z.B. das *Apache Jakarta Project* den Web Container *Tomcat* [Tom] ständig weiter. Ein anderer, sehr verbreiteter Application Server ist der *JBoss Application Server*, der von der Firma Red Hat entwickelt und betreut wird.

7.2. Verzeichnisstruktur der Java EE-Webschicht

Ein Java EE-Anwendungsprogramm besteht aus einer Vielzahl von Dateien gleichen und unterschiedlichen Typs. Der Dateityp unterscheidet sich z.B. für HTML-Dateien, XML-Dateien und Java-Dateien, und man kann den Dateityp an der jeweiligen Dateinamenserweiterung erkennen. Die Java EE-Spezifikation gibt vor, wie die Verzeichnisstruktur, in die alle benötigen Dateien eingeordnet werden, auszusehen hat. Wir betrachten zunächst lediglich die Verzeichnisstruktur der Web-Schicht, die Teil des Java EE-Servers ist und somit den Server-seitigen Teil der Benutzungsschnittstelle ausmacht. Die Web-Schicht enthält neben Dateien, die Servlet-Klassen oder Hilfsklassen enthalten[6] , JSP-Seiten, JSF-Seiten, sowie statische HTML- und Bilddateien. In Kurseinheit 6 wird die Verzeichnisstruktur der Anwendungslogik als zweite Schicht des Java EE-Servers vorgestellt, die Dateien mit EJB- und Entity-Klassendefinitionen enthält.

Damit der Web Container des Application Servers auf die Dateien der Web-Schicht zugreifen kann, müssen Entwickler darauf achten, die Verzeichnisstruktur für die Web-Schicht auf Ihrem Rechner so zu realisieren, wie durch die Java EE-Spezifikation vorgegeben wird. Bevor das entwickelte Java EE-Anwendungsprogramm, d.h. die realisierte Verzeichnisstruktur dieses Programms mit allen zugehörigen Dateien, auf den Server kopiert wird, wird die gesamte Verzeichnisstruktur in einem *Archiv* zusamengefasst. Ein Archiv ist eine Datei, die als Container für mehrere Dateien fungiert und diese ggf. komprimiert, um den Speicherbedarf zu reduzieren. Ein Java EE-Anwendungsprogramm, d.h. die Verzeichnisstruktur mit den entsprechenden Dateien, kann zu einem *JAR*-Archiv (*Java ARchive*) oder einem *WAR*-Archiv (*Web ARchive*) gepackt werden. Ein WAR-Archiv ist ein JAR-Archiv mit der Restriktion, die für ein Java EE-Anwendungsprogramm benötigte Verzeichnisstruktur einzuhalten. Beide Archivtypen basieren auf dem *ZIP*-Dateiformat (engl. zipper = Reißverschluss), das sich als Archivformat auf Rechnern etabliert hat. Um die verschiedenen Archivtypen voneinander unterscheiden zu können und deren Einsatzzweck zu definieren, ist für JAR-Archive die Dateinamenserweiterung „.jar“ und für WAR-Archive „.war“ vorgesehen. Der Vorteil eines Web Archives besteht in der einfachen Verteilung (Distribution) bzw. Weitergabe eines Anwendungsprogramms, besonders dann, wenn sein Verzeichnis sehr viele Dateien enthält.

Das Installieren, Konfigurieren und Ausführen eines Java EE-Anwendungsprogramms wird auch als *Deployment* bezeichnet. In Tabelle 7.1 ist die Verzeichnisstruktur[7] einer Java EE-Webanwendung angegeben. Es ist darauf zu achten, dass zwischen Groß- und Kleinbuchstaben

[6] Überlicherweise wird für jede Klasse eine Datei erzeugt, die den Namen der enthaltenen Klasse erhält.

[7] Während der Entwicklung wird meist nicht direkt innerhalb des Web-Anwendungsverzeichnisses des Web Container gearbeitet, sondern – meistens mittels einer integrierten Entwicklungsumgebung (kurz IDE für Integrated Development Environment) – eine eigene Verzeichnisstruktur verwendet, die durch die Entwicklungsumgebung in der Phase des Deployment in ein vom Application Server nutzbares JAR- oder WAR-Archiv umgewandelt wird.

unterschieden wird.

Verzeichnis bzw. Datei	Inhalt
`/Name`	Der Name der jeweiligen Web-Anwendung ist zugleich ihr Wurzelverzeichnis. Bei Distribution als WAR tritt der Name nicht in der Verzeichnisstruktur auf, sondern wird als Name für das Archiv herangezogen.
`/Name/WEB-INF/`	In diesem Verzeichnis befinden sich alle Dateien (inkl. der Klassendefinitionen), die nur aus der Anwendung heraus sichtbar sind. Ein Client (Browser) hat auf dieses Verzeichnis sowie seine Unterverzeichnisse im Gegensatz zu einem Servlet keinen direkten Zugriff.
`/Name/WEB-INF/classes/`	Hier sind die übersetzten Java-Klassen abgelegt.
`/Name/WEB-INF/lib/*.jar`	Hier werden Java-Bibliotheken abgelegt, die von der jeweiligen Web-Anwendung genutzt werden.
`/Name/WEB-INF/web.xml`	Dies ist die Konfigurationsdatei (*Deployment Descriptor*) der Web-Anwendung. Sie ist im *XML-Format* verfasst und seit der Java EE Version 5 optional.
`/Name/META-INF/`	Hier werden bei WAR-Archiven (wie bei JAR-Dateien) Metadaten über das Archiv und die darin enthaltenen Dateien gespeichert.
`/Name/*`	Alle anderen Verzeichnisse und Dateien sind direkt über HTTP adressierbar und wie statische Inhalte abrufbar, z.B. eine HTML-Seite.

Tabelle 7.1.: Verzeichnisstruktur einer Java EE-Web-Anwendung

7.3. Konfiguration einer Java EE Web-Anwendung

Eine *Konfiguration* ist eine Menge von Parametern, Eigenschaften und manchmal auch Regeln, die ein Anwendungsprogramm näher beschreiben bzw. die ein Anwendungsprogramm zu befolgen hat. Wird der Code eines Anwendungsprogramms auf einem Server abgelegt, so sollten bestimmte Konfigurationselemente nicht direkt im Programmcode geändert werden, insbesondere wenn sie mehrfach vorkommen. Dies gilt verstärkt, wenn es sich um ein kompiliertes Programm handelt, dessen Programmcode nicht zur Verfügung steht oder nach einer Änderung nicht mit dem kompilierten Programm übereinstimmt.

Oft sollen aber Parameter, Eigenschaften oder das Verhalten eines Anwendungsprogramms zur Laufzeit geändert werden können. All diese Daten werden in einer Konfigurationsdatei festgehalten. In Java EE entspricht der sogenannte *Deployment-Descriptor* einer solchen Konfigurationsdatei. Dies ist eine XML-Datei mit dem Dateinamen „web“ und der Dateinamenserweiterung „.xml“ (web.xml), die Java EE-spezifische Parameter, Eigenschaften und Regeln definiert.

Seit der Java EE Version 5 gibt es eine weitere Möglichkeit, Konfigurationen vorzunehmen: *Annotationen*. Beide Varianten werden in den nächsten beiden Abschnitten näher beschrieben. Durch die Verwendung von Annotationen wird der Deployment Descriptor in den meisten Fällen obsolet. Werden sowohl Annotationen als auch ein Deployment Descriptor zur Konfiguration einer Web-Anwendung verwendet, werden die durch Annotationen gesetzten Einstellungen durch entsprechende Einstellungen des Deployment Descriptors überschrieben und somit ersetzt.

7.3.1. Der Deployment Descriptor web.xml

Mit dem Deployment Descriptor können die folgenden Bereiche von Web-Anwendungen konfiguriert werden:

- ServletContext-Initialisierungsparameter
- Session-Konfiguration
- Servlet-Deklarationen
- Servlet-Mappings
- Filter-Definitionen
- Filter-Mappings
- Welcome File List
- Fehlerseiten

Der Deployment-Descriptor ist in XML verfasst. Alle Konfigurationen (d.h. die zu der jeweiligen Konfiguration gehörenden Tags) müssen sich zwischen dem Anfangs-Tag `<web-app>` und dem End-Tag `</web-app>` befinden. Zusätzlich befindet sich in der ersten Zeile der XML-Prolog (siehe Kurseinheit 1).

Den nach der Servlet-Spezifikation 3.1 definierten „Rahmen“ der Datei web.xml zeigt Codeausschnitt 7.1.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <web-app xmlns="http://java.sun.com/xml/ns/javaee"
4           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5           xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
6               http://java.sun.com/xml/ns/javaee/web-app_3_1.xsd"
7           version="3.1">
8
9      .
10     .
11     .
12
13 </web-app>
```

Listing 7.1: Aufbau des Deployment Descriptors web.xml

An Stelle der drei Punkte können verschiedene Bereiche konfiguriert werden. Exemplarisch werden die Servlet Mappings, die Servlet Deklarationen und die Welcome File List besprochen.

Das Verzeichnis /WEB-INF sowie seine Unterverzeichnisse sind für Client-Anfragen nicht erreichbar. Servlets[8], die für die Bearbeitung von Benutzeranfragen konzipiert sind, liegen im von außen nicht zugänglichen Verzeichnis /WEB-INF/classes. Für alle Servlets, die direkt von einem Client angesprochen werden sollen, müssen daher im Deployment Descriptor Abbildungsregeln von Anfrage-URLs auf Servlets eingetragen werden. Dafür sind für jedes Servlet zwei Einträge notwendig:

Codeausschnitt 7.2 zeigt den ersten Eintrag (im einfachsten Fall), der ein Servlet deklariert.

```
1 <servlet>
2     <servlet-name>MusterServlet</servlet-name>
3     <servlet-class>servletsammlung1796.MusterServlet</servlet-class>
4 </servlet>
```

Listing 7.2: Servlet-Deklaration im Deployment Descriptor web.xml

Über `<servlet-name>` wird dem Servlet ein beliebiger in der Web-Anwendung eindeutiger Name zugewiesen, hier „MusterServlet“. Mit `<servlet-class>` wird der Klassenname des Servlets inklusive Paketstruktur angegeben.

Der zweite Eintrag legt die Abbildungsregel fest:

```
1 <servlet-mapping>
2    <servlet-name>MusterServlet</servlet-name>
3    <url-pattern>/start</url-pattern>
4 </servlet-mapping>
```

Listing 7.3: Servlet Mapping Element im Deployment Descriptor `web.xml`

Mit `<servlet-name>` wird das entsprechende Servlet im ersten Eintrag angesprochen. Mit `<url-pattern>` wird die URL (hier: `/start`) angegeben, unter der das Servlet gefunden werden soll. Dabei bezieht sich die Pfadangabe auf das Wurzelverzeichnis der Web-Anwendung. Wenn die Web-Anwendung in dem Verzeichnis „Kurs1796Apps“ liegt und auf dem Host „www.FernUni-Hagen.de“ läuft, dann kann das gerade deklarierte Servlet über

`http://www.FernUni-Hagen.de/Kurs1796Apps/start`

erreicht werden. Im `<url-pattern>`-Eintrag kann als URL auch ein *URL-Muster* wie z.B. *.do angegeben werden. Dieses bedeutet, dass alle URLs der Web-Anwendung, die auf „.do“ enden, von diesem Servlet bearbeitet werden. Seit der Servlet-Spezifikation Version 2.5 dürfen auch mehrere `<url-pattern>`-Einträge in einem `<servlet-mapping>` stehen.

[8] Klasse, die als Servlet definiert wurde

Mit der Konfiguration der *Welcome File List* können Startseiten der Web-Anwendung festgelegt werden, die zurückgeliefert werden, wenn der Client keine explizite Startseite angefordert hat, sondern lediglich die URL `http://.../Webanwendung` angibt. Dann wird vom Web Container automatisch die erste verfügbare Ressource[9] in der Welcome File List ausgewählt. Um z.B. das oben deklarierte Servlet als Startseite festzulegen, genügt der in Codeausschnitt 7.4 dargestellte Eintrag in der Welcome File List[10].

```
1 <welcome-file-list>
2     <welcome-file>start</welcome-file>
3 </welcome-file-list>
```

Listing 7.4: Welcome File List Element im Deployment Descriptor web.xml

7.3.2. Konfiguration durch Annotationen

Durch *Annotationen* lassen sich in der Programmiersprache Java Meta-Informationen in den Quelltext einbeziehen. Dabei wird zwischen zwei Arten von Annotationen unterschieden, Annotationen, die die Semantik des Programmcodes ändern und Annotationen, die die Semantik nicht ändern (wie z.B. die `@Override` Annotation). Annotationen, die die Semantik nicht beeinflussen, beinhalten Informationen über den Programmcode selber. Annotationen werden im Quelltext vor Klassen, Methoden oder Atrributen notiert und durch ein „@“ Zeichen als Annotation gekennzeichnet, auf das der Typ der Annotation folgt. Anschließend kann eine in runden Klammern eingeschlosse Liste von Parametern folgen, die durch Kommata getrennt werden. Jedem angegebenen Parameter kann innerhalb der Annotation ein Wert zugewiesen werden. Dies geschieht, indem zuerst der Parameter bezeichnet und danach, getrennt durch ein Gleichheitszeichen, der Parameterwert angegeben wird.

Durch den *Annotationstyp* wird festlegt, in welchem Kontext die Annotation benutzt werden darf (z.B. nur vor Klassen oder Methoden), welche Parameter (Name und Typ) angegeben werden können oder müssen, wie die Standardwerte für nicht angegebene Parameter lauten etc. Auf der Basis dieser Informationen nimmt der Compiler beim Übersetzen des Quellcodes auch eine Gültigkeitsprüfung vor. Abgesehen von den Standard-Annotationstypen aus dem Java-Paket `java.lang` (wie `Override`) müssen Annotationstypen – genau wie Klassen oder Interfaces – importiert werden, damit sie in einer Java-Datei genutzt werden können.

Die Annotationen werden vom Java-Compiler zusammen mit dem Quellcode verarbeitet, und die darin enthaltenen Meta-Informationen werden mit in die compilierten Java-Klassen eingebunden. Dabei haben diese Meta-Informationen zwar keinen direkten Einfluss auf den annotierten Code, sie werden jedoch zur Laufzeit von anderen Codeteilen (z.B. einem Framework) durch

[9] Eine Ressource ist alles, was über eine URI identifiziert werden kann (KE 1, Abschnitt 2.2).

[10] Im Normalfall genügt ein einziger Eintrag im Welcome File. Mehrere Einträge werden verwendet, falls verschiedene Unterverzeichnisse des Webanwendungs-Stammverzeichnisses per URL angesprochen werden können und in diesen Verzeichnissen jeweils unterschiedliche Startseiten liegen, z.B. „index.html“ in „/A“ und „index.jsp“ in „/B“.

Introspektion[11] ausgelesen. So kann z.B. der Web Container benötigte Meta-Informationen über Servlets direkt aus den Annotationen in Servlet-Klassen auslesen, statt sie aus einer separaten XML-Datei (Konfigurationsdatei) beziehen zu müssen. Indirekt können Annotationen also durchaus Einfluss auf das Verhalten einer Software haben.

Im Rahmen von Java EE gibt es zwei Kategorien von Annotationstypen: vordefinierte und selbstdefinierte Annotationstypen. Ein Application Server, der die Java EE-Spezifikation implementiert, umfasst eine ganze Reihe vordefinierter Annotationstypen, die direkt genutzt werden können und Java EE-spezifische Parameter zur Verfügung stellen. Darüber hinaus können Entwickler eigene Annotationstypen definieren, deren Parameterwerte durch Introspektion ausgelesen werden können. Vordefinierte Annotationen werden meist bei der Kompilierung ausgewertet, selbstdefinierte Annotationen zur Laufzeit eines Anwendungsprogramms.

In Codeausschnitt 7.5 ist ein Beispiel für die Verwendung der `Override`-Annotation aus dem Paket `java.lang` dargestellt.

```
1 public class MeineKlasse extends MeineOberklasse {
2    @Override
3    public boolean methode1(String param1) {
4       ...
5    }
6 }
```

Listing 7.5: Override Annotation

Diese Annotation hat keinerlei Auswirkungen auf das Programmverhalten, sodass sich das Programm mit und ohne die `Override`-Annotation identisch verhält. Durch die Annotation wird der Compiler veranlasst zu überprüfen, ob die Methode `methode1` der Unterklasse `MeineKlasse` die Methode `methode1` der Oberklasse `MeineOberklasse` überschreibt. Sollte es in der Oberklasse keine entsprechende Methode geben, gibt der Compiler beim Übersetzen eine Fehlermeldung aus. Diese Annotation dient also der Vermeidung von Programmierfehlern.

Im Rahmen dieses Kurses wird zur Konfiguration von Servlets lediglich die Annotation `@WebServlet` benötigt und besprochen[12]. Die `@WebServlet` Annotation deklariert eine Java-Klasse als Servlet. Die einzigen Parameter, für die Parameterwerte angegeben werden müssen, sind `value` und `urlPatterns`[13]. Ist kein Parameterwert für `value` angegeben, muss ein Parameterwert für `urlPatterns` angegeben werden und umgekehrt. Es dürfen aber nicht für beide Parameter zugleich Parameterwerte angegeben werden. Beiden Parametern kann als Wert nur eine nichtleere Liste von URLs zugewiesen werden, über die das Servlet angesprochen werden soll. Dabei bezieht sich die Pfadangabe auf das Wurzelverzeichnis der Web-Anwendung. Wenn kein anderer Parameter angegeben wird, sollte der Parameter `value` und ansonsten `urlPatterns` verwendet werden. Des Weiteren lässt sich über den optionalen Parameter `name` dem Servlet ein Name zuweisen.

[11] Introspektion bezeichnet in der Programmierung das Konzept, dass ein Programm seine eigene Struktur kennt und diese ggf. verändern kann.

[12] Eine vollständige Auflistung und Dokumentation aller Servlet-Annotationen findet sich in [Serv/Spec].

[13] Eine vollständige Auflistung und Dokumentation aller Parameter der `@WebServlet` Annotation findet sich unter [JEE/API] im Paket javax.servlet.annotation.

Die Codeausschnitte 7.6[14] und 7.7 stellen jeweils ein Beispiel für eine Klasse `MusterServlet` dar, welche durch die Annotation `@WebServlet` als Servlet deklariert wird. Wie die Servlet-Klasse genau aufgebaut sein muss, wird in Abschnitt 8.4 erläutert.

```
1  package kurs01796.ke3.servletbeispiele
2
3  import java.io.IOException;
4  import javax.servlet.ServletException;
5  import javax.servlet.annotation.WebServlet;
6  import javax.servlet.http.HttpServlet;
7  import javax.servlet.http.HttpServletRequest;
8  import javax.servlet.http.HttpServletResponse;
9
10 @WebServlet(value={"/start","/musterServlet"})
11 public class MusterServlet extends HttpServlet {
12    protected void doGet(HttpServletRequest request, HttpServletResponse
          response) throws ServletException, IOException {
13          ...
14    }
15    ...
16 }
```

Listing 7.6: Servlet Deklaration durch Annotation @WebServlet

```
1  package kurs01796.ke3.servletbeispiele
2
3  import java.io.IOException;
4  import javax.servlet.ServletException;
5  import javax.servlet.annotation.WebServlet;
6  import javax.servlet.http.HttpServlet;
7  import javax.servlet.http.HttpServletRequest;
8  import javax.servlet.http.HttpServletResponse;
9
10 @WebServlet(name="MusterServlet", urlPatterns={"/start","/musterServlet"})
11 public class MusterServlet extends HttpServlet {
12    protected void doGet(HttpServletRequest request, HttpServletResponse
          response) throws ServletException, IOException {
13          ...
14    }
15    ...
16 }
```

Listing 7.7: Servlet Deklaration durch Annotation @WebServlet mit mehreren Parametern

Wenn wieder davon ausgegangen wird, dass die Web-Anwendung in dem Verzeichnis „Kurs1796Apps“ liegt und auf dem Host „www.FernUni-Hagen.de“ läuft, kann das Servlet „MusterServlet“ in beiden Beispielen über die URLs

[14] Der Parameterbezeichner und das Gleicheitszeichen können weggelassen werden, wenn kein weiterer Parameter angegeben wird: `@WebServlet("/start","/musterServlet")`.

`http://www.FernUni-Hagen.de/Kurs1796Apps/start` und
`http://www.FernUni-Hagen.de/Kurs1796Apps/musterServlet`

erreicht werden. Während in Codeausschnitt 7.7 dem Servlet der Name „MusterServlet“ zugewiesen und daher der Parameter `urlPatterns` verwendet wird, wird in Codeausschnitt 7.6 als einziger Parameter `value` verwendet. Das Servlet erhält in Codeausschnitt 7.6 standardmäßig den voll qualifizierten Klassennamen als Namen: `... .servletbeispiele.MusterServlet`.

8. Servlets

Die *Servlet Technologie* ist neben der JavaServer Pages Technologie (JSP) die wichtigste Technologie. Ein *Servlet* ist eine spezielle Java-Klasse, welche die Abarbeitung der Anfragen (Requests) steuert, die von einem Client an den Server gesendet werden. Zunächst nimmt der Web Container den Request entgegen und gibt dann die relevante Information in objektorientierter Form an das Servlet weiter. Fordert der Request Ergebnisse an, die nur mit Wissen der Anwendungslogik geliefert werden können, ruft das Servlet dazu die entsprechenden Methoden von EJB-Klasseninstanzen auf. Das zurückgelieferte Ergebnis (Response) kann das Servlet selbst ausgeben oder einer JSP-Seite zur Verfügung stellen, die daraus ein Ausgabedokument für den Client erstellt.

8.1. Request-Ablauf

Dieser Abschnitt gibt einen Überblick über die Aufgaben, die von einem Servlet und dem Web Container übernommen werden, wenn ein Client eine Request-Nachricht an den Java EE-Server sendet. Wie später noch zu sehen ist, sollte stets eine JSP-Seite für die Ausgabe genutzt werden, da dies eine bessere Trennung von Programmlogik und Ausgabe bietet. Den Ablauf eines Standard-Requests mit Verwendung einer JSP-Seite für die Ausgabe behandelt Abschnitt 9.1. Um zum Einstieg den Ablauf aber möglichst einfach zu halten, verzichten wir hier zunächst auf den Einsatz einer JSP-Seite, d.h. das Servlet übernimmt auch deren Aufgabe.

Das Sequenzdiagramm in Abbildung 8.1 stellt den Ablauf eines Standard-Requests dar. Dabei wird davon ausgegangen, dass der Browser per POST-Methode ein Formular abschickt und als Antwort ein HTML-Dokument geliefert wird. Der Web Container und der Anwendungskern sind hier in einfacher Form dargestellt. Auch werden die meisten Antwortnachrichten (gestrichelte Linien mit nicht gefüllter Pfeilspitze) zur besseren Übersicht nicht dargestellt.

Zunächst nimmt der Web Container den HTTP-Request, der vom Client gesendet wird, entgegen. Er wandelt ihn in ein entsprechendes `HttpServletRequest`-Objekt um, das die Request-Informationen in objektorientierter Form aufnimmt. Neben diesem gefüllten Anfrageobjekt erzeugt er auch ein leeres Antwortobjekt vom Typ `HttpServletResponse`, das später vom Servlet gefüllt wird. Das Antwortobjekt stellt ein Objekt der Klasse `PrintWriter` bereit, in das später das Servlet das als Antwort zu sendende Dokument mittels der Methode `println(...)` schreibt. Zwar könnte dieser Methode der gesamte Inhalt der zu sendenden HTML-Seite in Form einer einzigen Zeichenkette übergeben werden, aus Gründen der besseren Übersicht sollten allerdings mehrere `println(...)` Methodenaufrufe verwendet werden. Der

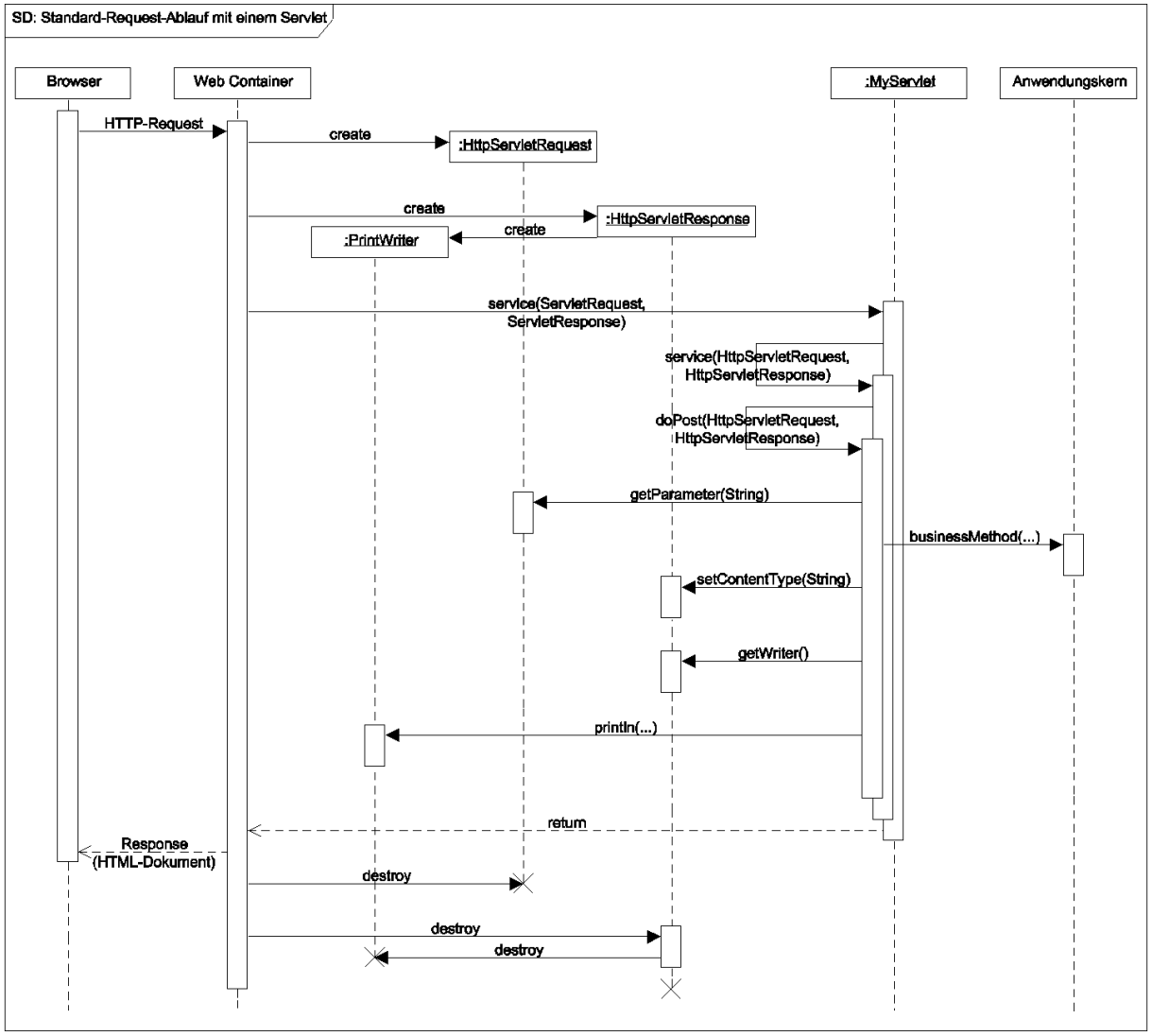

Abbildung 8.1.: Ablauf eines Standard-Request

Web Container ruft anschließend mit dem Anfrage- und Antwortobjekt als Parameter die als `public` deklarierte `service(...)`-Methode des Servlet-Objekts der Klasse MyServlet auf. Diese Methode steuert die als `protected` deklarierte `service(...)`-Methode des Servlet an. Diese dient dem Servlet als eine Art Dispatcher (engl. dispatch „abschicken, abfertigen“), der die Anfrage an die der HTTP-Übertragungsmethode entsprechenden `doXXX()`-Methode weiterleitet (XXX steht für `Post`, `Get`, `Delete` etc.). Im Beispiel aus Abbildung 8.1 wird an die `doPost(...)`-Methode weitergeleitet. Diese Methode liest mit `getParameter(String)` einen in dem Request übermittelten Parameter (z.B. die Eingaben der Benutzerin in den Formularfeldern) aus dem Anfrageobjekt. Dieser Aufruf muss für jeden übermittelten Parameter wiederholt werden. Der Einfachheit halber ist im Sequenzdiagramm nur ein Aufruf dargestellt. Anschließend wird, ggf. mit den erhaltenen Parametern, der Anwendungskern aufgerufen. Nach dessen Ausführung beginnt mit dem zurückgelieferten Ergebnis die Erzeugung des entsprechenden HTML-Dokuments. Dazu wird zunächst mit `setContentType(String)` das Dokument-

format für die Antwort gesetzt – im Fall unseres Beispiels `text/html`. Anschließend wird mit `getWriter()` das `PrintWriter`-Objekt geholt, in dem mittels `println(...)` das HTML-Dokument erstellt wird. Auch hier werden üblicherweise mehrere `println(...)`-Aufrufe nötig sein, der Einfachheit halber ist aber nur ein Aufruf dargestellt. Die Response wird schließlich vom Web Container an den Browser übergeben. Abschließend werden das Anfrage- und das Antwortobjekt zerstört.

Die Schnittstellen und Klassen `HttpServletRequest`, `HttpServletResponse` und `PrintWriter` sowie die abstrakte Klasse `HttpServlet` werden gemäß der Servlet-Spezifikation vom Web Container zur Verfügung gestellt. Die Entwicklerin muss lediglich das Servlet (hier: `MyServlet`) als Spezialisierung von `HttpServlet` implementieren und die `doXXX(...)`-Methoden des Servlet programmieren (s. Abschnitt 8.4).

8.2. Aufgabe, Lebenszyklus, Thread-Sicherheit

Ein Servlet ist eine Java-Klasse, die von der abstrakten Klasse `HttpServlet` abgeleitet ist. Oft wird der Begriff Servlet aber auch für eine Instanz einer Servlet-Klasse gebraucht. Wenn keine Missverständnisse möglich sind, soll auch in diesem Kurs nicht zwischen Klassen und Instanzen von Servlets unterschieden werden. Ein Servlet bietet die Möglichkeit, je nach Bedarf die folgenden Aufgaben auszuführen[1]:

- *Die vom Client gesendeten Daten lesen*: Das sind insbesondere die Daten, die die Benutzerin in Formularfeldern eingegeben hat.
- *Die vom Client gesendeten HTTP Request Header lesen*: Diese Daten werden vom Web-Server bestimmt. Dazu gehören unter anderem Cookies, Statuscodes und vom Browser unterstützte Medienformate.
- *Ergebnisse erzeugen*: Das Servlet ruft den Anwendungskern auf, um Berechnungen, Datenbank-Zugriffe usw. durchführen zu lassen.
- *Die HTTP Response Header setzen*: Das Servlet kann explizit Dokumententyp, Statuscodes, Cookies und andere Informationen setzen und sie dem Browser übergeben.
- *Das Antwort-Dokument erstellen*: Das Servlet erzeugt das Antwort-Dokument, z.B. in HTML. Dabei können die vom Anwendungskern zurückgelieferten Ergebnisse dynamisch in das Antwort-Dokument eingebunden werden.

Beispiel

Nachfolgend ist zum schnellen Einstieg das klassische „Hello World“-Beispiel als Servlet dargestellt. Dieses Servlet nimmt nur die letzen beiden der oben genannten fünf Aufgaben wahr, indem es mit `setContentType(...)` den Dokumententyp im Response Header setzt und mit

[1] Den sinnvollen Zuschnitt von Servlet-Aufgaben diskutiert Kapitel 10.

`println(...)` die Ausgabe erzeugt, nachdem es zuvor mit `getWriter()` ein `PrintWriter`-Objekt mit dem Namen `out` beschafft hat, in das die Ausgabe geschrieben werden kann.

```
1  package de.fernuni.kurs1796.beispiele;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  import javax.servlet.annotation.WebServlet;
7
8  @WebServlet(name="HelloWorld",urlPatterns={"/helloWorld"})
9  public class HelloWorld extends HttpServlet {
10
11   protected void doGet(HttpServletRequest request,
12                        HttpServletResponse response)
13                 throws ServletException, IOException {
14           response.setContentType("text/html");
15           PrintWriter out = response.getWriter();
16           out.println("<html><body>Hello World</body></html>");
17   }
18 }
```

Listing 8.1: „Hello World“-Servlet

Ein Servlet hat einen fest definierten Lebenszyklus, der aus den drei Phasen *Initialisierung*, *Request-Bearbeitung* und *Zerstörung* besteht und über so genannte Callback-Methoden realisiert wird. Callback-Methoden sind in der Regel leere Methoden, die von einer Oberklasse zur Verfügung gestellt und bei Eintritt vorher definierter Ereignisse aufgerufen werden, in diesem Fall vom Web Container. In den abgeleiteten Klassen werden sie bei Bedarf entsprechend implementiert. Die Phasen inklusive ihrer entsprechenden Callback-Methoden sind in Tabelle 8.1 angegeben.

Der Web Container startet für die Dauer der Anwendung genau eine Instanz pro Servlet, sodass alle Anfragen sich diese Instanz teilen müssen. Da pro Request ein *Thread*[2] erzeugt wird und mehrere Threads gleichzeitig auf die einzige Servlet-Instanz zugreifen können, muss ein Servlet *Thread-sicher* (thread safe) sein. Thread-Sicherheit bedeutet, dass eine Software-Komponente mehrfach parallel ausgeführt werden kann, ohne dass sich die einzelnen Ausführungen gegenseitig beeinflussen oder behindern.

Um Thread-Sicherheit zu garantieren, gibt es im Wesentlichen drei Möglichkeiten. Die einfachste Möglichkeit ist, *keine Instanzvariablen*, sondern ausschließlich (Methoden-) lokale Variablen zu benutzen. In Java sind nämlich lokale Variablen stets auch Thread-lokal. Die konsequente Benutzung lokaler Variablen bedeutet, dass alle von den Methoden benötigten Variablen als Parameter übergeben werden müssen. Werden Instanzvariablen genutzt, d.h. greifen mehrere Threads auf die Attribute eines einzigen Objekts zu, könnte ein Thread die Wertebelegungen

[2] Ein Thread (auch leichtgewichtiger Prozess genannt) bezeichnet einen Ausführungsstrang innerhalb eines Programms. Auf einem Rechner mit mehreren Prozessoren ist eine echte parallele Ausführung von Code möglich, indem z.B. jedem Prozessor ein eigener Thread zugewiesen wird.

Phase	**Callback-Methode**	**Erläuterung**
Initialisierung	`init()`	Diese Methode wird beim Erzeugen des Servlet vom Web Container aufgerufen. Dies geschieht beim Starten der Web-Anwendung oder spätestens vor der ersten Request-Bearbeitung. Das Servlet existiert in der Regel solange, bis der Web Container oder die Web-Anwendung gestoppt wird. Beim Initialisieren können z.B. eine Datenbankverbindung, die für die gesamte Laufzeit des Servlet bestehen soll, aufgebaut oder einmalige Berechnungen durchgeführt werden.
Request-Bearbeitung	`service(...)`	Der Web Container ruft bei jedem Request die (öffentliche) `service(...)`-Methode auf, die gemäß der HTTP-Übertragungsmethode an die passende `doXXX(...)`-Methode, etwa `doGet(...)` oder `doPost(...)`, weiterleitet.
Zerstörung	`destroy()`	Diese Methode wird beim Zerstören des Servlets aufgerufen, in der Regel erst beim Stoppen der Web-Anwendung. Notwendige Aufräumarbeiten, wie etwa das Freigeben von Speicherplatz, die Schließung einer Datenbankverbindung oder die Speicherung von Daten, wie z.B. den Wert einer Variable, die als Aufruf-Zähler für Requests fungiert, können hier erledigt werden.

Tabelle 8.1.: Lebenszyklus eines Servlets und zugehörige Callback-Methoden

dieser Attribute ändern, sodass alle anderen Threads nun mit diesen neuen Werten arbeiten müssen.

Die zweite Möglichkeit besteht daraus, alle Instanzvariablen in der `init()`-Methode des Servlets mit Werten zu füllen und bei allen Aufrufen der `service(...)`-Methode nur lesend auf diese Instanzvariablen zuzugreifen, also keine Veränderungen mehr an diesen vorzunehmen (Ausnahme: Die `destroy()`-Methode darf bei der Zerstörung des Servlets die Werte aller Instanzvariablen löschen). Dadurch, dass nur lesend auf die Instanzvariablen zugegriffen wird, können keine Synchronisierungsprobleme auftreten.

Als dritte Möglichkeit kann man Lese- und Schreibvorgänge auf Instanzvariablen erlauben, diese Zugriffe dann allerdings in geeigneter Weise mittels `synchronized()` synchronisieren. Einen virtuellen Warenkorb beim Online-Shopping auf diese Art zu realisieren würde allerdings bedeuten, dass alle Benutzer sich ein und denselben Warenkorb teilen. Im Allgemeinen ist daher diese Art, Thread-Sicherheit zu garantieren, nicht sehr zweckmäßig. Bis auf in Ausnahmesituationen sollten einzelne Anfragen den Zustand eines Servlet nicht ändern dürfen.

8.3. Scopes

Web-Komponenten besitzen die Möglichkeit, mit anderen Web-Komponenten Daten auszutauschen. Die folgenden zwei Szenarien geben Beispiele für den Datenaustausch zwischen Web-Komponenten:

- Bei dem Request-Ablauf in Abbildung 8.1 übernahm ein einziges Servlet die Aufgaben, die Anfrage des Clients anzunehmen, zu bearbeiten und ein entsprechendes HTML-Dokument als Antwort zu erstellen und an den Client zurückzusenden. Servlets können aber auch Anfragen an andere Servlets weiterleiten, um eine Aufgabenteilung vorzunehmen. So kann ein Servlet die Authentifizierung und Autorisierung der Benutzerin[3] sowie die Anwendungskernaufrufe übernehmen und anschließend an ein anderes Servlet (bzw. eine JavaServer Page) weiterleiten, das nur für die Erstellung des HTML-Dokuments zuständig ist. Die beiden Servlets müssen dazu Daten austauschen – in diesem Beispiel gibt das erste Servlet dem zweiten Servlet Informationen darüber, welche Daten benötigt werden.
- Beim Online-Shopping könnte ein Servlet dafür zuständig sein, alle Anfragen zu bearbeiten, bei denen Waren in den virtuellen Warenkorb gelegt werden, während ein anderes Servlet alle Anfragen bearbeitet, die die Bezahlung und den Versand dieser Waren betreffen. Die beiden Servlets tauschen dazu Daten aus – in diesem Beipiel übergibt das erste Servlet die Daten des virtuellen Warenkorbs an das zweite Servlet.

[3] Unter Authentifizierung versteht man den Nachweis, dass eine Nutzerin genau die Person ist, für die sie sich ausgibt. Die Autorisierung umfasst z.B. die Zuweisung von Zugriffsrechten und beantwortet die Frage worauf die authentifizierte Person zugreifen darf.

Damit Daten für die Laufzeit eines Java EE-Anwendungsprogramms verfügbar sind, müssen diese Daten temporär gespeichert werden. Die Länge eines solchen Zeitraums definiert der sogenannte *Scope* (engl. Reichweite, Gültigkeitsbereich). Ein Scope fungiert als Behälter für Daten. Die Spezifikation der Servlet Technologie kennt mehrere dieser Scopes, die unterschiedliche Lebensdauern für Objekte definieren, die sich innerhalb des jeweiligen Scopes befinden. Ein Scope ist in Java EE ein Objekt, das über Attribute verfügt, in denen die zu speichernden Daten hinterlegt werden.

Um auf diese Attribute zuzugreifen werden u.a. die folgenden vier Methoden durch ein Scope-Objekt zur Verfügung gestellt. Durch die Methode `setAttribute(...)` wird ein Objekt `object` unter dem Namen name im Scope-Objekt gespeichert. Die konkreten Datentypen, die in Attributen abgelegt werden können, werden im nächsten Abschnitt genannt. Da ein Scope-Objekt keine feste Anzahl von Attributen besitzt, in denen Daten gespeichert werden können, sondern diese in einer Liste gespeichert werden, muss jedes zu speichernde Objekt durch einen Namen eindeutig identifizierbar gemacht werden. Die Methode `getAttribute(...)` liest den Wert des Attributes mit dem übergebenen Namen aus und gibt das unter diesem Namen gespeicherte Objekt zurück. Die Methode `getAttributeNames(...)` gibt eine Liste von Namen aller in diesem Request enthaltenen Attribute zurück, und die Methode `removeAttribute(...)` entfernt ein Attribut mit dem übergebenen Namen aus der Liste aller verfügbaren Attribute.

`public void setAttribute(String name, Object object),`
`public Object getAttribute(String name),`
`public Enumeration getAttributeNames()` und
`public void removeAttribute(String name).`

Im Folgenden betrachten wir die wichtigsten Scopes, die in Java EE zur Verfügung stehen.

Als *Request Scope* wird das durch die Schnittstelle `HttpServletRequest` definierte Request-Objekt bezeichnet, das die Daten für die Dauer eines Request-Durchlaufs aufbewahrt. So kann z.B. ein Servlet den Anwendungskern aufrufen, die Ergebnisse im Request-Objekt ablegen und an ein anderes Servlet weiterleiten. Letzteres liest die Ergebnisse aus dem Request-Objekt wieder aus und erstellt mit diesen Informationen das HTML-Dokument. Das Request-Objekt wird vom Web Container bei jedem Request erzeugt, dem Servlet zur Verfügung gestellt und nach Ablauf des Requests gelöscht.

Als *Session Scope* wird das durch die Schnittstelle `HttpSession` definierte Session-Objekt bezeichnet, das die Daten für die Dauer einer Session aufbewahrt. Die dort abgelegten Informationen sind von allen innerhalb dieser Session ausgeführten Anfragen aus erreichbar. So kann z.B. das Servlet einer Anfrage Waren in den virtuellen Warenkorb legen und den Warenkorb im Session-Objekt aufbewahren. Das Servlet einer anderen Anfrage könnte schließlich diesen Warenkorb auslesen, die Rechnung erstellen und den Versand der darin enthaltenen Waren veranlassen. Nach Ablauf der Session werden die Daten automatisch gelöscht. Das Session-Objekt wird vom Request-Objekt mittels der Methode `getSession()` angesprochen.

Als *Application Scope* wird das durch das Interface `ServletContext` definierte Context-Objekt bezeichnet, das die für die Webanwendung globalen Daten für die gesamte Dauer der

Webanwendung aufbewahrt. Das Objekt steht allen Komponenten der Web-Schicht zur Verfügung und wird erst beim Beenden der Web-Anwendung automatisch gelöscht. Das Context-Objekt wird von einem Servlet mittels der Methode `getServletContext()` angesprochen.

Der vierte Scope, der als Page Scope bezeichnet wird, besitzt nur JSP-lokale Daten und ist somit nur für JSP-Seiten relevant. Er wird in Abschnitt 9.1 wieder aufgegriffen.

8.3.1. Java Beans

Daten, die in einem Scope gespeichert werden können, sind z.B. Zeichenketten (Strings) oder Standarddatentypen als Objekte von Wrapper-Klassen[4]. Standardmäßig werden jedoch JavaBeans-Objekte zur Speicherung verwendet [JB/Spec]. Eine JavaBean ist eine Java-Klasse, die bestimmte Anforderungen erfüllt:

- sie besitzt einen öffentlichen Konstruktor
- sie besitzt keine öffentlichen Attribute
- sie besitzt für jedes Attribut, das von außen benutzbar sein soll, eine öffentliche Methode, die den Wert dieses Attributs setzt (`set`-Methode) und eine öffentliche Methode, die den Wert dieses Attributs zurückgibt (`get`-Methode).

Ein Attribut, für das mindestens eine der beiden Methoden vorhanden ist, wird auch Property (Eigenschaft) genannt. Eine Property, zu der keine `set`-Methode existiert, ist eine Read-Only-Property, entsprechend wird eine Property ohne `get`-Methode als Write-Only-Property bezeichnet.

Die `get`- und `set`-Methoden werden als *Getter* bzw. *Setter* und – zusammenfassend – als Accessor Methods bezeichnet. Die JavaBeans-Spezifikation [JB/Spec] schreibt vor, dass dem Präfix `get` bzw. `set` der Bezeichner der Property mit großem Anfangsbuchstaben folgt. Getter dürfen keinen Parameter und Setter genau einen Parameter vom Typ der Property besitzen. Der Rückgabetyp des Getters ist der Typ des entsprechenden Attributs.

Beispielsweise würden für eine Property `name` vom Typ einer Zeichenkette der Getter `public String getName()` und der Setter entsprechend `public void setName(String value)` lauten.

Bei booleschen Attributen kann statt `get` auch das Präfix `is` verwendet werden.

[4] Eine Wrapper-Klasse ist eine Klasse, die einen primitiven Datentyp wie z.B. `int`, `boolean`oder `double`„umhüllt“. Entsprechende Variablen können dann als Objekte gehandhabt werden.

8.4. Servlet API

Bei der Entwicklung von Java EE-Anwendungsprogrammen wird das Java EE Software Development Kit (siehe Abschnitt 7.1) genutzt. Dieses stellt den Entwicklern Schnittstellen auf Quellcodeebene zur Verfügung. Eine derartige Schnittstelle wird *Programmierschnittstelle* (engl. application programming interface - API) genannt. Jede Programmierschnittstelle stellt einen Programmteil des Application Servers dar, der es der Entwicklerin ermöglicht, ihr eigenes Programm an den Application Server anzubinden. Diese Anbindung besteht aus der Nutzung von Klassen und Methoden oder der Implementierung von Schnittstellen. Um auf die Funktionalitäten eines Servlets zugreifen zu können, wird die *Servlet-API* verwendet. Alle Klassen dieser API sind in dem Java-Paket `javax.servlet` und seinen Unterpaketen enthalten[5]. Diese Pakete sind bereits im Java EE SDK vorhanden.

Die folgende Abbildung zeigt die wichtigsten Klassen- und Schnittstellendefinitionen der Servlet-API. Die nicht eingefärbten Klassen- und Schnittstellendefinitionen sind innerhalb des Web Containers im Application Server bereits implementiert. Die Klasse `HttpServlet` und ihre Oberklasse `GenericServlet` sind abstrakt. `HttpServlet` ist eingefärbt, da jedes Servlet von der Entwicklerin als Unterklasse von `HttpServlet` zu implementieren ist. Eine gestrichelte Linie mit offener Pfeilspitze bezeichnet eine *Benutzungsabhängigkeit* (UML: use dependency), eine gestrichelte Linie mit geschlossener Pfeilspitze eine *Realisierungsabhängigkeit* (UML: realize dependency, Java: `implements`).

Im Folgenden stellen wir die wichtigsten in diesem Kapitel bisher erwähnten Klassen, Schnittstellen und Methoden vor. Die von einer Oberklasse geerbten Methoden sind am Ende der jeweiligen Zeile mit einem Stern (*) gekennzeichnet.

Die abstrakte Klasse HttpServlet

Die abstrakte Klasse `HttpServlet` gibt das Grundgerüst einer Servlet-Klasse vor, d.h. eigene Servlets werden von `HttpServlet` abgeleitet. Der Web Container ruft bei einem Request die öffentliche Methode

```
public void service(ServletRequest req, ServletResponse res) *
```

auf. Diese Methode dient dem Servlet nur als eine Art Dispatcher (siehe Abschnitt 8.1), der die Anfragen gemäß ihrer HTTP-Übertragungsmethode auf die entsprechende `doXXX()`-Methode verteilt. Die Entwickler müssen sich nur um die Implementierung dieser `doXXX()`-Methoden kümmern. Nachfolgend sind beispielhaft zwei dieser Methoden aufgelistet.

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
protected void doPost(HttpServletRequest req, HttpServletResponse resp)
```

Neben diesen Methoden können bei Bedarf auch die beiden Methoden

[5] Ein Java-Paket ist eine Menge von Dateien, die Klassendefinitionen, Schnittstellendefinitionen, Java-Enumerationen und/oder definierte Java-Exceptions enthalten.

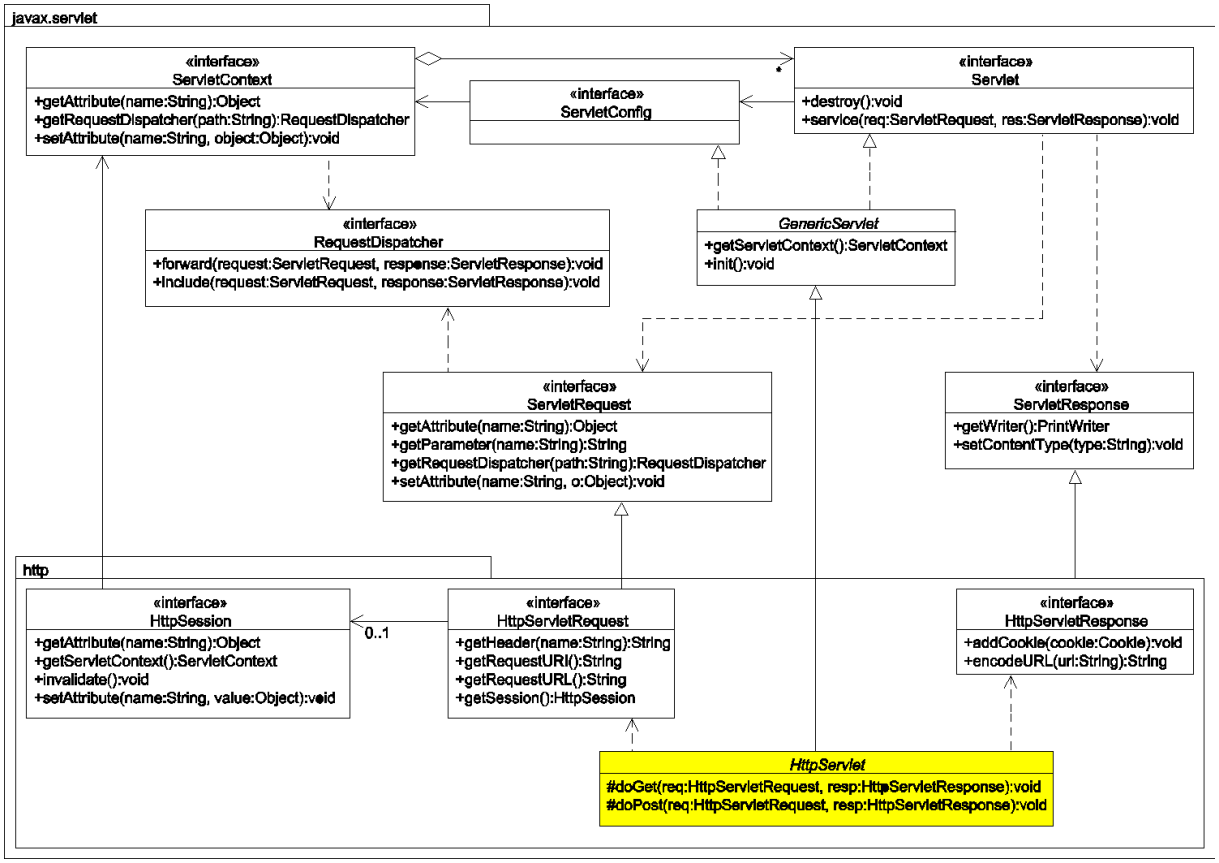

Abbildung 8.2.: Die wichtigsten Klassen und Schnittstellen der Servlet API

```
public void init() *
public void destroy() *
```

implementiert werden. `init()` wird bei der Erzeugung der Servlet-Instanz aufgerufen, `destroy()` bei seiner Zerstörung.

Die Schnittstelle HttpServletRequest

Bei einem Request erzeugt der Web Container ein Anfrageobjekt vom Typ `HttpServletRequest` (Request Scope) und füllt es mit den Daten des HTTP-Request. Neben den in Abschnitt 8.3 bereits erwähnten vier Methoden zum Bearbeiten von Attributdaten in den Scopes bietet das Request-Objekt noch weitere wichtige Methoden an.

Mit den Methoden

```
public Enumeration getParameterNames() *
public String getParameter(String name) *
public String[] getParameterValues(String name) *
public Map getParameterMap() *
```

können die im Request übermittelten Parameterangaben ausgelesen werden, wie z.B. der Parametersuche mit dem Wert `computer` in der URL `www.kaufhaus.de?suche=computer`. Ein großer Unterschied zwischen Attributen und Parametern einer Anfrage besteht darin, dass in Attributen auch Java-Objekte gespeichert werden können, in Parametern lediglich Zeichenketten (Strings).

Mit den Methoden

```
public Enumeration getHeaderNames()
public String getHeader(String name)
public Enumeration getHeaders(String name)
```

können die im Request übermittelten Header ausgelesen werden. Die Methode `getHeaderNames()` gibt eine Auflistung der Namen aller Felder des Headers zurück. Die Methode `getHeader(...)` erwartet einen Feldnamen des Headers und gibt den dazugehörigen Wert zurück. Z.B. kann für das Feld Accept-Charset, das den vom Client erwarteten Zeichensatz im Antwortdokument spezifiziert, der Wert `utf-8` ausgelesen werden. Die Methode `getHeaders(...)` kann mehrere Werte für ein einzelnes Feld auslesen, z.B. könnte der Client angeben, dass er mehrere Zeichensätze akzeptiert.

Die beiden Methoden

`public String getRequestURI()` und
`public StringBuffer getRequestURL()`

beziehen sich auf die Request-URI. Leider sind die Namen ein wenig unglücklich gewählt, tatsächlich gibt `getRequestURI()` nur den relativen Pfad zurück. Dagegen liefert `getRequestURL()` die ganze URI ohne Query-String zurück. Bei der HTTP-Anfrage

```
GET http://www.FernUni-Hagen.de/index.html HTTP/1.1
```

wird durch `getRequestURI()` „index.html“ zurückgegeben, während `getRequestURL()` „http://www.FernUni-Hagen.de/index.html“ liefert.

Die Methode

```
public HttpSession getSession()
```

gibt das Session-Objekt (Session Scope) zurück. Es wird ein neues Session-Objekt erzeugt, falls noch keines existiert.

Die Methode

```
public RequestDispatcher getRequestDispatcher(String path) *
```

gibt ein Objekt vom Typ `RequestDispatcher` (s.u.) für ein bestimmtes Servlet, eine JSP-Seite oder ein statisches HTML-Dokument zurück, die Angabe geschieht über den Parameter `path`. Mit diesem Objekt kann z.B. die erhaltene Anfrage an andere Servlets oder JavaServer Pages weitergeleitet werden.

Die Schnittstelle HttpServletResponse

Bei einer Anfrage erzeugt der Web Container neben dem Anfrageobjekt auch ein Antwortobjekt vom Typ `HttpServletResponse`, das für die Antwort von einem Servlet gefüllt wird

und nach der Erstellung noch keine Daten enthält. Dazu enthält diese Schnittstelle u.a. Definitionen von Methoden zum Manipulieren der Response-Header und zum Spezifizieren des Antwortdatenbereichs.

Die Methode

```
public void setContentType(String type) *
```

setzt den Wert (oder die Werte) für das Header-Feld `Content-Type` in der Response zur Definition des Typs des Datenbereichs, etwa „`text/html; charset=utf-8`“. Der Browser des Clients kann mit Hilfe dieses Feldes feststellen, um was für ein Dokument es sich handelt und welcher Zeichensatz benutzt wurde, um das Dokument zu erstellen.

Die Methode

```
public PrintWriter getWriter() *
```

gibt ein Objekt der Klasse PrintWriter zurück. Mit der Methode `println(...)` dieses Objektes kann der Datenbereich der Antwort gefüllt werden. Als Parameter erwartet diese Methode eine Zeichenkette oder einen Ausdruck, der zu einer Zeichenkette ausgewertet werden kann. Nach der Ausgabe der Zeichenkette wird automatisch ein Zeilenumbruch durchgeführt. Soll kein Zeilenumbruch erfolgen, so ist die Methode `print(...)` zu verwenden.

Die Methode

```
public String encodeURL(String url)
```

führt das URL-Rewriting durch, d.h. sie setzt die Session-ID in die übergebene URL ein. Falls Cookies eingesetzt werden – dies erkennt der Web Container –, lässt die Methode die URL unverändert. Unabhängig davon, ob Cookies eingesetzt werden oder nicht, sollte aber diese Methode für *jede* URL aufgerufen werden, die in einer Webseite vorkommt und bei deren Aufruf die Session nicht verloren gehen soll. Somit ist die Web-Anwendung auch auf den Fall vorbereitet, dass der Browser keine Cookies akzeptiert.

Die Methode

```
public void addCookie(Cookie cookie)
```

setzt einen Cookie, der bei einem späteren Request abgefragt werden kann. Als Parameter wird ein Objekt vom Typ `Cookie` übergeben, das vor Aufruf der Methode vorhanden sein oder erzeugt werden muss.

Die Schnittstelle HttpSession

Diese Schnittstelle ermöglicht den Zugriff auf das Session-Objekt (dieses Objekt wird als Session-Scope bezeichnet und beinhaltet die Daten einer Session). Neben den in Abschnitt 8.3 bereits erwähnten vier Methoden zum Bearbeiten von Daten in den Scopes bietet das Session-Objekt noch zwei weitere wichtige Methoden an:

Die Methode

```
public void invalidate()
```

kann eine Session für ungültig erklären. Alle in dem Session-Objekt abgelegten Daten gehen verloren. Nachfolgende Requests mit der Session-ID dieser Session erhalten eine Fehlermeldung, wenn versucht wird auf den Session-Scope zuzugreifen.

Die Methode

```
public ServletContext getServletContext()
```

gibt das Objekt des Typs `ServletContext` zurück. Für jede Web-Anwendung gibt es nur ein einziges `ServletContext`-Objekt.

Die Schnittstelle ServletContext

Diese Schnittstelle bietet Zugriff auf das Context-Objekt (Application Scope). Analog zur Schnittstelle `HttpServletRequest` wird unter anderem als wichtige Methode

```
public RequestDispatcher getRequestDispatcher(String path)
```

bereitgestellt. Sie liefert ein Objekt vom Typ `RequestDispatcher` für ein bestimmtes Servlet, eine JSP-Seite oder ein statisches HTML-Dokument zurück. Mit `RequestDispatcher` wird die Anfrage an ein anderes Servlet oder eine JSP-Seite weitergeleitet. Im Gegensatz zu der gleichnamigen Methode in `HttpServletRequest` akzeptiert diese Methode keinen relativen Pfad, an den weitergeleitet werden soll.

Die Schnittstelle RequestDispatcher

Mit dieser Schnittstelle kann ein Servlet die Anfrage an ein anderes Servlet, eine JSP-Seite oder statische Seite weiterleiten, um z.B. die Erstellung des HTML-Dokumentes von einem anderen Servlet übernehmen zu lassen. Dazu werden zwei Methoden angeboten. Die Ziel-Ressource wird schon beim Aufruf mittels `getRequestDispatcher(String path)` festgelegt (s.o.).

Die Methode

```
public void forward(ServletRequest request, ServletResponse response)
throws ServletException, IOException
```

reicht den Request an eine andere Ressource (Servlet, JSP-Seite etc.) weiter. Vorbedingung ist, dass noch keine Ausgabe in den Datenbereich der Antwort mit Hilfe eines Objektes vom Typ `PrintWriter` geschrieben wurde. Nachdem die Ressource abgearbeitet wurde, d.h. nachdem z.B. ein über den `RequestDispatcher` angesprochenes Servlet die Anfrage bearbeitet hat, veranlasst der `RequestDispatcher` das Absenden der Antwort.

Die Methode

```
public void include(ServletRequest request, ServletResponse response)
```

reicht den Request nur vorübergehend an eine andere Ressource (Servlet, JSP-Seite etc.) wei-

ter. Nach Beendigung der inkludierten Ressource geht die Kontrolle wieder auf das aufrufende Servlet oder die aufrufende JSP-Seite über. Diese Methode ermöglicht es, die Ausgabe von mehreren Servlets und JSP-Seiten gemeinsam erzeugen zu lassen. So können z.B. ein Servlet für die Annahme der Anfrage des Clients und drei weitere Servlets, die mit Hilfe des `RequestDispatcher` und der Methode `include(...)` eingebunden werden, für die Erstellung des Ausgabedokumentes zuständig sein, ein Servlet für den Kopfbereich der auszugebenden HTML-Seite, ein Servlet für den Navigationsbereich und ein Servlet für den Hauptbereich der Seite.

8.5. Beispiel: Systemanmeldung mit Servlets

Zum Abschluss des Kapitels über Servlets soll ein sehr einfaches Web-Anwendungsprogramm mit dem Namen „SystemanmeldungServlets“ entwickelt werden. Da das Beispiel die Implementierung von Servlets erläutern soll, wird auf JSP-Seiten verzichtet. Die aus Sicht des Software Engineering bessere Lösung mit JSP-Seiten wird in Abschnitt 9.7 vorgestellt.

Das Web-Anwendungsprogramm „SystemanmeldungServlets“ nimmt den Namen und das Passwort entgegen, überprüft beide und zeigt bei korrekter Eingabe eine Erfolgsseite, bei falscher Eingabe wieder das Anmeldeformular inklusive einer Fehlermeldung an.

Für die Web-Anwendung werden zwei Servlets eingesetzt. Das erste Servlet – `ShowSignInServlet` – hat die Aufgabe das Anmeldeformular ohne Fehlermeldung (s. Abbildung 8.3) anzuzeigen, falls die eingegebenen Daten richtig sind und das Anmeldeformular mit einer Fehlermeldung auszugeben (s. Abbildung 8.4), wenn falsche Daten in das Formular eingegeben wurden. Beim Absenden des Formulars an den Server sollen die Daten an das zweite Servlet – `CheckSignInServlet` – gesendet werden. Das Servlet überprüft die Eingabe und zeigt im Erfolgsfall die in Abbildung 8.5 dargestellte Web-Seite an. In unserem Beispiel werden die Eingabedaten als richtig erkannt, wenn für den Namen „kurs1796“ und für das Passwort „geheim“ eingegeben werden.

Das zur Abbildung 8.3 gehörende HTML-Dokument, das an den Client gesendet wird, ist durch Codeausschnitt 8.2 gegeben.

```
1  <html>
2    <head>
3      <title>Beispiel Systemanmeldung</title>
4    </head>
5    <body>
6      <img src="images/LogoFeu.jpg" alt="FernUniversit&auml;t in Hagen">
7      <h1>Beispiel &quot;Systemanmeldung&quot;</h1>
8      <h3>aus dem Kurs 01796, Kurseinheit 3</h3>
9      <form method="POST" action="/SystemanmeldungServlets/CheckSignInServlet
           ">
10       <table cellpadding="10" style="border-collapse: collapse; border-
           width: 0">
11         <tr>
```

```
12            <td>Benutzername:</td>
13            <td><input type="text" name="username" size="20"></td>
14          </tr>
15          <tr>
16            <td>Passwort:</td>
17            <td><input type="password" name="password" size="20"></td>
18          </tr>
19        </table>
20        <p><input type="submit" value="Anmelden" name="buttonLogin"></p>
21      </form>
22    </body>
23  </html>
```

Listing 8.2: HTML-Quelltext des Anmeldeformulars

Innerhalb der HTML-Seite ist ein Formular mit zwei Attributen definiert. Zum einen sollen die Fomulardaten per POST-Methode an den Server übertragen werden (`method="POST"`), zum anderen wird im `action`-Attribut der Pfad des Servlets angegeben, das die Verarbeitung der Formulardaten übernimmt, wenn das Formular bei Betätigung der `submit`-Schaltfläche an den Server gesendet wird. Im Fomular befinden sich zwei Texteingabefelder, eines für die Eingabe des Namens der Benutzerin und eines für die versteckte Eingabe eines Passwortes.

Bei einer falschen Eingabe des Namens und/oder des Passwortes soll das Anmeldeformular erneut angezeigt werden. In diesem Fall soll allerdings eine Fehlermeldung angezeigt werden, wie in Abb. 8.4. dargestellt.

Das Servlet `ShowSignInServlet`, das die Anmeldeseite erzeugt, ist in Codeausschnitt 8.3 dargestellt. Um das Servlet `ShowSignInServlet` zu implementieren, wird eine Klasse mit dem Namen `ShowSignInServlet` definiert, die von der abstrakten Klasse `HttpServlet` erbt und die zwei Methoden `doGet(...)` und `doPost(...)` implementiert. Direkt oberhalb der Klasse wird die Annotation `@WebServlet` benutzt, um die Servlet-Klasse mit dem Attribut `name` eindeutig im Web Container identifizierbar zu machen und mit dem Attribut `urlPatterns` den dort angegebenen Pfad auf die Servlet-Klasse abzubilden. Die importierten Java-Pakete sind notwendig, um die abstrakte Klasse `HttpServlet`, die WebServlet-Annotation und die Klasse `PrintWriter` nutzen zu können.

In diesem Beispiel werden alle GET- und POST-Anfragen des Clients gleich behandelt. Aus diesem Grund besteht die `doPost`-Methode lediglich aus einer Weiterleitung ihrer Parameter an die Methode `doGet(...)`. Innerhalb der `doGet`-Methode wird ein Objekt out vom Typ `PrintWriter` mit Hilfe des `response`-Objektes erzeugt. Dieses wird dazu genutzt, den Inhalt der HTML-Seite zu erzeugen, die an den Client zurückgesendet wird.

Um bei falscher Eingabe eine Fehlermeldung anzuzeigen, ist eine Abfrage vorgesehen, die überprüft, ob im Request Scope eine Fehlerinformation abgelegt wurde. Ist dies der Fall, wird zusätzlicher HTML-Code ausgegeben, der den Fehler in roter Schrift darstellt. Im Wesentlichen erzeugt `ShowSignInServlet` mittels `println(...)` Zeile für Zeile das HTML-Dokument.

```
1  package kurs01796.ke3.servletbeispiel;
2
```

```
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.annotation.WebServlet;
6  import javax.servlet.http.*;
7
8  @WebServlet(name="ShowSignInServlet", urlPatterns={"/ShowSignInServlet"})
9  public class ShowSignInServlet extends HttpServlet {
10       @Override
11     protected void doGet(HttpServletRequest request, HttpServletResponse
           response) throws ServletException, IOException {
12         response.setContentType("text/html");
13         PrintWriter out = response.getWriter();
14         //HTML-Dokument erstellen
15         out.println("<html>");
16         out.println("<head>");
17         out.println("<title>Beispiel Systemanmeldung</title>");
18         out.println("</head>");
19         out.println("<body>");
20         out.println("<img src=\"images/LogoFeu.jpg\" alt=\"FernUniversit&
               auml;t in Hagen\">");
21         out.println("<h1>Beispiel &quot;Systemanmeldung&quot;</h1>");
22         out.println("<h3>aus dem Kurs 01796, Kurseinheit 3</h3>");
23         out.println("<form method=\"POST\" action=\"/
               SystemanmeldungServlets/CheckSignInServlet\">");
24         out.println("<table cellpadding=\"10\" style=\"border-collapse:
               collapse; border-width: 0\">");
25         out.println("<tr><td>Benutzername:</td>");
26         out.println("<td>");
27         out.println("<input type=\"text\" name=\"username\" size=\"20\"></
               td></tr>");
28         out.println("<tr><td>Passwort:</td>");
29         out.println("<td>");
30         out.println("<input type=\"password\" name=\"password\" size
               =\"20\"></td></tr>");
31         out.println("</table>");
32         //Wenn ein Fehler aufgetreten ist, dann soll eine
33         //entsprechende Meldung angezeigt werden
34         if (request.getAttribute("errorOccurred") != null) {
35             out.println("<p><span style=\"color: rgb(255, 0, 0);\">Sie
                   konnten nicht authentifiziert werden.</span></p>");
36         }
37         out.println("<p><input type=\"submit\" value=\"Anmelden\" name=\"
               buttonLogin\"></p>");
38         out.println("</form>");
39         out.println("</body>");
40         out.println("</html>");
41     } //Ende doGet(...)
42
43       @Override
44     protected void doPost(HttpServletRequest request, HttpServletResponse
           response) throws ServletException, IOException {
45         doGet(request, response);
46     }
```

```
47 }
```

Listing 8.3: ShowSignInServlet

Das zweite Servlet – `CheckSignInServlet` – soll die Korrektheit der Anmeldedaten überprüfen und bei korrekter Anmeldung die in Abbildung 8.5 dargestellte Erfolgsseite anzeigen:

Das zugehörige HTML-Dokument, das an den Client gesendet wird, ist in Codeausschnitt 8.4 angegeben.

```
1  <html>
2    <head>
3      <title>
4        Beispiel Systemanmeldung - erfolgreich authentifiziert
5      </title>
6    </head>
7    <body>
8      <img src="images/LogoFeu.jpg" alt="FernUniversit&auml;t in Hagen">
9      <h1>Beispiel &quot;Systemanmeldung&quot;</h1>
10     <h3>aus dem Kurs 01796, Kurseinheit 3</h3>
11     <p>Sie wurden erfolgreich als Benutzer &quot;kurs1796&quot;
          authentifiziert.</p>
12   </body>
13 </html>
```

Listing 8.4: HTML-Quelltext für den geschützten Bereich

Das Servlet `CheckSignInServlet`, das zur Verarbeitung der Anmeldedaten im `action`-Attribut des Formlars angegeben wurde, ist in Codeausschnitt 8.5 zu sehen. Sind die Anmeldedaten fehlerhaft, so soll das Anmeldeformular mit Fehlermeldung angezeigt werden.

Das Servlet `CheckSignInServlet` wird wie das Servlet `ShowSignInServlet` als Klasse definiert, die von der abstrakten Klasse `HttpServlet` erbt. Auch sind die Methoden `doGet(...)` und `doPost(...)` implementiert, und die Methode `doPost(...)` leitet ihre übergebenen Parameter an die `doGet`-Methode weiter. Innerhalb der Methode `doGet(...)` werden die eingegebenen Formulardaten mit Hilfe der Request-Parameter ausgelesen, hierzu wird das Objekt `request` genutzt. Nach der Ablage der Parameter in den Variablen `username` und `password` kann die Validierung dieser Daten vorgenommen werden. Eine Fehlernachricht wird dann ausgegeben, wenn kein Name angegeben wurde, wenn der Name mit der Zeichenkette „kurs1796“ nicht übereinstimmt (Groß-Klein-Schreibung wird ignoriert), wenn kein Passwort angegeben wurde oder wenn das Passwort nicht mit der Zeichenkette „geheim“ übereinstimmt. Stimmen die Eingaben mit dem richtigen Namen und dem richtigen Passwort überein, wird der HTML-Code für den geschützten Bereich mit Hilfe eines `PrintWriter`-Objektes ausgegeben. Ist dies nicht der Fall, wird im Request-Scope ein neues Attribut mit dem Namen `errorOccurred` und dem Wert `true` abgelegt. Danach wird ein Objekt vom Typ `RequestDispatcher` mit Hilfe der Methode `getRequestDispatcher(...)` des Objektes `request` erzeugt. Die Methode `getRequestDispatcher` gibt dieses Objekt als Rückgabewert zurück. Der Parameterwert, der an diese Methode übergeben wird, stellt den relativen Pfad zum Servlet `ShowSignInServlet` dar. Mit einem Aufruf der Methode `forward(...)` des

RequestDispatcher-Objektes wird der Request an das Servlet ShowSignInServlet weitergeleitet, dabei ist selbstverständlich das neu hinzugefügte errorOccurred-Attribut im Request-Scope vorhanden.

```
1  package kurs01796.ke3.servletbeispiel;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.annotation.WebServlet;
5  import javax.servlet.http.*;
6
7  @WebServlet(name="CheckSignInServlet", urlPatterns={"/CheckSignInServlet"})
8  public class CheckSignInServlet extends HttpServlet {
9        @Override
10     protected void doGet(HttpServletRequest request, HttpServletResponse
           response) throws ServletException, IOException {
11         //die Benutzereingaben aus dem Request holen
12         String username = (String) request.getParameter("username");
13         String password = (String) request.getParameter("password");
14         //pruefen, ob der korrekte Benutzername und das korrekte
15         //Passwort angegeben wurden
16         if ((username == null) || !username.equalsIgnoreCase("kurs1796") ||
               (password == null) || !password.equals("geheim")) {
17             //nicht erfolgreiche Authentifizierung:
18             //Information, dass ein Fehler aufgetreten ist,
19             //im Request Scope ablegen:
20             request.setAttribute("errorOccurred", "true");
21             //weiterleiten an ShowSignInServlet zwecks
22             //Ausgabe der Anmeldeseite:
23             RequestDispatcher rd = request.getRequestDispatcher("/
                  ShowSignInServlet");
24             rd.forward(request, response);
25         } else {
26             response.setContentType("text/html");
27                   PrintWriter out = response.getWriter();
28             //HTML-Dokument erstellen
29             out.println("<html>");
30             out.println("<head>");
31             out.println("<title>Beispiel Systemanmeldung - erfolgreich
                  authentifiziert</title>");
32             out.println("</head>");
33             out.println("<body>");
34             out.println("<img src=\"images/LogoFeu.jpg\" alt=\"
                  FernUniversit&auml;t in Hagen\">");
35             out.println("<h1>Beispiel &quot;Systemanmeldung&quot;</h1>");
36             out.println("<h3>aus dem Kurs 01796, Kurseinheit 3</h3>");
37             out.println("<p>Sie wurden erfolgreich als Benutzer &quot;" +
                  username + "&quot; authentifiziert.</p>");
38             out.println("</body>");
39             out.println("</html>");
40             out.close();
41         } } //Ende doGet(...)
42
```

```
43       @Override
44     protected void doPost(HttpServletRequest request, HttpServletResponse
           response) throws ServletException, IOException {
45         doGet(request, response);
46 } }
```

Listing 8.5: CheckSignInServlet

Codeausschnitt 8.6 zeigt den Deployment Descriptor web.xml. Für dieses Beispiel beinhaltet er nur die Welcome File List, da beide definierten Klassen bereits über Annotationen als Servlets definiert wurden. In die Welcome File List wird das erste Servlet – `ShowSignInServlet` – eingetragen, damit dieses automatisch zum Einsatz kommt, wenn vom Client keine spezielle Ressource, wie z.B. eine Web-Seite, verlangt wird. Innerhalb des öffnenden und schließenden `welcome-file`-Tags ist lediglich der durch die Annotation definierte Name des Servlets anzugeben.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <web-app xmlns="http://java.sun.com/xml/ns/javaee"
4           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5           xsi:schemaLocation= "http://java.sun.com/xml/ns/javaee
6           http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
7
8    <welcome-file-list>
9      <welcome-file>
10         ShowSignInServlet
11     </welcome-file>
12   </welcome-file-list>
13 </web-app>
```

Listing 8.6: Der Deployment Descriptor web.xml

Abschließend noch die zugehörige Verzeichnisstruktur inklusive der Dateien:

```
SystemanmeldungServlets/images/LogoFeu.jpg
SystemanmeldungServlets/WEB-INF/web.xml
SystemanmeldungServlets/WEB-INF/classes/kurs01796/ke3/
servletbeispiel/ShowSignInServlet.class
SystemanmeldungServlets/WEB-INF/classes/kurs01796/ke3/
servletbeispiel/CheckSignInServlet.class
```

Kopiert man diese Verzeichnisstruktur inklusiver ihrer Dateien in das vom Web Container dafür vorgesehene Verzeichnis, so kann die Web-Anwendung über einen Browser angesprochen werden. Für den Fall, dass die Web-Anwendung auf dem Host `localhost:8080` läuft, ist die zugehörige URL

```
http://localhost:8080/SystemanmeldungServlets
```

anzugeben.

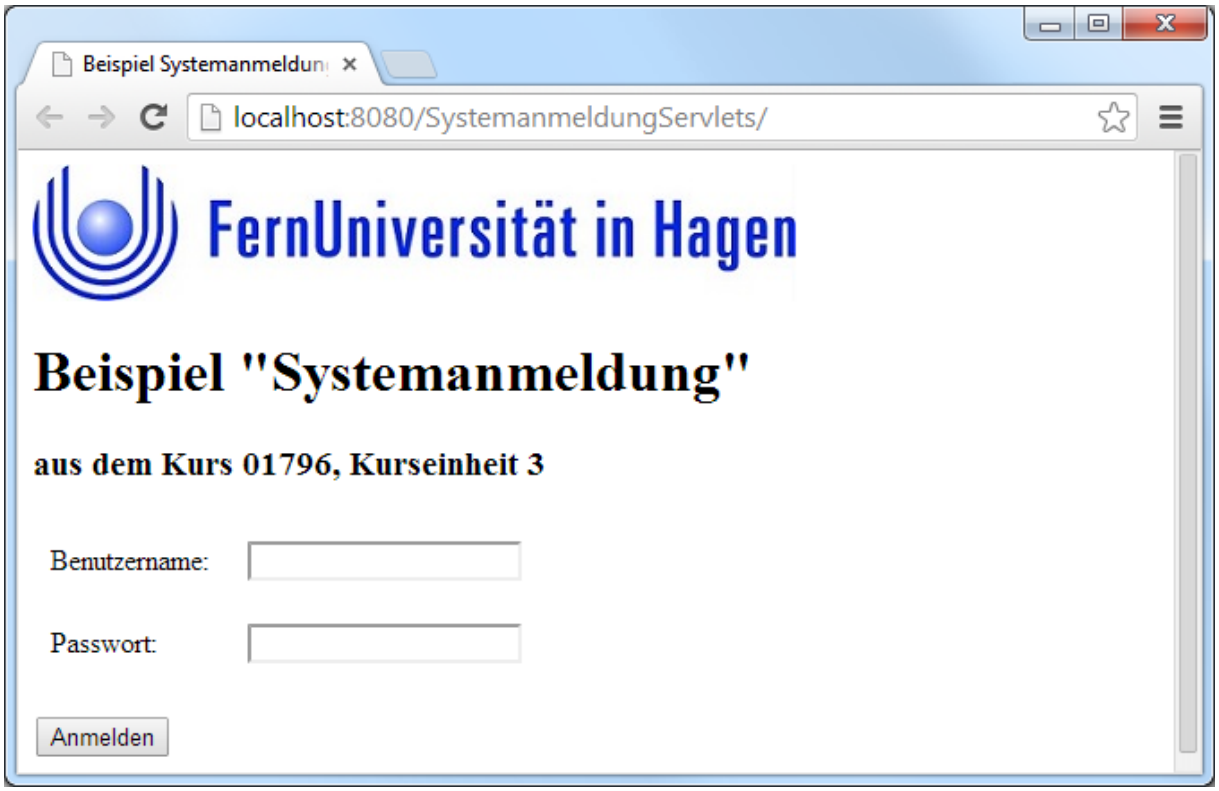

Abbildung 8.3.: Screenshot des Anmeldeformulars

Abbildung 8.4.: Screenshot des Anmeldeformulars bei nicht korrekter Anmeldung

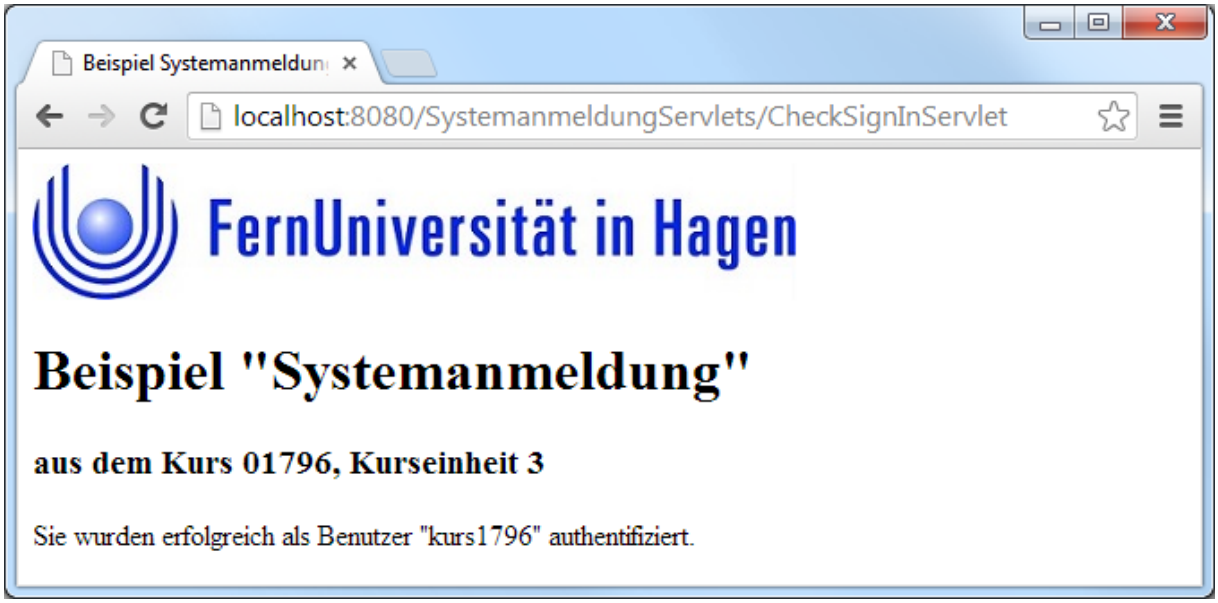

Abbildung 8.5.: Screenshot des „Mitgliederbereichs“

Diese Seite bleibt aus technischen Gründen frei!

9. JavaServer Pages

Ein Ausgabedokument kann durch ein Servlet direkt erzeugt werden, unter Verwendung der Methode `println(...)` einer `PrintWriter`-Instanz. Dies ist aber nicht nur eine mühsame Angelegenheit, auch werden die Auszeichnungssprache HTML und die Programmiersprache Java vermischt, mit negativen Folgen für die Testbarkeit und Wartbarkeit. Zudem können die so codierten Web-Seiten erst zum Ausführungszeitpunkt bezüglich ihrer Gestaltung endgültig beurteilt werden. Diese Nachteile werden vermieden, wenn das Ausgabedokument mit Hilfe einer *JavaServer Pages-Seite* (JSP-Seite) erzeugt wird.

JavaServer Pages ist eine Technologie (Spezifikation, Werkzeug), die

- den Entwurf dynamischer Web-Seiten unterstützt, bei denen
- Java-Programmcode in statischem HTML-Code eingebettet ist.
- JSP-spezifische Tags unterstützen dabei die Trennung zwischen Programmlogik und Seiten-Layout bzw. Benutzungsschnittstelle.

Da JSP-Seiten beliebigen eingebetteten Java-Programmcode enthalten können, könnte die gesamte Ausführungslogik auch darin enthalten sein, mit ähnlichen Nachteilen wie oben für reine Servlet-Lösungen genannt. Dieser Java-Anteil sollte also möglichst gering sein. Trotzdem haben wir auch hier eine Mischung aus HTML und Java. Im Gegensatz zur Servlet-Lösung ist hier HTML *außen* und Java *eingebettet*, was sich als deutlich günstiger als die umgekehrte Wahl herausstellt.

Eine JSP-Seite ist ein textbasiertes Dokument, das aus oben genannten Gründen auch mit geringen Programmierkenntnissen erstellt werden kann. Gegenüber einer ausschließlich Servletbasierten Lösung ergibt sich so der weitere Vorteil, dass auch Web-Designer mit nur geringen Programmierkenntnissen ein derartiges Dokument erstellen können. Dies ermöglicht eine sinnvolle Arbeitsteilung zwischen Entwicklern und Web-Designern, was sicherlich auch zu der raschen Verbreitung der JSP-Technologie beigetragen hat.

Im Rahmen unserer Architektur ist eine JSP-Seite eine Web-Komponente, die durch den Web Container des Application Servers verwaltet wird und bei dem ersten Request, der diese JSP-Seite anfordert, in ein Servlet transformiert wird. Wurde die JSP-Seite bereits zuvor durch einen Request angefordert, wird direkt das bereits existierende, auf dieser JSP-Seite basierende Servlet aufgerufen.

9.1. Request-Ablauf

Analog zu Kapitel 8 über Servlets gibt dieser Abschnitt einen Überblick über die Aufgaben, die von einem Servlet im Zusammenspiel mit einer JSP-Seite übernommen werden, wenn ein Client eine Request-Nachricht an den Java EE-Server sendet. Abbildung 9.1 stellt den Ablauf eines Standard-Requests dar. Dabei wird wieder davon ausgegangen, dass der Browser per POST-Methode ein Formular abschickt und als Antwort ein HTML-Dokument an den Client zurückgesendet wird. Neben einem Servlet, das für die Annahme des Requests zuständig ist, wird also eine JSP-Seite zur Erzeugung des Ausgabedokumentes eingesetzt. Für eine technisch korrekte Darstellung müsste die JSP-Seite eigentlich durch ein Servlet ersetzt werden, da sie beim ersten Aufruf in ein Servlet übersetzt wird. Das Response-Objekt und das `PrintWriter`-Objekt sind zudem nicht aufgeführt, da diese Objekte direkt durch den Web Container verwaltet werden.

Bis einschließlich des Aufrufs des Anwendungskerns stimmt dieser Ablauf mit dem Ablauf aus Abbildung 8.1 überein. Danach werden die Daten, die die JSP-Seite zum Erstellen des HTML-Dokumentes benötigt – z.B. die Ergebnisse der Anwendungskernaufrufe – mit der Methode `setAttribute(...)` im Request-Scope (Objekt der Klasse `HttpServletRequest`) abgelegt. Anschließend wird über dieses Objekt ein Objekt der Klasse `RequestDispatcher` angefordert, das die Weiterleitung vom Servlet zur JSP-Seite ermöglicht. Durch den Aufruf der Methode `forward(...)` des `RequestDispatcher`-Objektes wird dieses dazu veranlasst die Methode `_jspService(...)` aufzurufen. Die `_jspService(...)`-Methode ist Teil des Servlets, das aus der Übersetzung der JSP-Seite entsteht. Sie ist mit der `service(...)`-Methode eines „echten“ Servlets vergleichbar. Bei der Übersetzung der JSP-Seite in das entsprechende Servlet werden der statische Anteil in Form von `println(...)`-Methodenaufrufen und der dynamische Anteil, z.B. Ausdrücke und benutzerdefinierte Tags (siehe Abschnitte 9.5 und 9.6), in Form von regulärem Java-Code in die `_jspService(...)`-Methode übernommen. Die Übersetzung wird vom Web Container durchgeführt.

Um die Ergebnisse des Anwendungskernaufrufes in das Ausgabedokument übernehmen zu können, liest das aus der JSP-Seite resultiertende Servlet die im Request-Scope abgelegten Attribute mit der Methode `getAttribute(...)`. Nachdem alle notwendigen Operationen seitens dieses Servlets durchgeführt wurden, schickt das `RequestDispatcher`-Objekt die Response vom Web Container an den Browser.

9.2. Template-Text, JSP-Kommentare und JSP-Direktiven

In den Abschnitten 9.2 bis 9.6 werden verschiedene Konstrukte, die durch die JavaServer Pages Technologie zur Verfügung stehen, vorgestellt. Mit ihnen können z.B. Metadaten angegeben, statischer Text formuliert sowie dynamisch erzeugter Inhalt angezeigt werden, aber auch Berechnungen, Datenbankzugriffe, Anwendungskernaufrufe, usw. durchgeführt werden.

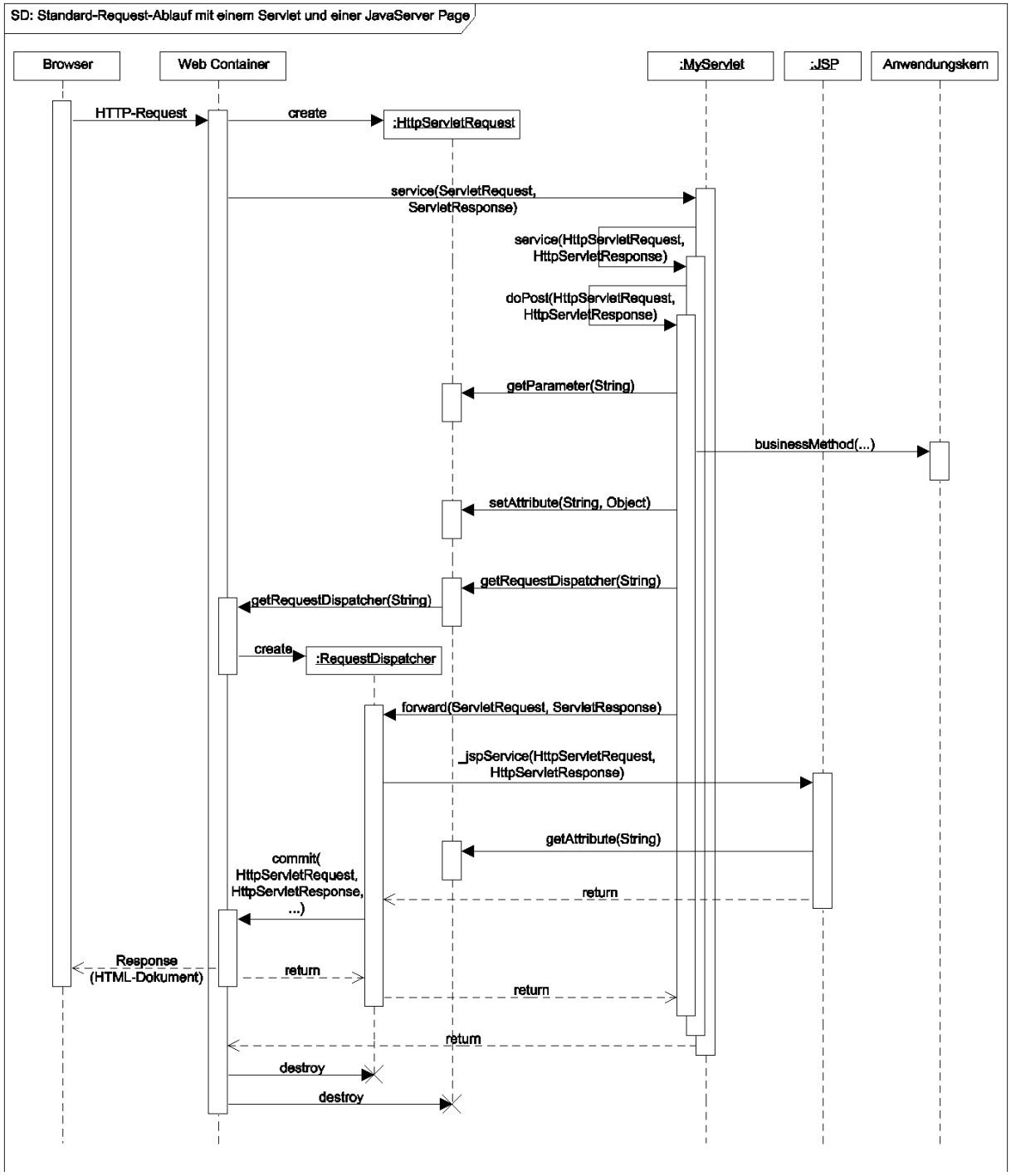

Abbildung 9.1.: Standard-Request-Ablauf mit einem Servlet und einer JSP-Seite

9.2.1. Template-Text

Als *Template-Text* (engl. template = Vorlage, Schablone) wird jede Form von statischem Text, der an den Client gesendet wird, bezeichnet. Bei HTML zählen z.B. normaler Text, HTML-Tags und HTML-Kommentare dazu. Eine JSP-Seite, die nur Template-Text enthält, sieht demzufolge aus wie ein normales HTML-Dokument (siehe Codeausschnitt 9.1).

```
1 <html>
2 <head><title>Dies ist eine JSP-Seite</title></head>
3   <body>
4     <!-- Dies ist ein HTML-Kommentar -->
5     <h1>Hello World</h1>
6   </body>
7 </html>
```

Listing 9.1: Eine JSP-Seite, die nur aus Template-Text besteht

9.2.2. JSP-Kommentar

Ein JSP-Kommentar ist ein Entwickler-Kommentar, der beim Übersetzen der JSP-Seite unberücksichtigt bleibt und somit – im Gegensatz zu einem HTML-Kommentar – nicht Teil der an den Client gesendeten Response ist. Ein JSP-Kommentar wird zwischen `<%-- und --%>` eingefügt, z.B. `<%-- Ich bin ein Kommentar --%>`.

9.2.3. JSP-Direktive

Eine JSP-Direktive enthält Informationen bzw. Anweisungen, die der Web-Container in die Verarbeitung einer JSP-Seite mit einbezieht. Sie können als Nachrichten an diesen verstanden werden. Mit Hilfe einer JSP-Direktive lassen sich z.B. Java-Klassen und Java-Pakete importieren, sodass diese innerhalb des durch die JSP-Seite erzeugten Servlets genutzt werden können. Auch sind die Einbindung weiterer JSP-Seiten oder HTML-Fragmente aus anderen Dateien und die Einbindung von Tag-Bibliotheken (Tag Library, siehe Abschnitt 9.6) möglich.

Eine JSP-Direktive hat immer die Form

`<%@ directive attribut1="wert1" attribut2="wert2"... %>`,

wobei für *directive*, *attribut* und *wert* konkrete Angaben einzusetzen sind. Tabelle 9.1 zeigt die drei möglichen Direktiven und einige wichtige Attribute und deren Bedeutungen.

Direktive	Attribute	Bedeutung
`page`	`import`	Import-Anweisung für die aus der JSP erzeugte Servlet-Klasse; mehrere importierte Klassen und Pakete können durch Kommata getrennt aufgeführt werden.
	`errorPage`	Relative URI einer anderen JSP-Seite, die angesteuert wird, falls bei Ausführung einer JSP-Seite eine Exception auftritt und die Verarbeitung nicht fortgeführt werden kann.
	`isErrorPage`	Erklärt eine JSP-Seite als Fehlerseite, auf die von einer anderen JSP-Seite über das Attribut `errorPage` verwiesen werden kann.
	`contentType`	Gibt den Dokumenttyp der Response an. Der Standardwert ist `text/html`.
`include`	`file`	Relative URI einer anderen JSP-Seite oder eines HTML-Fragments, die/das an der angegebenen Stelle zur Übersetzungszeit eingefügt wird
`taglib`	Bindet eine Tag-Bibliothek ein, deren Tags von dieser Stelle an in der JSP-Seite benutzt werden können.	
	`prefix`	Gibt ein frei wählbares Präfix für die Tags der Tag-Bibliothek an, um diese innerhalb der JSP-Seite ansprechen zu können.
	`uri`	Gibt eine URI an, die auf die Tag-Bibliothek verweist. Diese URI kann eine URL, eine URN oder ein Pfadname sein.

Tabelle 9.1.: Wichtige Direktiven

9.3. Scripting-Elemente

Mit Hilfe von Scripting-Elementen kann eine JSP-Seite dynamisch erzeugte Inhalte einbinden, aber auch Berechnungen, Datenbankzugriffe, Anwendungskernaufrufe usw. durchführen.

9.3.1. JSP-Deklaration

Mit einer JSP-Deklaration können Variablen und Methoden deklariert werden. Diese Variablen und Methoden werden, bei der Übersetzung der JSP-Seite in ein Servlet, Teil der Servlet-Klasse, und zwar außerhalb der `_jspService(...)`-Methode. Bei Variablendeklarationen gilt daher dieselbe Vorsicht bezüglich Thread-Sicherheit wie bei einem Servlet, da auch von einer JSP-Servlet-Klasse nur eine einzige Instanz erzeugt wird. JSP-Deklarationen werden zwischen `<%!` und `%>` eingefügt, z.B. `<%! private int ergebnis; %>`.

9.3.2. JSP-Ausdruck

Ein JSP-Ausdruck ist ein auswertbarer Java-Ausdruck. Der JSP-Ausdruck wird in dem an den Client gesendeten Antwortdokument durch seinen Wert ersetzt. Ein JSP-Ausdruck wird zwischen `<%=` und `%>` eingefügt. Zum Beispiel hat der Ausdruck `<%= (10 + 20) %>` den Wert 30. Der Ausdruck `<%= "Das Datum: "+ (holeDatum())%>` verknüpft eine statische Zeichenkette mit der Datumsausgabe. Es muss darauf geachtet werden, dass der angegebene Gesamtausdruck in eine Zeichenkette umgewandelt werden kann (soll z.B. das Ergebnis einer Rechnung ausgegeben werden, so wird die resultierende Zahl implizit in eine Zeichenkette umgewandelt). Außerdem darf zwischen `<%=` und `%>` nur ein einziger Ausdruck stehen.

9.3.3. JSP-Scriptlet

Ein JSP-Scriptlet ist eine Java-Anweisung, mit der z.B. eine Berechnung, ein Datenbankzugriff oder ein Anwendungskernaufruf durchgeführt werden kann. Es wird nur dann eine für den Client sichtbare Ausgabe produziert, wenn die Java-Anweisung dies explizit mittels der Methode `println(...)` veranlasst. Ein JSP-Scriptlet wird zwischen `<%` und `%>` eingefügt, z.B. `<% if (Math.random() < 0.5) out.println("gewonnen!"); %>`. JSP-Deklarationen und JSP-Ausdrücke lassen sich offenbar meist durch JSP-Scriptlets ersetzen. Was sind also die Vorteile von JSP-Ausdrücken und JSP-Deklarationen gegenüber JSP-Scriptlets? Ein JSP-Ausdruck vereinfacht die Ausgabe, denn die Methode `println(...)` muss nicht genutzt werden. Allerdings darf kein Semikolon am Ende des Ausdrucks stehen. JSP-Deklarationen können Variablen, Konstanten, Klassen und Methoden deklarieren. Variablen, Konstanten und Klassen können auch innerhalb eines JSP-Scriptlets deklariert werden, nicht jedoch Methoden. Bei der Übersetzung einer JSP-Seite in eine Servlet-Klasse werden alle innerhalb einer JSP-Deklaration definierten Elemente zu Variablen der Servlet-Klasseninstanz. Alle definierten Methoden werden zu Methoden der Servlet-Klasseninstanz. Alle in einem JSP-Scriptlet enthaltenen Deklarationen und Anweisungen werden in der Methode `_jspService(...)` der Servlet-Klasse gekapselt. Z.B. werden in einer JSP-Scriptlet deklarierte Variablen zu lokalen Variablen dieser Methode. Da eine Methode (in diesem Fall die Methode `_jspService(...)`) keine Methodendefinition enthalten darf, ist also eine Methodendefinition innerhalb eines JSP-Scriptlets nicht erlaubt.

Beispiel

Codeausschnitt 9.2 zeigt eine JSP-Seite, die alle in diesem und vorigem Abschnitt behandelten JSP-Konstrukte enthält.

```
1 <%-- Dies ist JSP-Kommentar, der in der Response nicht mehr existiert --%>
2 <!-- Dies ist normaler HTML-Kommentar, der in der Response mitgesendet,
      aber vom Browser nicht dargestellt wird -->
3 <html>
4   <head><title></title></head>
5   <body>
6     <h1>
7       Der Münzwurf zeigt
8       <% if (Math.random() < 0.5) { %>
```

```
9           &amp;quot;Kopf&amp;quot;.
10        <% } else { %>
11          &amp;quot;Zahl&amp;quot;.
12        <% } %>
13      </h1>
14      <%@ page import="java.util.Random"errorPage="error.jsp" %>
15      <%! private int wuerfeln() {
16            return (new Random()).nextInt(6) + 1;
17          } %>
18      <h2>Der Würfel ist gefallen und zeigt
19          <% out.println(wuerfeln()); %> Auge(n).</h2>
20      <h3>Der Würfel ist noch mal gefallen und zeigt jetzt
21          <%= wuerfeln() %> Auge(n).</h3>
22      </body>
23  </html>
```

Listing 9.2: JSP-Seite mit Template-Text, JSP-Kommentar, JSP-Direktive und Scripting-Elementen

Die Browser-Darstellung des aus der JSP-Seite resultierenden HTML-Dokumentes ist in Abbildung 9.2 zu sehen.

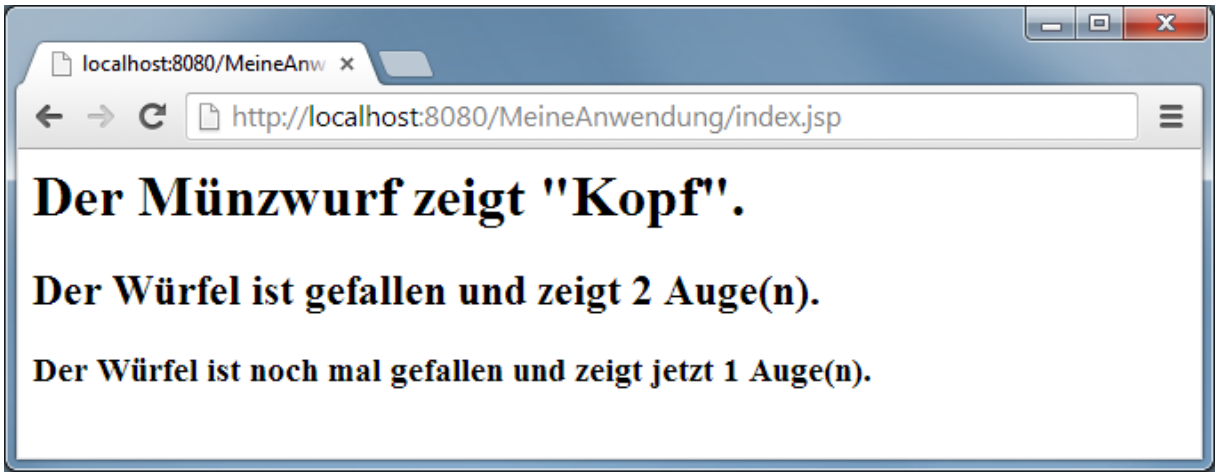

Abbildung 9.2.: Darstellung des Ausgabedokumentes

Den Scripting-Elementen stehen eine Reihe impliziter Objekte zur Verfügung. In Codeausschnitt 9.2 wurde bereits das implizite Objekt out für die Ausgabe der Response benutzt. Die wichtigsten impliziten Objekte sind in Tabelle 9.2 dargestellt.

9.4. JSP-Aktionen

Die JavaServer Pages Technologie ermöglicht eine konsequentere Trennung von Benutzungsschnittstelle und Programmlogik durch die Verwendung von Scripting-Elementen, die in statische HTML-Dokumente eingefügt werden und die Verbindung zur Programmlogik darstellen.

Objekt	Klasse/Interface	Beschreibung
`pageContext`	`PageContext`	Liefert das `PageContext`-Objekt, das den Page Scope darstellt und Zugriff auf alle anderen in dieser Tabelle angegebenen Objekte bietet.
`request`	`HttpServletRequest`	Das Request-Objekt, das den Request Scope darstellt.
`response`	`HttpServletResponse`	Das Response-Objekt, das die Response (Ausgabe) enthält.
`session`	`HttpSession`	Das Session-Objekt, das den Session Scope darstellt.
`application`	`ServletContext`	Das Context-Objekt, das den Application Scope darstellt.
`out`	`JspWriter`	Ausgabeobjekt der Response, mit dem über die Methode `println(...)` wie bei einem Servlet Ausgaben erzeugt werden können.

Tabelle 9.2.: Implizite Objekte für Scripting-Elemente

So können größere Java-Programmteile in Scripting-Elementen „gekapselt“ werden, aber Web-Designer ohne Programmierkenntnisse müssen sich wieder mit Java-Programmcode „herumschlagen“.

JSP-Aktionen decken einen Teil der Funktionalität von Scripting-Elementen ab, ihre Syntax entspricht allerdings der XML-Syntax, weshalb sie für Nicht-Programmierer einfacher zu handhaben sind. Mit JSP-Aktionen können u.a. Objekte erzeugt und bearbeitet, auszuwertende Ausdrücke erstellt, Requests an andere JSP-Seiten oder Servlets weitergeleitet oder Java-Applets integriert werden. Eine JSP-Aktion besitzt die Form

```
<jsp:action attribut1="wert1" attribut2="wert2"... />
```

oder die Form

```
<jsp:action attribut1="wert1" attribut2="wert2"... > ... </jsp:action>,
```

wobei `action`, `attribut` und `wert` durch konkrete Angaben zu ersetzen sind. JSP-Aktionen gliedern sich damit nahtlos in HTML-Code ein. Im Folgenden werden einige wichtige JSP-Aktionen kurz erläutert.

Die JSP-Aktion

```
<jsp:forward page="..."/>
```

bewirkt den Aufruf der `forward(...)`-Methode des `RequestDispatcher`. Einziges Attribut ist `page`, das den Pfad der Zielressource angibt.

Die JSP-Aktion

```
<jsp:include page="..." flush="..."/>
```

bewirkt den Aufruf der `include(...)`-Methode des RequestDispatcher. Das Attribut `page` gibt wieder den Pfad der Zielressource an. Mit dem Attribut `flush`, das den Wert `true` oder `false` annehmen kann, wird festgelegt, ob vor dem Inkludieren die bisher erfolgte Ausgabe schon an den Client gesendet werden soll oder nicht. Der Unterschied zwischen dieser JSP-Aktion und der JSP-Direktive `<%@include file="..."%>` ist der Zeitpunkt und die Art des Einbindens der angegebenen Ressource. Die JSP-Aktion bindet die Ausgabe der inkludierten Ressource erst zur Anfragezeit in die aktuelle JSP-Seite ein. Die JSP-Direktive bindet den Code der inkludierten Ressource bereits zur Übersetzungszeit in die aktuelle JSP-Seite ein. Durch das Einbinden des Codes zur Übersetzungszeit ist es der inkludierten Ressource möglich, Variablen, die in der aufrufenden JSP-Seite deklariert sind, zu nutzen und zu manipulieren. Umgekehrt hat die aufrufende JSP-Seite Kenntnis von Details der inkludierten Ressource. Bei der JSP-Aktion sind die inkludierte Ressource und die aufrufende JSP-Seite voneinander unabhängig und haben keine Kenntnis voneinander.

Die JSP-Aktion

```
<jsp:useBean ... scope="page|request|session|application"/>
```

instanziiert ein neues Objekt einer JavaBean[1]. Mit `class` wird der voll qualifizierte Klassenname[2] angegeben, von dessen Klasse das Objekt erzeugt werden soll. Das Attribut `id` bezeichnet den Variablennamen des neuen Objekts und `scope` gibt den Scope an, in dem das Objekt abgelegt werden soll. Wird kein Sope angegeben, so wird standardmäßig im Page Scope gespeichert. Beispielsweise wird mit

```
<jsp:useBean id="beo" class="lebewesen.tiere.voegel.Beo"/>
```

eine neue Instanz von Beo erzeugt und im Page Scope unter dem Namen `beo` abgelegt, sofern nicht bereits ein Objekt mit diesem Namen existiert.

Mit der JSP-Aktion

```
<jsp:setProperty name="..." property= value="..."/>
```

können Attribute (properties) von JavaBeans manipuliert werden, die in einem der vier Scopes liegen. Das Attribut name gibt den Namen an, unter dem die JavaBean im Scope abgelegt ist, `property` bezeichnet den Namen des zu ändernden Attributes und value gibt den neuen Wert dieses Attributes an. Beispielsweise wird mit

```
<jsp:setProperty name="aktie" property="kurs"value="120"/>
```

der Wert des Attributes `kurs` des Objektes `aktie` auf 120 gesetzt.

[1] Siehe Abschnitt 8.3.1 - JavaBeans

[2] Ein voll qualifizierter Klassenname enthält auch den Paketnamen, in dem sich die Klasse befindet und die darüber liegende Struktur

Die JSP-Aktion

```
<jsp:getProperty name="..." property="..."/>
```

gibt Attributwerte von JavaBeans aus, die in einem der vier Scopes gespeichert sind. Das Attribut `name` gibt wieder den Namen an, unter dem die JavaBean im Scope abgelegt ist, und property bezeichnet das zu lesende Attribut. Beispielsweise wird mit

```
<jsp:getProperty name="auto" property="hersteller"/>
```

der Wert des Attributes `hersteller` des Objektes `auto` ausgegeben.

9.5. Expression Language

Scripting-Elemente erlauben es, Berechnungen, Anwendungskernaufrufe und sogar Datenbankzugriffe aus einer JSP-Seite heraus vorzunehmen. Letztlich kann mit ihnen sogar Anwendungslogik implementiert werden, die bei einer vernünftigen Softwarearchitektur, wie der Java EE-Architektur, in die Verantwortung des Anwendungskerns fällt. Scripting-Elemente sind daher kritisch zu hinterfragen, am besten sollten sie gar nicht benutzt werden. Dasselbe gilt für die JSP-Aktionen, die Objekte erzeugen, manipulieren oder Weiterleitungen an andere JSP-Seiten vornehmen. Schränkt man eine JSP-Seite sinnvollerweise dahingehend ein, dass sie nur die Präsentation übernimmt und über HTML hinausgehende Techniken ausschließlich zur Darstellung von dynamisch erzeugtem Inhalt einsetzt, muss die JSP-Seite im Grunde lediglich die in einem Scope abgelegten Ergebnisse von Aufrufen des Anwendungskerns auslesen können. Genau diese Stoßrichtung unterstützen die *Expression Language* (EL) und die mächtigere *JavaServer Pages Standard Tag Library* (JSTL). Mit der EL und der JSTL beschäftigen sich dieser und der folgende Abschnitt.

Mit der Expression Language (EL), deren Spezifikation in der aktuellen Version 3.0 zur Verfügung steht (vgl. [JSP/Spec]), kann man von einer JSP-Seite auf in einem Scope gespeicherte Objekte und, sofern dies JavaBeans sind, auch auf deren Properties (Eigenschaften) zugreifen. Die EL verfügt über die in Java bekannten relationalen, logischen und arithmetischen Operatoren und – ähnlich wie Scripting-Elemente – auch über implizite Objekte. Die EL lässt sich darüber hinaus um sogenannte Funktionen erweitern.

Die EL richtet sich in erster Linie an Web-Designer. Ihre Syntax ist daher bewusst einfach gehalten und ihr Funktionsumfang eingeschränkt. So ist es z.B. nicht ohne weiteres möglich, Objekte zu erzeugen oder zu modifizieren.

Ein EL-Ausdruck hat die Form

```
$<Ausdruck>,
```

wobei `<Ausdruck>` den auszuwertenden Ausdruck angibt. Ein solcher Ausdruck darf sowohl im Template-Text einer JSP-Seite als auch in Attributwerten von Tags (der folgende Abschnitt 9.6 zur JSTL enthält Beispiele) eingesetzt werden. Ein mit einem Dollarzeichen eingeleiteter

EL-Ausdruck wird direkt bei der Übersetzung der JSP-Seite ausgewertet.

Zusätzlich unterstützt die EL auch Ausdrücke der Form

```
#<Ausdruck>
```

für verzögerte Auswertung, die ausschließlich in bestimmten Attributwerten stehen dürfen und vornehmlich im Zusammenhang mit JavaServer Faces[3] eingesetzt werden. Eine verzögerte Auswertung bedeutet, dass alle EL-Ausdrücke, die mit einem Doppelkreuz anfangen, erst dann ausgewertet werden, wenn der Wert des Ausdrucks benötigt wird. Solche Ausdrücke dürfen im Template-Text von JSP-Seiten nicht vorkommen, die JSP-Seite lässt sich sonst nicht übersetzen. Möchte man die Zeichen `#{}` ausgeben, ohne dass sie als EL-Ausdruck interpretiert werden, muss ein Backslash vorangestellt werden: \#{}.

9.5.1. Zugriff auf Objekte

Die Expression Language sucht die in einem Ausdruck angesprochenen Objekte zunächst im Page Scope, danach im Request Scope, dann im Session Scope und schließlich im Application Scope. Der erste „Treffer“ wird zurückgeliefert.

Auf Properties von JavaBeans oder Elemente von Listen, Arrays oder *Maps*[4], die in einem Scope liegen, kann über „.“ oder „[ ]“ zugegriffen werden. Die folgenden Beispiele erläutern die verschiedenen Zugriffsvarianten.

Um den Vornamen einer Person (die z.B. als JavaBean modelliert ist) auszugeben, die als `kunde` in einem Scope abgelegt ist, kann entweder

`${kunde.firstName}` oder `${kunde["firstName"]}`

benutzt werden.

Geschachtelte Objekte (z.B. ein Objekt `adresse` einer JavaBean als Property der JavaBean `kunde`) können entweder über

`${kunde.adresse.strasse}` oder über `${kunde["adresse"]["strasse"]}`

angesprochen werden.

Auf Elemente von Maps kann über den Wert des Schlüssels zugegriffen werden. Um z.B. das Element anzuzeigen, das unter dem Schlüssel „deutschland“ in der Map `hauptstaedte` einsortiert ist, kann entweder

`${hauptstaedte.deutschland}` oder `${hauptstaedte["deutschland"]}`

[3] JavaServer Faces (JSF) ist ein in der Java EE-Spezifikation enthaltenes ereignisbasiertes Framework für die Benutzungsschnittstelle von Web-Anwendungen, auf das im späteren Kurstext noch eingegangen wird.

[4] Eine Map enthält Objekte in strukturierter Form und ist wie ein Wörterbuch aufgebaut. Jeder Eintrag einer Map besteht aus einem Schlüssel (key) und dem zugehörigen Wert (value), dabei darf jeder Schlüssel nur einmal innerhalb einer Map vorkommen.

benutzt werden. Die erste Variante ist nur erlaubt, wenn der Map-Schlüsselwert der Java-Namenskonvention für Variable entspricht. So darf der Schlüsselwert nicht mit einer Zahl beginnen, da Variablennamen in Java nicht mit einer Zahl beginnen.

Auf die Elemente von Arrays oder Listen kann nur über die Variante mit den eckigen Klammern zugegriffen werden, wahlweise mit oder ohne Hochkommata:

`${warenkorb[0]}` oder `${warenkorb["0"]}`.

Als Index kann auch der Name eines in einem Scope abgelegten Objektes verwendet werden. Damit ist es möglich, einen Map-Schlüsselwert zur Anfragezeit zu berechnen.

Ist der String „deutschland“ unter dem Namen `land` in einem Scope abgelegt, dann führen die Zugriffe

`${hauptstaedte[land]}` und `${hauptstaedte["deutschland"]}`

zum selben Ergebnis.

Eine große Vereinfachung bringt die sogenannte implizite Fehlerbehandlung mit sich. Gegeben sei noch einmal der Ausdruck `$kunde.adresse.strasse`. Wenn `strasse` den Wert `null` besitzt, und sogar wenn kunde den Wert null besitzt[5], wird keine Fehlermeldung erzeugt, sondern einfach `null` zurückgeliefert und nichts dargestellt. Das erspart das mühsame Abfragen aller Objekte und Attribute auf `null`, bevor der EL-Ausdruck ausgewertet wird. Bei Arrays oder Listen wird ebenfalls `null` zurückgeliefert, falls der angegebene Index nicht im zulässigen Bereich liegt.

9.5.2. Relationale, logische und arithmetische Operatoren

Alle relationalen, logischen und arithmetischen Operatoren, die in Java zulässig sind, werden auch von der Expression Language unterstützt. Bei den relationalen und logischen Operatoren gibt es immer auch eine textuelle Alternative, die dem Nicht-Programmierer den Umgang erleichtern soll. Für == gibt es beispielsweise `eq`, für `&&` gibt es `and`, für `!` gibt es `not` usw. Der folgende Ausdruck ist z.B. ein gültiger EL-Ausdruck, der zu `true` ausgewertet wird:

```
${(1 + 2) eq "3"}
```

Dieser Ausdruck zeigt außerdem, dass die EL `implizite Typkonvertierungen` vornimmt, sofern dies möglich ist. Eine implizite Typkonvertierung ist eine Umwandlung (Konvertierung) eines Datentyps in einen anderen. Implizite Typkonvertierungen müssen nicht explizit programmiert werden und tauchen somit nicht im Programmcode auf. Die Programmiersprache Java dagegen führt implizite Typkonvertierungen nur dann durch, wenn sie ohne Informationsverlust erfolgen können. Z.B. kann die Umwandlung einer Integer-Zahl (keine Nachkommastellen) in eine Double-Zahl (mit Nachkommastellen) ohne Informationsverlust erfolgen, umgekehrt würden die Nachkommastellen der Double-Zahl verloren gehen.

[5] In Java bezeichnet `null` einen Nullwert, der das Fehlen eines Wertes signalisiert.

Objekt	Klasse/Interface	Beschreibung
`pageContext`	`PageContext`	Liefert das `PageContext`-Objekt, das den Page Scope darstellt.
`pageScope`	`Map`	beinhaltet alle im Page Scope abgelegten Objekte.
`requestScope`	`Map`	Beinhaltet alle im Request Scope abgelegten Objekte.
`sessionScope`	`Map`	Beinhaltet alle im Session Scope abgelegten Objekte.
`applicationScope`	`Map`	Beinhaltet alle im Application Scope abgelegten Objekte.
`initParam`	`Map`	Enthält für jeden Kontextinitialisierungsparameter den dazugehörigen Wert als Zeichenkette.
`param`	`Map`	Enthält für jeden Anfrageparameter den dazugehörigen Wert als Zeichenkette.
`paramValues`	`Map`	Enthält für jeden Anfrageparameter die dazugehörigen Werte als Zeichenketten-Array.
`header`	`Map`	Enthält für jeden Header den zugehörigen Wert als Zeichenkette.
`headerValues`	`Map`	Enthält für jeden Header die zugehörigen Werte als Zeichenketten-Array.
`cookie`	`Map`	Enthält für jeden Cookie das zugehörige Objekt vom Typ `Cookie`. Existieren mehrere Cookies mit demselben Namen, dann wird ein beliebiger zurückgeliefert.

Tabelle 9.3.: Implizite Objekt von JSP-Seiten für die Expression Language

Zusätzlich gibt es noch den Operator `empty`, der genau dann `true` zurückliefert, wenn das Argument `null` oder der leere String, die leere Map, das leere Array oder die leere Liste ist. Ansonsten wird `false` zurückgegeben. Ein EL-Ausdruck mit dem `emtpy`-Operator kann z.B. folgendermaßen aussehen: `${empty variable1}`.

9.5.3. Implizite Objekte

Auch für EL-Ausdrücke in JSP-Seiten stehen `implizite Objekte` zur Verfügung, auf die ohne explizite Erzeugung immer zugegriffen werden kann (s. Tabelle 9.3)

Mittels `${param["benutzername"]}` wird z.B. der im Request übermittelte Parameter `benutzername` ausgegeben.

Beispiel

Codeausschnitt 9.3 zeigt eine JSP-Seite, die die Lieferanschrift eines Kunden ausgibt. Mit Hilfe der Expression Language werden dazu die Attributwerte des Objektes kunde in die Ausgabe eingesetzt, indem auf dem Objekt `kunde` die Getter `getNachname()`, `getVorname()`, `getStrasse()`, `getPlz()` und `getStadt()` ausgeführt werden.

```
1  <html>
2    <head><title>Lieferanschrift</title></head>
3    <body>
4    <h3>Ihre Lieferanschrift</h3>
5    <p>Name:          ${kunde.nachname}<br>
6       Vorname:       ${kunde.vorname}<br>
7       Stra&szlig;e:  ${kunde.strasse}<br>
8       PLZ:           ${kunde.plz}<br>
9       Ort:           ${kunde.stadt}</p>
10   </body>
11 </html>
```

Listing 9.3: Einsatz der Expression Language in einer JSP-Seite

Dieses Beispiel zeigt den Regelfall, dass sich der in der EL zu verwendende Property-Bezeichner einfach aus dem Bezeichner des anzusprechenden Getters ableitet, indem das Präfix `get` weggelassen und der darauf folgende Buchstabe zu einem Kleinbuchstaben konvertiert wird. Ist allerdings der zweite Buchstabe nach dem Präfix `get` ebenfalls ein Großbuchstabe, so wird auch der erste nicht verändert. Würde in obigem Beispiel der Getter zur Postleitzahl also nicht `getPlz()`, sondern `getPLZ()` lauten, so wäre in der EL der Ausdruck `${kunde.PLZ}` zu verwenden, nicht etwa `${kunde.pLZ}`.

9.5.4. Funktionen

Die EL lässt sich um (selbstdefinierte) *Funktionen* erweitern. Ein vordefinierter Standardsatz von Funktionen, der in erster Linie der Zeichenketten-Verarbeitung dient, ist in der *JavaServer Pages Standard Tag Library* enthalten. Diese Funktionen werden am Ende des nächsten Abschnitts behandelt.

9.6. JavaServer Pages Standard Tag Library

Aus einer JSP-Seite heraus kann man mit Hilfe der Expression Language auf in einem Scope gespeicherte Objekte zugreifen, um sie dann für die Darstellung aufzubereiten. Die EL richtet sich in erster Linie an Nicht-Programmierer und ist daher bewusst einfach gehalten. Sollte ihre Ausdrucksmächtigkeit nicht ausreichen, bietet sich (zusätzlich) die Verwendung der mächtigeren, aber auch komplexeren *JavaServer Pages Standard Tag Library* (JSTL) an. Die durch die JSTL verfügbaren Tags erlauben Operationen wie z.B. Iterationen über Mengen von Objekten, beding-

core	Funktion
`core`	Tags für die Erstellung von URLs, für einfache Schleifen, bedingte Anweisungen, Ausnahmebehandlungen, etc.
`xml`	Tags für die Bearbeitung von XML-Dokumenten
`fmt`	Tags für Internationalisierung sowie Datums- und Zahlenformate
`sql`	Tags für das Einbinden relationaler Datenbanken
`functions`	Funktionen für die Expression Language

Tabelle 9.4.: Bibliotheken der JSTL

te Auswertungen oder spezielle Formatierungen der Ausgabe. Die JSTL ist eine so genannte benutzerdefinierte Tag Library und besitzt eine eigene Spezifikation [JSTL/Spec], die nicht in der JSP-Spezifikation enthalten ist.

Tabelle 9.4 zeigt die fünf Bibliotheken, in die sich die JSTL gliedert.

Um Tags der JSTL in einer JSP-Seite verwenden zu können, muss die entsprechende Bibliothek über die `taglib`-Direktive (vgl. Abschnitt 9.2) am Anfang der JSP-Seite bekannt gemacht werden. Die Abkürzung „taglib“ steht für Tag Library. Die Direktive benötigt eine URI, die auf die Bibliothek verweist, und ein Präfix, das den Namensraum festlegt, mit dem die Bibliothek anschließend in der JSP-Seite angesprochen werden kann. Unter einem Namensraum versteht man einen Bereich, in dem ein oder mehrere bestimmte Bezeichner verwendet werden können.

Im Folgenden wird nur die `core`-Bibliothek behandelt, die den Kern der JSTL bildet.

Die URI für die Core-Bibliothek lautet

```
http://java.sun.com/jsp/jstl/core
```

Das Präfix kann frei gewählt werden, als Standard für die Core-Bibliothek haben sich allerdings `c` und `core` herausgebildet. Die `taglib`-Direktive, welche die Core-Bibliothek der JSP-Seite bekannt macht, sieht demnach so aus:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
```

Anschließend können alle Tags der Core-Bibliothek in der JSP-Seite verwendet werden, indem den Tags das Präfix c mit einem Doppelpunkt vorangestellt wird.

Beim *URL-Rewriting*, von dem der Web Container ausgeht, wenn Cookies nicht zur Verfügung stehen, muss die Session-ID an jede URL angehängt werden, um die ID bei einem Request nicht zu verlieren. Dies kann mit Scripting-Elementen und der Methode `encodeURL(...)` der Schnittstelle `HttpServletResponse` (vgl. Abschnitt 8.4) durchgeführt werden. Um auf Scripting-Elemente verzichten zu können, bietet die `core`-Bibliothek das `url`-Tag an. Dieses Tag überprüft u.a., ob URL-Rewriting notwendig ist, und führt dieses ggf. durch. Zusätzlich wird der Context-Pfad der Web-Anwendung der URL vorangestellt. Codeausschnitt 9.4 zeigt ein Beispiel für das `url`-Tag. Wenn die Web-Anwendung „Systemanmeldung“ heißt, würde die URL in Codeausschnitt 9.4 zu

```
/Systemanmeldung/MeinServlet;jsessionid=C9E506D...
```

umgeschrieben.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html><head><title></title></head>
  <body>
    ...
    <form method="post" action="<c:url value='/MeinServlet'/>">
      ...
    </form>
    <a href="<c:url value='/MeinServlet'/>">Dies ist ein Link</a>
    ...
  </body>
</html>
```

Listing 9.4: Das url-Tag

Die Core-Bibliothek bietet zwei Möglichkeiten für bedingte Auswertungen an. Das `if`-Tag führt den vom Tag umschlossenen Rumpf nur aus, wenn die Bedingung, die durch das Attribut `test` geprüft wird, zu *wahr* ausgewertet wird. Codeausschnitt 9.5 zeigt ein Beispiel für das `if`-Tag. Das Beispiel zeigt übrigens auch, dass die EL mit der JSTL sehr gut kombinierbar ist.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html><head><title></title></head>
  <body>
    ...
    <c:if test="${empty warenkorb}">
      Ihr Warenkorb ist zurzeit leer.
    </c:if>
    ...
  </body>
</html>
```

Listing 9.5: Das if-Tag der JSTL

Durch die Kombination der Tags `choose`, `when` und `otherwise` kann eine `if-then-else`-Anweisung simuliert werden. Zuerst wird die Bedingung des ersten `when`-Tags geprüft. Wird diese zu *wahr* ausgewertet, wird der Rumpf dieses Tags ausgeführt. Der restliche Teil des `choose`-Tags wird daraufhin nicht mehr ausgeführt. Falls die Bedingung des ersten `when`-Tags zu falsch ausgewertet wird, so wird der Rumpf nicht ausgeführt und es wird die Bedingung des nächsten `when`-Tags überprüft. Kann keine Bedingung zu *wahr* ausgewertet werden, wird nur der Rumpf des `otherwise`-Tags ausgeführt, falls dieses vorhanden ist.

Codeausschnitt 9.6 zeigt ein Beispiel für die Tags `choose`, `when` und `otherwise`.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html><head><title></title></head>
  <body>
    ...
    <c:choose>
      <c:when test="${user.powerSeller}">
        Sonderauslosung für alle PowerSeller nächste Woche!
      </c:when>
      <c:when test="${user.seller}">
        Wir haben ein neues Prämienprogramm für Sie!
      </c:when>
      <c:otherwise>
        Werden Sie noch heute Mitglied!
      </c:otherwise>
    </c:choose>
    ...
  </body>
</html>
```

Listing 9.6: Die choose-, when-, otherwise-Tags der JSTL

Als Schleifenkonstruktor fungiert das Tag `<c:forEach>`. Mit diesem Tag kann eine Menge von Objekten, etwa ein Array, eine Liste oder eine Map, durchlaufen werden. Mit dem Attribut `items` wird die in einem beliebigen Scope abgelegte Menge angegeben. Das optionale Attribut `var` legt den Namen einer Laufvariable fest, über welche in jedem Schleifendurchlauf das jeweils aktuelle Element angesprochen werden kann. In Codeausschnitt 9.7 wird eine Menge von Produkten komplett durchlaufen und so die ID und der Name von jedem Produkt ausgegeben.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html><head><title></title></head>
  <body>
    ...
    <c:forEach items="${produkte}" var="produkt">
      ${produkt.id} ${produkt.name}<br>
    </c:forEach>
    ...
  </body>
</html>
```

Listing 9.7: Das forEach-Tag der JSTL zum Mengendurchlauf

Mit den Attributen `begin` und `end` des `forEach`-Tags kann alternativ auch eine typische `for`-Schleife mit Start- und Endwert realisiert werden. Eine Schleife, die hundertmal „Hello World!“ ausgibt, zeigt Codeausschnitt 9.8.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html><head><title></title></head>
  <body>
    <c:forEach begin="1" end="100" var="i">
      ${i} - Hello World!<br>
```

```
6       </c:forEach>
7     </body>
8   </html>
```

Listing 9.8: Das forEach-Tag der JSTL als for-Schleife

Die `core`-Bibliothek bietet noch weitere Funktionen, wie Redirects, Ausnahmebehandlung, Einbinden externer Ressourcen (JSP-Seiten, HTML-Dokumente, etc.), die hier aber nicht näher erläutert werden.

Die JSTL bietet auch zusätzliche Funktionen für die Expression Language, die im Wesentlichen zur Zeichenketten-Verarbeitung gedacht sind. Der Aufruf einer solchen Funktion besitzt die Syntax

`ns:f(a1,a2,...,an),`

wobei `ns` den Namensraum bzw. das Präfix angibt, `f` den Funktionsnamen bezeichnet und `a1` bis `an` die Funktionsparameter darstellen.

Die Funktionen sind in der Function Tag Library der JSTL zusammengefasst, die mit der Direktive

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn"%>
```

bekannt gemacht werden muss. Als Präfix hat sich `fn` durchgesetzt.

Die Benutzung der Function Tag Library wird exemplarisch anhand der Funktion `startsWith(String, String)` in Kombination mit dem `if`-Tag erläutert. Die Funktion überprüft, ob der Wert des zweiten Parameters mit dem Anfang des Wertes des ersten Parameters übereinstimmt:

```
<c:if test='${fn:startsWith(produktname, "konserve")}' >...</c:if>
```

Sollten die Funktionen der EL und der JSTL einmal nicht ausreichen, dann können eigene Tag Libraries erstellt werden, in denen man die benötigten Funktionen mit Hilfe von Tags definieren kann. Auf das Erstellen solcher benutzerdefinierter Tag Libraries wird hier nicht näher eingegangen, sondern auf [Tu/Sa/Le] verwiesen. Ein Beispiel für eine benutzerdefinierte Tag Library ist die hier besprochene JSTL.

9.7. Beispiel: Systemanmeldung mit Servlet und JSP-Seiten

Zum Abschluss des Kapitels über die JavaServer Pages Technologie wird noch einmal das Beispiel der Systemanmeldung aus Abschnitt 8.5 aufgegriffen, bei dem jetzt für die eigentliche Bearbeitung ein Servlet und für die Ausgabe zwei JSP-Seiten eingesetzt werden. Das Anwendungsprogramm erhält den Namen SystemanmeldungJSP. Die erste JSP-Seite `signIn.jsp` hat nur die Aufgabe, das Anmeldeformular anzuzeigen. Beim Absenden des Formulars an den

Server sollen die Daten an das Servlet – der Einfachheit halber wieder `CheckSignInServlet` genannt – gesendet werden. Nach erfolgreicher Prüfung der Anmeldedaten soll die zweite JSP-Seite `signedIn.jsp` die Darstellung der Erfolgsseite übernehmen.

In Codeausschnitt 9.9 ist der Quelltext der JSP-Seite `signIn.jsp` aufgeführt (für die Darstellung durch einen Browser siehe Abbildung 8.3).

```
1  <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
2  <html>
3    <head>
4      <title>Beispiel Systemanmeldung</title>
5    </head>
6    <body>
7      <p><img src="images/LogoFeu.jpg"
8              alt="FernUniversit&auml;t in Hagen"></p>
9      <h1>Beispiel &quot;Systemanmeldung&quot;</h1>
10     <h3>aus dem Kurs 01796, Kurseinheit 3</h3>
11     <form method="POST"
12           action="<c:url value='/CheckSignInServlet'/>">
13       <table cellpadding="10"
14              style="border-collapse: collapse; border-width: 0">
15         <tr>
16           <td>Benutzername:</td>
17           <td><input type="text" name="username" size="20"></td>
18         </tr>
19         <tr>
20           <td>Passwort:</td>
21           <td><input type="password" name="password" size="20"></td>
22         </tr>
23       </table>
24       <c:if test="${not empty errorOccurred}">
25         <p><span style="color: rgb(255, 0, 0);">Sie konnten nicht
26           authentifiziert werden.</span></p>
27       </c:if>
28       <p><input type="submit" value="Anmelden" name="buttonLogin"></p>
29     </form>
30   </body>
31 </html>
```

Listing 9.9: Die JSP-Seite `signIn.jsp`

Da bei falscher Eingabe eine Fehlermeldung angezeigt werden soll, wird mit einem `if`-Tag der JSTL geprüft, ob im (Request) Scope eine Fehlerinformation abgelegt wurde. Ist dies der Fall, dann wird zusätzlicher HTML-Code ausgegeben, der den Fehler in roter Schrift darstellt (Zeilen 24-27 in Codeausschnitt 9.9.

Beim Absenden des Formulars an den Server wird das (einzige) Servlet angesteuert. Die Ansteuerung wird mit dem JSTL-Tag `url` umgesetzt, das sich in der Wertangabe für das Attribut action des Formulars befindet. Bei der Übersetzung der JSP-Seite wird zuerst das `url`-Tag ausgewertet. Dieses liefert eine Zeichenkette, die als Wert an das Attribut action übergeben wird. Der Quelltext des Servlets ist in Codeausschnitt 9.10 angegeben (vgl. dazu auch Codeaus-

schnitt 8.5). Die Ausgabe der Erfolgsseite wird nun von der zweiten JSP-Seite `signedIn.jsp` übernommen.

```
1  package kurs01796.ke3.jspbeispiel;
2
3  import java.io.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6
7  @WebServlet(name="CheckSignInServlet", urlPatterns={"/CheckSignInServlet"})
8  public class CheckSignInServlet extends HttpServlet {
9     @Override
10     protected void doGet(HttpServletRequest request, HttpServletResponse
           response) throws ServletException, IOException {
11        //die Benutzereingaben aus dem Request holen
12        String username = (String) request.getParameter("username");
13        String password = (String) request.getParameter("password");
14        //pruefen, ob der korrekte Benutzername und das korrekte
15        //Passwort angegeben wurden
16        if ((username == null) || !username.equalsIgnoreCase("kurs1796") ||
              (password == null) || !password.equals("geheim")) {
17           //nicht erfolgreiche Authentifizierung fuehrt wieder
18           //zur Anmeldeseite zurueck
19           //vorher Information, dass ein Fehler aufgetreten ist,
20           //im Request Scope ablegen
21           request.setAttribute("errorOccurred", "true");
22           RequestDispatcher rd = request.getRequestDispatcher("/signIn.
                 jsp");
23           rd.forward(request, response);
24        } else {
25           //erfolgreiche Authentifizierng fuehrt zur "Erfolgsseite"
26           //Benutzernamen im Request Scope ablegen,
27           //damit die JSP-Seite ihn anzeigen kann
28           request.setAttribute("username", username)
29           RequestDispatcher rd = request.getRequestDispatcher("/WEB-INF/
                 pages/signedIn.jsp");
30           rd.forward(request, response);
31        }
32     }
33
34     @Override
35     protected void doPost(HttpServletRequest request, HttpServletResponse
           response) throws ServletException, IOException {
36        doGet(request, response);
37     }
38 }
```

Listing 9.10: Die Servlet-Klasse `CheckSignInServlet`

Zur Erzeugung der Erfolgsseite entnimmt die JSP-Seite `signedIn.jsp` den Benutzernamen mittels der Expression Language aus dem (Request) Scope:

```
<html>
  <head>
    <title>Beispiel Systemanmeldung - erfolgreich authentifiziert</title>
  </head>
  <body>
    <p><img src="images/LogoFeu.jpg" alt="FernUniversit&auml;t in Hagen"></
       p>
    <h1>Beispiel &quot;Systemanmeldung&quot;</h1>
    <h3>aus dem Kurs 01796, Kurseinheit 3</h3>
    <p>Sie wurden erfolgreich als Benutzer &quot; ${username} &quot;
       authentifiziert.</p>
  </body>
</html>
```

Listing 9.11: Die JSP-Seite signedIn.jsp

Die Welcome File List des Deployment Descriptors enthält lediglich die JSP-Seite `signIn.jsp`.

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

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
             http://java.sun.com/xml/ns/javaee/web-app_3_1.xsd"
         version="3_1">

  <welcome-file-list>
    <welcome-file>
      signIn.jsp
    </welcome-file>
  </welcome-file-list>

</web-app>
```

Listing 9.12: Der Deployment Descriptor web.xml

Abschließend noch die zugehörige Verzeichnisstruktur inklusive der Dateien:

```
SystemanmeldungJSP/signIn.jsp
SystemanmeldungJSP/images/LogoFeu.jpg
SystemanmeldungJSP/WEB-INF/web.xml
SystemanmeldungJSP/WEB-INF/pages/signedIn.jsp
SystemanmeldungJSP/WEB-INF/classes/CheckSignInServlet.class
```

Kopiert man diese Verzeichnisstruktur inklusive ihrer Dateien in das vom Web Container dafür vorgesehene Verzeichnis, dann kann die Web-Anwendung über einen Browser angesprochen werden. Für den Fall, dass die Web-Anwendung auf dem Host `localhost:8080` läuft, ist die zugehörige URL `http://localhost:8080/SystemanmeldungJSP`.

Diese Seite bleibt aus technischen Gründen frei!

10. Servlets und JavaServer Pages Technologie aus Software Engineering Sicht

In die Besprechung der Web-Komponenten Servlet und JSP-Seiten sind wiederholt Bemerkungen eingeflossen, die ihre aus Software Engineering Sicht sinnvolle Benutzung betreffen. Dies soll abschließend noch einmal aufgegriffen und vertieft werden.

Grundlage der Diskussion sind die Software Engineering Grundregeln *Trennung von Verantwortlichkeiten* (Separation of Concerns, SoC) sowie das *Kohäsionskriterium* (s. [SE I]). SoC besagt, dass die Aufgaben (Verantwortlichkeiten) der Softwarekomponenten (Module, Klassen usw.) so zugeschnitten sein sollten, dass keine Überlappungen entstehen. Das Kohäsionskriterium fordert, dass jede Softwarekomponente für nur genau eine fachlich/logisch zusammengehörende Aufgabe zuständig bzw. verantwortlich sein sollte. Auf der Basis von SoC und dem Kohäsionskriterium wollen wir nun die Verantwortlichkeiten von Servlet und JSP-Seite diskutieren. Wie wir sehen werden, muss dazu auch der Anwendungskern mit seiner Verantwortlichkeit in die Betrachtung einbezogen werden.

Rein technisch gesehen wäre es grundsätzlich möglich, für die Implementierung eines Requests entweder nur ein einziges Servlet oder nur eine einzige JSP-Seite einzusetzen. Man könnte sogar eine ganze Web-Anwendung erstellen, die nur ein einziges Servlet bzw. nur eine einzige JSP-Seite verwendet. Dass so etwas nicht sinnvoll ist, liegt auf der Hand, aber wie sollen die Aufgaben auf Servlet und JSP-Seite aufgeteilt werden, damit SoC und Kohäsionskriterium optimal erfüllt werden? Weiter hilft hier ein Blick auf das *Model-View-Controller Pattern*, kurz MVC-Pattern, das seit vielen Jahren als eine grundlegende Strukturierungsvorgabe für interaktive Systeme dient. Das *Model* dieses Strukturmusters ist für die Anwendungslogik und Anwendungsobjekte verantwortlich, der Controller steuert den Ablauf von Interaktionen und die *View* ist für die Präsentation der Ausgabe zuständig. Controller und *View* sind Teil der Benutzungsschnittstelle, das Model entspricht dem Anwendungskern.

Auch die Java EE-Spezifikation baut auf dem MVC-Pattern auf: Servlets sind dafür gedacht, die Aufgabe des Controller umzusetzen, JSP-Seiten sollen die View realisieren und EJBs zusammen mit Entities das Model implementieren.

10.1. Servlet

Dem MVC-Pattern folgend besteht die Verantwortlichkeit eines Servlets ausschließlich darin, die Abarbeitung von Requests zu steuern. Bei der Abarbeitung eines Requests übernimmt das Servlet zunächst vom Web Container die relevante Request-Information. Dann ruft es die Anwendungslogik auf, falls der Request Ergebnisse anfordert, die nur mit Kenntnis der Anwendungslogik geliefert werden können, und hinterlegt die Ergebnisdaten in einem Scope. Schließlich leitet das Servlet an die JSP-Seite weiter, die mit den Scope-Daten das Ausgabedokument zusammenstellt.

Ein Servlet sollte nicht die Ausgabe erstellen, denn das gehört zu der Verantwortlichkeit einer JSP-Seite. Wie bereits ausgeführt, stellt die Ausgabe in einem Servlet mittels `println(...)` nicht nur eine mühsame Angelegenheit dar, sondern vermischt auch die Codierungsformen HTML und Java – mit negativen Folgen für die Testbarkeit und Wartbarkeit. Hinzu kommt, dass die so codierten Web-Seiten erst zum Ausführungszeitpunkt endgültig beurteilt werden können. Ein Servlet sollte auch keine Anwendungsfunktionalität implementieren, sondern den Anwendungskern lediglich aufrufen. Die Implementierung der Anwendungsfunktionalität fällt allein in die Verantwortung des Anwendungskerns und wird nicht auf andere Komponenten verstreut. Dadurch lassen sich Testen und Ändern der Anwendungsfunktionalität so weit wie möglich auf den Anwendungskern beschränken.

10.2. JSP-Seite

Dem MVC-Pattern folgend besteht die Verantwortlichkeit einer JSP-Seite ausschließlich in der Präsentation der Ausgabe. Über HTML hinausgehende Techniken werden nur zur Darstellung von dynamisch erzeugtem Inhalt eingesetzt. Dazu muss die JSP-Seite im Grunde lediglich die vom Servlet in einem Scope abgelegten Daten, in erster Linie die Ergebnisse von Aufrufen des Anwendungskerns, auslesen können. Hierfür reichen die Expression Language oder die Java Standard Tag Library (evtl. unter Zuhilfenahme zusätzlicher, eigener benutzerdefinierter Tags) völlig aus.

Eine JSP-Seite sollte keine Aufgaben eines Servlets übernehmen, also nicht mittels Scripting-Elementen den Anwendungskern aufrufen oder Weiterleitungen an andere JSP-Seiten vornehmen. Auch bei diesen Aufrufen und Weiterleitungen wurden die Codierungsformen HTML und Java vermischt, sodass entweder die Entwickler das Web-Design mitübernehmen oder die Web-Designer über Java-Kenntnisse verfügen müssten. Ersteres ist in der Regel nicht empfehlenswert, Letzteres meist nicht gegeben. Darüber hinaus ergibt sich eine schlechtere Testbarkeit und Wartbarkeit. Bei Änderungen der Anwendungsfunktionalität müssten nicht nur die Aufrufe in Servlets, sondern zusätzlich noch in JSP-Seiten angepasst werden. Erst recht sollte eine JSP-Seite keine Anwendungsfunktionalität implementieren, denn dies wäre offensichtlich noch schlechter als Anwendungsfunktionen lediglich aufzurufen. Die Implementierung der Anwendungsfunktionalität ist allein Aufgabe des Anwendungskerns.

10.3. Anwendungskern

Dem MVC-Pattern folgend besteht die Verantwortlichkeit des Anwendungskerns ausschließlich in der Anwendungsfunktionalität, die mittels EJBs und Entities implementiert ist. Der Anwendungskern ist reiner Dienstleister für Servlets und reagiert nur auf deren Aufrufe.

10.4. Vorteile der strikten Umsetzung des MVC-Patterns

Welche Vorteile ergeben sich nun aus der beschriebenen Arbeitsteilung zwischen Servlet, JSP-Seite und Anwendungskern? Grundsätzlich fördert eine am MVC-Pattern orientierte disjunkte Aufteilung in kohäsive Verantwortlichkeiten die Komplexitätsbeherrschung. Zudem können Entwicklungstätigkeiten besser auf die Entwickler verteilt werden, indem z.B. die Erstellung von Servlets, JSP-Seiten, EJBs und Entities an die jeweiligen Spezialisten vergeben werden kann. Außerdem werden Test und Fehlersuche vereinfacht. Auch die Wartbarkeit wird verbessert, da Änderungen, z.B. der Darstellung der Ausgabe oder der Anwendungsfunktionalität, weit besser lokal gehalten, d.h. auf JSP-Seiten bzw. den Anwendungskern eingeschränkt werden können. Das MVC-Muster wird in Kurseinheit 4 noch einmal näher erläutert.

Abschließend sei noch darauf hingewiesen, dass bei einer Web-Anwendung mit vielen Web-Seiten die View-Komponente sehr viele JSP-Seiten enthält. Zur Darstellung einer Web-Seite wird nämlich mindestens eine JSP-Seite benötigt, oft sind es aber mehrere JSP-Seiten, die jeweils für einzelne Bereiche einer Web-Seite zuständig sind. Ähnlich sieht es mit der Controller-Komponente aus. Sie kann mehrere Servlets umfassen, die für unterschiedliche Aufgaben zuständig sind. So kann es ein Servlet geben, das alle Requests bearbeitet, die mit der Produktverwaltung zu tun haben, ein anderes ist nur für Requests verantwortlich, die die Kundenverwaltung betreffen usw. Ein anderer Ansatz, der auch vom Webframework JSF (siehe Kurseinheit 5) umgesetzt wird, arbeitet mit dem so genannten *Page Controller Pattern*, bei dem jeder Request von ein und demselben Servlet entgegengenommen wird. Die von dem jeweiligen Request geforderte individuelle Funktionalität wird durch weitere zum Controller gehörende Klassen realisiert, die vom Servlet entsprechend aufgerufen werden.

Die softwaretechnische Qualität einer Web-Anwendung wird aus all diesen Gründen wesentlich von der sauberen Aufteilung der Verantwortlichkeiten zwischen Servlet, JSP-Seite und Anwendungskern bestimmt.

Diese Seite bleibt aus technischen Gründen frei!

Kurseinheit 4

Softwarearchitekturmuster und Softwarearchitekturen für Webanwendungen

Das einführende Kapitel dieser Kurseinheit besteht aus einer Motivation, in der die Bedeutung der Softwarearchitektur für die Qualität eines Softwaresystems hervorgehoben wird, sowie einer Einführung in die Grundlagen von Softwarearchitekturen.

Im nachfolgenden Kapitel werden mit dem Client-Server-Architekturmuster, dem Schichtenarchitekturmuster und dem MVC-Architekturmuster drei Softwarearchitekturmuster vorgestellt, die im Kontext von Web-Anwendungen eine wichtige Rolle spielen. Ähnlich wie Entwurfsmuster beschreiben Architekturmuster bewährte Lösungsschemata. Softwarearchitekturen stellen dagegen konkrete Lösungen dar.

Das dritte und letzte Kapitel dieser Kurseinheit behandelt drei einschlägige Softwarearchitekturen, die Instanziierungen der vorgestellten Muster sind.

Diese Seite bleibt aus technischen Gründen frei!

11. Übersicht über Architekturmuster und Architekturen

Die eigentliche Realisierung eines Softwaresystems beginnt mit der Konzeption der *Softwarearchitektur*, auch kurz Architektur genannt, welche die zentralen Realisierungsentscheidungen und das grundsätzliche Strukturierungsschema des Softwaresystems festlegt. Eine Softwarearchitektur definiert die großen, strukturbildenden Bestandteile (Architekturkomponenten) und deren Zusammenspiel (ihre Benutzungsbeziehungen). Die Architekturkomponenten kapseln unterschiedliche Aufgabenbereiche und bündeln Softwareelemente, wie z.B. Klassen oder Schnittstellen, die selbst wieder in (Teil-)Komponenten zusammengefasst sein können. Jedes Softwareelement ist genau einer Architekturkomponente zugeordnet, d.h. das gesamte Softwaresystem wird in disjunkte[1] Architekturkomponenten aufgeteilt. Die Architektur stellt also formal eine Zerlegung des Softwaresystems dar, ihre Komponenten existieren aber meist, bevor das Softwaresystem geschrieben wird.

Architekturkomponenten beziehen sich auf das gesamte System. Daraus ergibt sich der globale Charakter einer Architektur. Hierin unterscheidet sie sich vom Softwareentwurf, dessen Realisierungsentscheidungen aus einer Verfeinerung der Architekturkomponenten bestehen; der Softwareentwurf ist somit lokaler Natur.

Ein bekannter Architekturansatz ist die *3-Schichten-Architektur* für Desktop-Anwendungsprogramme, bei der sich die Architekturkomponenten in Form übereinanderliegender Schichten darstellen lassen (vgl. [SE I]). Die obere Schicht realisiert die Benutzungsschnittstelle, die mittlere den Anwendungskern und die untere die (persistente) Datenhaltung. Die Idee dabei ist einfach: Jede Schicht darf nur tiefere Schichten benutzen.

Bei dem Entwurf einer Architektur sollen neben den funktionalen insbesondere auch die nichtfunktionalen Anforderungen an das Softwaresystem angemessen berücksichtigt werden. Zu den klassischen nicht-funktionalen Anforderungen, die eine Architektur zu unterstützen hat, gehören aus Benutzungssicht in erster Linie Performanz, Sicherheit, Verfügbarkeit, Zuverlässigkeit und Robustheit. Aus softwaretechnischer Sicht sind vor allem Testbarkeit und Integrierbarkeit, Wartbarkeit, Änderbarkeit, Portierbarkeit, Skalierbarkeit und Wiederverwendbarkeit wichtig. Dieser (nicht vollständige) Katalog von nicht-funktionalen Anforderungen verdeutlicht den entscheidenden Einfluss der Architektur auf die Gesamtqualität des Softwaresystems.

Da für ein konkretes Softwaresystem keine Architektur alle Anforderungen gleich gut erfüllen kann, werden in der Praxis die nicht-funktionalen Anforderungen meist mit einer Priorisierung

[1] Mengenlehre: Zwei Mengen heißen disjunkt, wenn sie kein gemeinsames Element besitzen

versehenen. Eine Architektur stellt daher immer einen Kompromiss hinsichtlich verschiedener Kriterien dar. Als *gute Architektur* für ein geplantes Softwaresystem bezeichnen wir eine Architektur, die (mindestens) die folgenden Kriterien weitgehend erfüllt:

- Sie bietet eine adäquate Basis zur Realisierung der funktionalen und nicht-funktionalen Anforderungen.
- Sie ist im Rahmen des Möglichen unabhängig von den Spezifika des geplanten Softwaresystems, d.h. sie ist so allgemein wie möglich und so speziell wie nötig.
- Die Architekturkomponenten besitzen jeweils eine hohe Kohäsion und befolgen die Grundregel der Trennung von Verantwortlichkeiten (Separation of Concerns), d.h. besitzen klar voneinander abgegrenzte Aufgabenbereiche.
- Die Architekturkomponenten sind so gewählt, dass ihre Kopplung möglichst gering ausfällt und ihr Zusammenspiel möglichst einfach und überschaubar ist. So sollten die Benutzungsbeziehungen zwischen ihnen möglichst keine Zyklen enthalten.

Das erste Kriterium leuchtet unmittelbar ein. Das zweite Kriterium zielt auf die Wiederverwendbarkeit ab; insbesondere sollte die Architektur so ausgelegt sein, dass sie eine Software-Produktfamilie tragen kann. Die Kriterien 3 und 4 verbessern die Komplexitätsbeherrschung und insbesondere die Wartbarkeit und Änderbarkeit. Außerdem ermöglichen sie eine weitgehend unabhängige Entwicklung der Architekturkomponenten durch jeweilige Spezialisten. Zum Beispiel können bei Verwendung der 3-Schichten-Architektur Anwendungsprogrammierer den Anwendungskern entwickeln, während die Datenbank- bzw. GUI-Spezialisten die beiden anderen Schichten realisieren. Eine weitgehende Zyklenfreiheit erleichtert das Testen, insbesondere den Integrationstest.

12. Softwarearchitekturmuster

Softwarearchitekturmuster haben als Vorlagen (Blaupausen) für Softwarearchitekturen eine große Bedeutung. Ähnlich wie Entwurfsmuster beschreiben Architekturmuster bewährte Lösungsschemata. Aus einem Architekturmuster entsteht eine konkrete Architektur durch entsprechende Instanziierung. Umgekehrt bedeutet das aber nicht, dass einer Architektur stets ein Architekturmuster zugrunde liegen muss.

Je nach Einsatzbereich und den jeweils relevanten funktionalen und nicht-funktionalen Anforderungen haben sich verschiedene Architekturmuster als günstig erwiesen. Für den Bereich der Web-Anwendungen behandelt dieses Kapitel drei der wichtigsten Architekturmuster.

12.1. Client-Server-Architekturmuster

Das *Client-Server-Architekturmuster* ist ein einfaches, grundlegendes Muster, bei dem das Softwaresystem in zwei Architekturkomponenten zerlegt wird, eben in den Client und den Server. Diese Zerlegung sagt zunächst nichts darüber aus, ob die Architekturkomponenten auf einem gemeinsamen oder auf verschiedenen Rechnern realisiert werden. Die Aufgabenbereiche von Client und Server sind klar voneinander abgegrenzt: Der Server bietet Dienste an, die der Client nutzt. Hierdurch ergibt sich auch eine einfache Benutzungsstruktur zwischen den beiden Komponenten. Die umgekehrte Benutzung ist nicht möglich, der Server kennt den Client nicht einmal. Offensichtlich lassen sich aus dem Client-Server-Architekturmuster gute Architekturen gewinnen, sofern beide Komponenten kohäsive und klar voneinander abgegrenzte Aufgabenbereiche besitzen.

Abb. 12.1 zeigt das Client-Server-Architekturmuster. Der gestrichelte Pfeil verdeutlicht die Benutzungsbeziehung zwischen den beiden Komponenten.

12.2. Schichtenarchitekturmuster

Auch das *Schichtenarchitekturmuster* ist ein verbreitetes und bewährtes Muster. Man kann es als eine Verfeinerung des Client-Server-Architekturmusters betrachten, die den Server in rekursiver Weise wieder gemäß dem Client-Server-Architekturmuster strukturiert. Daraus ergibt sich, dass jede Architekturkomponente bis auf die „unterste“, die sich auf keiner anderen Architekturkomponente abstützt, nur genau eine andere Architekturkomponente benutzen darf. Schichtet

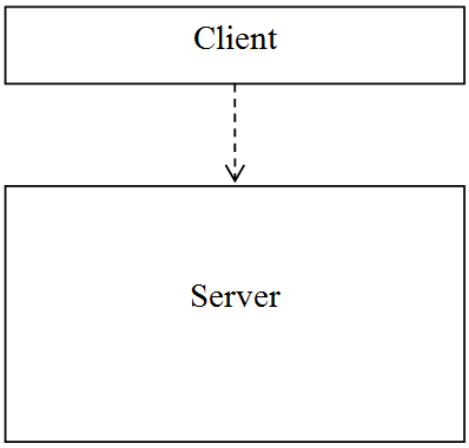

Abbildung 12.1.: Client-Server-Architekturmuster

man die Architekturkomponenten derart übereinander, dass (bis auf die unterste) jede von ihnen unmittelbar oberhalb der von ihr benutzten Architekturkomponente eingereiht wird, kann auf die Angabe der zulässigen Benutzung verzichtet werden, da diese sich aus der Anordnung der Schichten ergibt. Diese gängige und übersichtliche Darstellung der Architekturkomponenten in Form von Schichten hat zu der Bezeichnung Schichtenarchitektur(-muster) geführt. Historisch gesehen war die erste Ausprägung des Schichtenarchitekturmusters die 3-Schichten-Architektur, die zu den klassischen Architekturen für Anwendungen in kommerziellen Domänen gehört.

Bisher haben wir das strikte Schichtenarchitekturmuster erläutert, bei dem Benutzungen zwischen Schichten nur von einer Schicht zur nächsttieferen Schicht zulässig sind. Bei dem allgemeinen Schichtenarchitekturmuster hingegen darf eine Schicht auch auf andere, tiefere Schichten zugreifen. So kann unnötiges Durchschleifen von Operationsaufrufen und Objekten über Schichten hinweg vermieden werden. Derartige Zugriffe über Schichten hinweg sollten jedoch genau überlegt und gut begründet sein.

Abb. 12.2 stellt das allgemeine Schichtenarchitekturmuster mit einigen exemplarischen Benutzungsbeziehungen dar (vgl. auch [SE I]).

Wenn wir im Folgenden die Adjektive „strikt“ bzw. „allgemein“ weglassen und nur von Schichtenarchitektur(-muster) sprechen, ist stets die allgemeine Variante gemeint, die die strikte Variante mit einschließt.

Die Benutzungsbeziehungen zwischen Schichten sind schon aufgrund der Vorgaben des Schichtenarchitekturmusters zyklenfrei. Als weiterer Vorteil ergibt sich eine geringe Kopplung zwischen den Schichten. Eine Schicht benutzt nur im Ausnahmefall mehr als eine andere Schicht. Damit ist sie nur von wenigen anderen Schichten abhängig, was die Testbarkeit, Wartbarkeit und insbesondere die Änderbarkeit erhöht. Bei der strikten Variante ist die Kopplung zwischen den Schichten sogar minimal (was aber nicht notwendig zwischen Klassen verschiedener Schichten gelten muss!).

Offensichtlich lassen sich aus dem Schichtenarchitekturmuster gute Architekturen gewinnen,

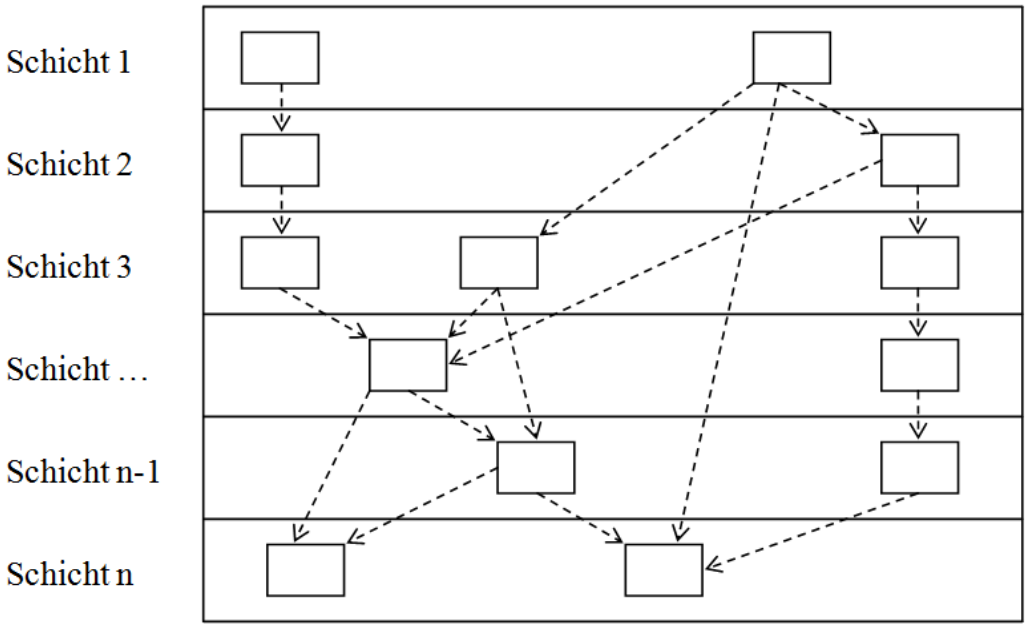

Abbildung 12.2.: Allgemeines Schichtenarchitekturmuster für n Schichten mit einigen exemplarischen Benutzungsbeziehungen

sofern deren Komponenten kohäsive und klar voneinander abgegrenzte Aufgabenbereiche sowie eine geringe Kopplung besitzen.

12.3. MVC-Architekturmuster

Das MVC-Architekturmuster wird häufig für interaktive Anwendungsprogramme verwendet. Es wurde bereits in den 1980er Jahren im Rahmen der Programmiersprache Smalltalk-80 eingeführt. Das MVC-Muster, von dem verschiedene Varianten (mit kleinen Unterschieden) kursieren, wird als Entwurfsmuster und auch als Architekturmuster eingesetzt. Wir erläutern eine gängige Variante am Beispiel von GUI-Anwendungen. Das MVC-Muster besteht aus den drei Komponenten Model (M), View (V) und Controller (C). Das Model entspricht dem Anwendungskern, die View ist nur für die Realisierung der Benutzungsoberfläche zuständig. Der Controller analysiert Benutzerereignisse, die eine Benutzerin z.B. durch Anklicken einer Schaltfläche ausgelöst hat, und steuert jeweils abhängig vom Analyseergebnis den zugehörigen Ablauf unter Rückgriff auf Model und View.

Abb. 12.3 stellt den von einem Benutzerereignis angestoßenen Ablauf im MVC-Architekturmuster dar.

Abhängig vom Benutzerereignis ruft der Controller das Model zur Umsetzung des Benutzerwunsches auf (1). Anschließend ruft er die View auf (2), damit sie das vom Model ermittelte

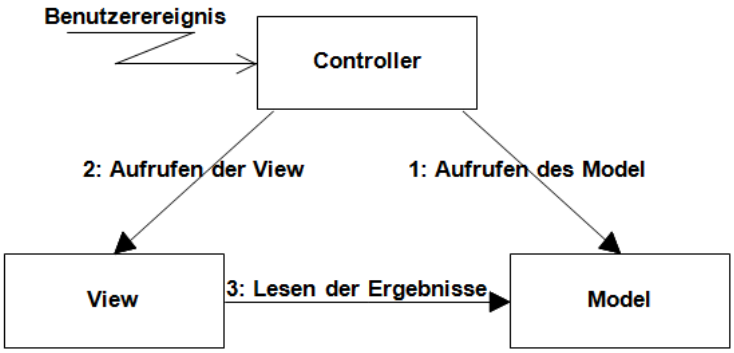

Abb. 12.3 Ablauf im MVC-Architekturmuster

Abbildung 12.3.: Ablauf im MVC-Architekturmuster

Ergebnis auf dem Bildschirm darstellt (3).

Im Gegensatz zu den beiden vorangegangenen Architekturmustern stehen beim MVC-Muster die Aufgabenbereiche der Komponenten grundsätzlich fest und werden nicht erst bei der Instanziierung einer konkreten Architektur bestimmt. Insofern lassen sich hier bereits Aussagen über die Qualität der Muster-Komponenten machen: Ihre Aufgabenbereiche sind kohäsiv und klar voneinander getrennt. Zudem sind die Benutzungsbeziehungen einfach und enthalten keine Zyklen. Jede Instanz-Architektur des MVC-Musters hat damit schon wesentliche positive Eigenschaften.

13. Softwarearchitekturen

Im Unterschied zu Softwarearchitekturmustern, die Schemata für Softwarearchitekturen anbieten, stellen letztere konkrete Lösungen dar. Dieses Kapitel behandelt drei im Kontext von Web-Anwendungen bekannte Softwarearchitekturen, die Instanziierungen der in Kapitel 12 behandelten Architekturmuster sind. Je nach Architektur wird der Browser mal als zusätzliche Komponente mit in die Darstellung der Architektur aufgenommen, wie z.B. bei der 5-Schichten-Architektur (s. Abschnitt 13.1), und mal nicht, wie z.B. bei der Model-1-Architektur (s. Abschnitt 13.2) und Model-2-Architektur (s. Abschnitt 13.3). Die Architekturüberlegungen beziehen sich natürlich stets nur auf die eigentliche Web-Anwendung, d.h. auf das zu entwickelnde Softwaresystem und nicht auf den Browser.

13.1. 5-Schichten-Architektur

Durch Anpassung an die Bedürfnisse von Web-Anwendungen hat sich als konkrete Ausprägung des Schichtenarchitekturmusters eine Schichten-Architektur etabliert. Dabei umfasst die eigentliche Web-Anwendung vier Schichten, zusammen mit dem Browser als oberster Schicht erhält man dann eine *5-Schichten-Architektur*. Abbildung 13.1 zeigt diese Architektur. Die Bezeichnungen für die Schichten vaiieren in der Literatur.

Die unterste Schicht repräsentiert die (persistente) *Datenhaltung*. Sie besteht in den meisten Fällen aus einem relationalen Datenbanksystem, das die persistente Speicherung der Anwendungsobjekte übernimmt.

Die darüber liegende Schicht enthält die *Anwendungsobjekte*, die oft, so auch in [SE I], als Entitäten bezeichnet werden. Diese Schicht repräsentiert somit das auf Entitätsklassen und ihre Beziehungen reduzierte Klassenmodell der Web-Anwendung.

Die nächste Schicht beherbergt mit der *Anwendungslogik* die fachliche Funktionalität der Web-Anwendung, d.h im Wesentlichen die Anwendungslogik sämtlicher Anwendungsfälle. In [SE I] wird die Anwendungslogik von Kontrollklassen realisiert. Den Begriff Kontrollklasse für Klassen der Anwendungslogik werden wir wegen der Verwechslungsgefahr mit dem Begriff Controller des MVC-Musters (vgl. Abschnitt 12.3) in diesem Kurs nicht verwenden.

Die Schicht darüber ist die *Web-Schicht*. Sie nimmt jeden Request entgegen, sorgt durch Aufrufen der Anwendungslogik oder ggf. der Anwendungsobjekte für die Umsetzung der jeweiligen Anfrage und liefert die Antwort an den Client zurück.

Die oberste Schicht bildet der *Browser*.

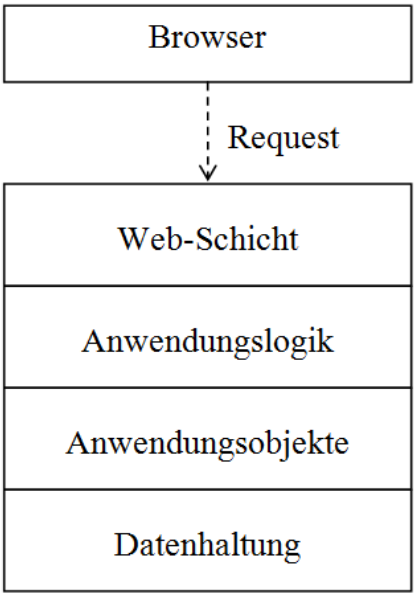

Abbildung 13.1.: 5-Schichten-Architektur für Web-Anwendungen

Wir präzisieren jetzt die Regeln für die schichtenübergreifende Benutzung der 5-Schichten-Architektur.

Ein Request des Browsers wird ausschließlich von der Web-Schicht entgegengenommen. Die WebSchicht benutzt die Anwendungslogik, kann aber in Ausnahmefällen, z.B. für reine Präsentationszwecke, auch auf die Anwendungsobjekte zugreifen. Die Anwendungslogik benutzt nur die Anwendungsobjekte und die Datenhaltung. Die Anwendungsobjekte greifen nicht auf die Datenhaltung, also auch auf keine weitere Schicht, zu. Die Datenhaltung wird also nur von der Anwendungslogik angesteuert. Abbildung 13.2 zeigt die Struktur der erlaubten Benutzungen in der 5-Schichten-Architektur.

Als Instanziierung des Schichtenarchitekturmusters ist die vorgestellte 5-Schichten-Architektur potentiell eine gute Architektur, da ihre Schichten – bis auf die Web-Schicht – kohäsive und klar voneinander abgegrenzte Aufgabenbereiche besitzen. Die Kohäsion der Web-Schicht, die sowohl die Steuerung der Anfrageverarbeitung als auch die Präsentation der Ergebnisse übernimmt, ist allerdings verbesserungswürdig. Spätestens mit der Verwendung des MVC-Musters sind die Vorteile der Trennung dieser beiden Aufgabenbereiche bekannt. Im Rahmen der Schichtenarchitektur könnte die Trennung durch zwei Web-Schicht-lokale Komponenten gemäß View und Controller ebenfalls erreicht werden.

In der Praxis, so auch bei Java EE, kommt zwischen den Schichten der Anwendungslogik und der Anwendungsobjekte gewöhnlich eine weitere Schicht für das *Persistenz-Framework* hinzu. Ein Persistenz-Framework kapselt den Zugriff auf ein Datenbanksystem, bildet Klassen auf Datenbanktabellen ab und ermöglicht so das Speichern und Laden von Objekten. Das Persistenz-

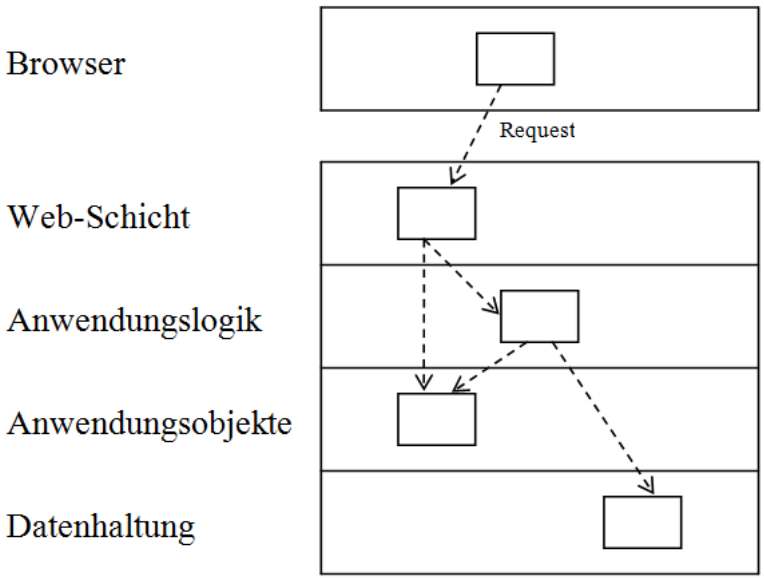

Abbildung 13.2.: Die zulässigen Benutzungen in der 5-Schichten-Architektur für Web-Anwendungen

Framework verarbeitet keine echte Anwendungslogik, sondern verbirgt lediglich die nichttriviale Technik der Kommunikation zwischen objektorientierter Software und relationalem Datenbanksystem. Für unsere methodischen Überlegungen schließen wir das Persistenz-Framework aus unseren Betrachtungen aus.

13.2. Model-1-Architektur

Die Einführung der JSP-Technologie hat das Entstehen der *Model-1-Architektur* begünstigt. Diese Architektur hatte zunächst eine gewisse Popularität erreicht, wurde dann aber wegen ihrer offensichtlichen Schwächen bald von der Model-2-Architektur (s. Abschnitt 13.3) weitgehend verdrängt.

Die Model-1-Architektur ging aus einer auf Web-Anwendungen zugeschnittenen Instanziierung des Client-Server-Architekturmusters hervor. Der Server wird hier Model genannt, weil er dem Model des MVC-Musters entspricht. Der Client wird als Web-Komponente bezeichnet, weil er einen Aufgabenbereich realisiert, der dem der Web-Schicht der 5-Schichtenarchitektur gleicht. Die Web-Komponente besteht im Wesentlichen nur aus JSP-Seiten, die zu ihrer eigentlichen Aufgabe – der Darstellung der Response – die Aufgaben von Servlets zusätzlich übernehmen. Dazu dienen Scripting-Elemente, über die Java-Code in eine JSP-Seite eingebettet werden kann. Im Unterschied zur 5-Schichten-Architektur werden bei der Model-1-Architektur technische Vorgaben für die Realisierung von Komponenten (hier der Web-Komponente) gemacht.

Abbildung 13.3 zeigt den Ablauf eines Standard-Requests in der Model-1-Architektur. Zum

besseren Verständnis ist der Browser ebenfalls aufgeführt. Zunächst schickt der Browser den Request an die in der URL angegebene JSP-Seite (1). Diese JSP-Seite führt ggf. verschiedene (Eingabe-)Prüfungen in Abhängigkeit von den übergebenen Parametern aus und ruft danach das Model zwecks Lieferung der gewünschten Informationen auf (2). Abschließend stellt die JSP-Seite aus den gelieferten Informationen und ihrem statischen Inhalt eine HTML-Seite zusammen und übergibt diese dem Browser als Response (diese Zusammenstellung übernimmt genau genommen das Servlet, in das die JSP-Seite übersetzt wird).

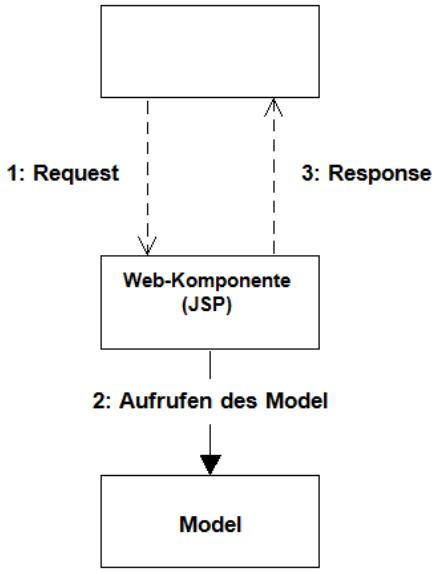

Abbildung 13.3.: Ablauf eines Standard-Request in der Model-1-Architektur

Die Aufgabenbereiche der Architekturkomponenten sind klar voneinander getrennt. Die Kohäsion der Web-Komponente ist, wie schon bei der Web-Schicht der 5-Schichten-Architektur, suboptimal. Erschwerend kommt hinzu, dass die Web-Komponente allein über JSP-Seiten realisiert wird. Je Anfrage werden nämlich sowohl die Steuerung der Anfrageverarbeitung als auch die Präsentation von einer einzigen JSP-Seite übernommen. Die JSP-Seiten besitzen damit nicht nur eine schlechte Kohäsion, sondern vermischen auch HTML- und Java-Code. Die Model-1-Architektur ist daher keine gute Architektur. Wir empfehlen sie noch nicht einmal für Kleinprojekte, bei denen die Programmiererin zugleich für die Gestaltung der Web-Seiten zuständig ist.

13.3. Model-2-Architektur

Die Nachteile der Model-1-Architektur wurden rasch offensichtlich. Als Instanziierung des MVC-Architekturmusters entstand daraufhin die *Model-2-Architektur*. Bei der Model-2-Architektur werden die Aufgaben der View von JSP-Seiten und die des Controllers von Servlets (abgesehen von möglichen Hilfsklassen) übernommen. Das Model entspricht wie üblich dem

Anwendungskern. Wie bei der Model-1-Architektur werden auch bei der Model-2-Architektur technische Vorgaben für die Realisierung von Komponenten (hier für View und Controller) gemacht.

Abbildung 13.4 zeigt den Ablauf eines Standard-Requests in der Model-2-Architektur.

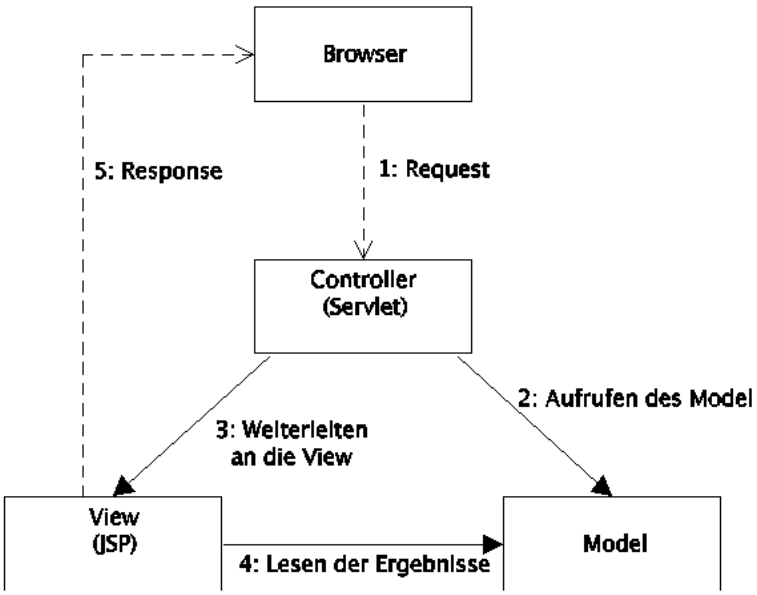

Abbildung 13.4.: Ablauf eines Standard-Request in der Model-2-Architektur

Zunächst trifft der Request beim Controller ein (1). Dieser führt ggf. (Eingabe-)Prüfungen für die übergebenen Parameter aus und ruft danach das Model auf, um die in der Anfrage gewünschten Informationen zu erhalten (2). Der Controller nimmt die Ergebnisse vom Model entgegen und leitet anschließend an die zugehörige JSP-Seite weiter (3), damit diese die abschließende HTML-Seite zusammenstellt. Die JSP-Seite liest die vom Controller zur Verfügung gestellten (Model-)Ergebnisse (4) und liefert zusammen mit ihrem statischen Inhalt die fertige HTML-Seite als Response an den Browser zurück (5).

Beispiel

Für das Beispiel einer einfachen Systemanmeldung zeigt Abbildung 13.5, wie die Architekturkomponenten Model, View und Controller durchlaufen werden. In diesem einfachen Beispiel gibt es keinen Zugriff der View auf einen Scope. Dieser Request-Ablauf ähnelt dem in Kurseinheit 3 behandelten Standard-Request-Ablauf. Im Grunde besteht der Unterschied nur darin, dass in dem Standard-Request-Ablauf auch ein Zugriff der JSP-Seite auf den Request-Scope vorkommt.

Mit dem Drücken der Schaltfläche „anmelden“ löst die Benutzerin einen Request aus, der die eingegebenen Daten für Benutzername und Passwort enthält. Der Request wird vom Web-Container entgegengenommen und dem Controller (Servlet) übergeben, dessen `doPost`-Methode die eigentliche Abarbeitung der Anfrage ausführt (1). Diese Methode ruft das Model auf, damit es die eingegebenen Benutzerdaten überprüft (2). Verläuft die Überprüfung erfolgreich, wird an die

JSP-Seite `erfolgreich.jsp` weitergeleitet (3a), die dann der Benutzerin die nächste Webseite präsentiert. Verläuft die Überprüfung nicht erfolgreich, wird die JSP-Seite `anmelden.jsp` angestoßen (3b), die der Benutzerin für einen weiteren Anmeldeversuch wieder die Eingangs-Webseite präsentiert.

Mit den vom MVC-Muster übernommenen Verantwortlichkeiten und Benutzungsbeziehungen der Architekturkomponenten stellt die Model-2-Architektur eine gute Architektur dar. Leider werden die Vorgaben der Architektur in der Praxis oft zu inkonsequent befolgt, hauptsächlich weil die Entwickler sich nicht an die strenge Arbeitsteilung zwischen Servlets und JSP-Seiten halten. So wird eine JSP-Seite oft nicht ausschließlich für die Präsentation der (Model-)Ergebnisse eingesetzt, d.h. es gibt nicht nur lesende Zugriffe auf das Model, sondern es wird auf das Model auch berechnend oder manipulierend zugegriffen, z.B. mit Hilfe von Scripting-Elementen.

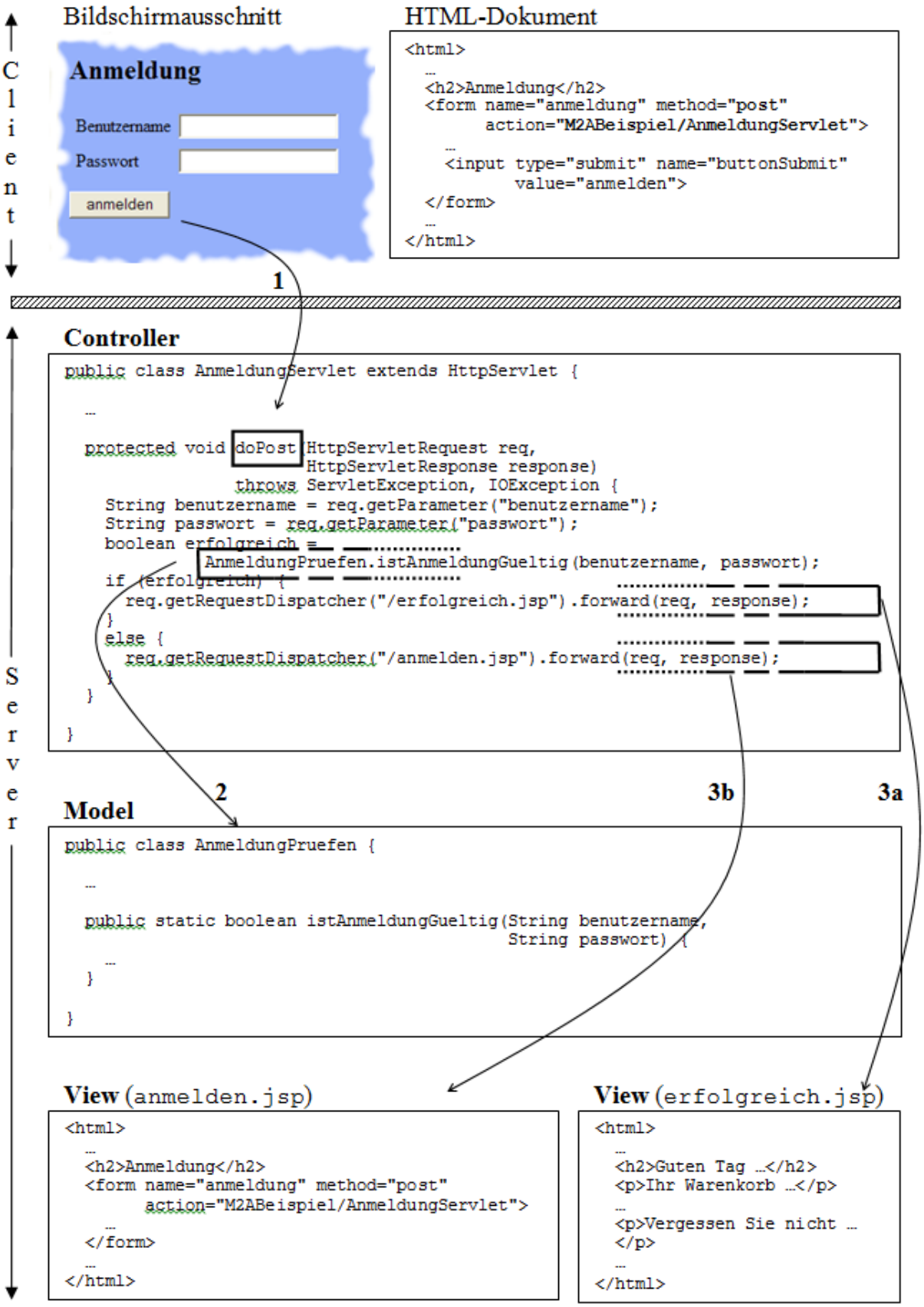

Abbildung 13.5.: Ablauf einer Systemanmeldung in der Model-2-Architektur

Diese Seite bleibt aus technischen Gründen frei!

Kurseinheit 5
Web-Framework JavaServer Faces

Die Implementierung der Web-Schicht komplexer Java EE Web-Anwendungsprogramme ist eine technisch anspruchsvolle Aufgabe. Es verwundert daher nicht, dass inzwischen etliche Frameworks angeboten werden, die den Entwicklern viel Programmierarbeit abnehmen.

Als Beispiel für ein derartiges Web-Framework wird in dieser Kurseinheit JavaServer Faces (JSF) vorgestellt, das von mehreren Technologieführern, z.B. Sun Microsystems, Oracle und IBM entwickelt wurde. JSF basiert auf dem MVC Muster und bietet eine vereinfachte Entwicklung von Benutzungsoberflächen für Java-Web-Anwendungen.

Diese Seite bleibt aus technischen Gründen frei!

14. Einstieg in JSF

Die Servlet-Technologie dient der Verarbeitung von Client-Anfragen und der Erzeugung entsprechender Antworten innerhalb eines Java EE Anwendungsprogramms. Die direkte Generierung von Ausgabedokumenten durch Servlets hat negative Auswirkungen auf die Testbarkeit und Wartbarkeit des Anwendungsprogammes, eine Trennung der Sicht (HTML) und der Anwendungslogik (Java) sollte stets beibehalten werden. Aufbauend auf der Servlet-Technologie bietet die JSP-Technologie eine Methode zur Integration von Java-Programmcode in HTML-Code. Die in JSP zur Verfügung stehenden Elemente ermöglichen im Gegensatz zu Servlets eine weitaus konsequentere Trennung der Anwendungsschichten. Die Integration von Java-Code kann allerdings auch missbraucht werden.

Mit *JavaServer Faces* (JSF) stellt diese Kurseinheit ein Framework für die Entwicklung von Benutzungsschnittstellen in Java-Web-Anwendungen vor, das eine strikte Trennung von Java- und HTML-Code vorsieht. Unterstützt wird diese Trennung durch die Model-2-Architektur[1], die als Grundlage für JSF dient.

Ist eine Web-Anwendung mit der JSF-Technologie entwickelt worden, übernimmt ein zentrales Servlet, das sogenannte *Faces Servlet* , die Funktion des Controllers innerhalb des MVC-Musters. Das Model wird meist durch eine oder mehrere JavaBeans umgesetzt. Um die View zu definieren, wurden in der vorherigen Kurseinheit JSP-Seiten bzw. JSP als Präsentationstechnologie verwendet. Die im Rahmen einer solchen Technologie verwendete Syntax wird auch *Seitendeklarationssprache* genannt. Implementierungen der JavaServer Faces-Technologie unterstützen zwei Seitendeklarationssprachen. Vor der JSF-Version 2.0 wurden JavaServer Pages für die Definition der View genutzt, ab Version 2.0 hat *Facelets* den Platz von JSP als Seitendeklarationssprache eingenommen. JSP wird im Rahmen von JSF nur noch aus Kompatibilitätsgründen unterstützt.

JavaServer Faces ist Teil des Java EE Standards und hat sich mittlerweile zum Standard-Framework zur Entwicklung von grafischen Benutzungsoberflächen etabliert.

In diesem Kurs wird die JSF-Spezifikation 2.2 und die entsprechende Referenzimplementierung von JavaServer Faces (*Mojarra*), welche Teil des *Glassfish-Anwendungs-Servers* ist, verwendet.

[1] Die Model-2-Architektur ist eine Spezialisierung der MVC-Architektur

14.1. Zentrale Bestandteile von JSF

Für die Entwicklung von JSF-Anwendungsprogrammen werden im Wesentlichen die zentralen Begriffe FacesServlet, Facelets, Komponente, Expression Language, Managed Bean, Validierung, Konvertierung, Ereignis und Navigation benötigt. Dieser Abschnitt enthält einen Überblick über diese Begriffe.

14.1.1. FacesServlet

Das *FacesServlet* ist ein Servlet (das Front Controller Servlet), das vom JSF-Framework bereitgestellt wird. Es entspricht einem zentralen Controller innerhalb der Model-2-Architektur. Alle Requests an eine JSF-Anwendung werden von diesem Servlet verarbeitet. Um dies zu gewährleisten, muss ein entsprechendes Servlet Mapping[2] im Deployment Descriptor `web.xml` vorgenommen werden. Das FacesServlet nimmt Requests (gemäß des Mapping) entgegen, kapselt alle für die weitere Anfrageverarbeitung und Antworterzeugung relevanten Informationen in einem sogenannten FacesContext-Objekt und übergibt die Kontrolle für die weitere Anfrageverarbeitung an ein Lifecycle-Objekt weiter. Bei jedem Request an ein JSF-Anwendungsprogramm wird eine Instanz des FacesContext-Objektes erzeugt. Dieses enthält alle Informationen über den Request und steht während der gesamten Phase der Anfrageverarbeitung (Lebenszyklus des Requests) zur Verfügung. Weiterhin wird zu jedem Request eine Instanz der Klasse Lifecycle erzeugt. Diese ist dafür zuständig den Lebenszyklus mit seinen Phasen zu kontrollieren (siehe Kapitel 15). Außer dem FacesServlet wird in JSF-Anwendungsprogrammen in der Regel kein weiteres Servlet benötigt.

14.1.2. Facelets

Facelets bezeichnet eine Sprache, mit der die Ansicht (View) bzw. Darstellung von JSF-Seiten in JSF-Anwendungsprogrammen definiert werden kann, also eine Seitendeklarationssprache (View Declaration Language - VDL). Die JSF-Technologie unterstützt neben Facelets als Seitendeklarationssprache auch JavaServer Pages. Die Vorteile des Einsatzes von Facelets werden in Kapitel 16 erläutert.

14.1.3. Komponente

Unter einer *Komponente* versteht man ein wiederverwendbares Element in einer JSF-Seite. Eine JSF-Seite ist üblicherweise durch eine Vielzahl von Komponenten definiert. Eine Komponente wird über ein entsprechendes öffnendes und ein schließendes Tag in eine JSF-Seite integriert und ist Teil einer Tag-Bibliothek, die zur Nutzung der Komponente in die JSF-Seite eingebunden werden muss. Eine Komponente kann z.B. ein einfaches Texteingabefeld sein, aber auch ein

[2] Siehe Abschnitt 7.3.1 in Kurseinheit 3

Kalender, eine Datentabelle oder eine Zusammensetzung aus mehreren Komponenten. In Kapitel 16 werden wichtige Komponenten, die zur Entwicklung eines JSF-Anwendungsprogramms genutzt werden können, vorgestellt.

14.1.4. Expression Language

Durch die *Expression Language* lassen sich Komponenten einer JSF-Seite mit Server-seitigen Objekten verbinden. In Kurseinheit 3 wurde bereits gezeigt, wie aus einer JSP-Seite heraus auf Attribute von JavaBeans zugegriffen werden kann. Auch in JSF wird die Expression Language dazu verwendet auf JavaBeans, sogenannte *Managed Beans* , zuzugreifen, Properties auszulesen, zu schreiben und Methoden einer Managed Bean aufzurufen. Anders als in JSP-Seiten, in denen auch per Java-Code auf Server-seitige Objekte zugegriffen werden kann, wird in JSF-Anwendungsprogrammen die Expression Language eingesetzt, um Beziehungen zwischen Komponenten der Präsentationsschicht und Server-seitigen Objekten zu erstellen.

14.1.5. Managed Beans

Managed Beans nehmen in einem JSF-Anwendungsprogramm eine wichtige Rolle ein. Eine Managed Bean ist eine JavaBean, die als Schnittstelle zwischen der Präsentationsschicht und der Anwendungslogik fungiert. Aus einer JSF-Seite heraus kann auf die Properties einer oder mehrerer Managed Beans zugegriffen oder Methoden aufgerufen werden. Eine der zentralen Aufgaben einer Managed Bean ist die „Aufbewahrung“ von Daten, die aus einer JSF-Seite stammen (z.B. Eingaben einer Benutzerin) und der Aufruf von Methoden des Anwendungskerns. Auch können Managed Beans Daten für Komponenten einer JSF-Seite zur Verfügung stellen. Kapitel 17 zeigt die Verwendung von Managed Beans.

14.1.6. Validierung

Gerade bei Anwendungsprogrammen mit viel Benutzerinteraktion und einer großen Anzahl einzugebener Daten ist die Wahrscheinlichkeit von Fehleingaben oder ausgelassenen Daten hoch. Damit ein Programm zur Laufzeit keine falschen Berechnungen durchführt, unsinnige Daten speichert oder gar komplett abstürzt, sollten stets Validierungen eingegebener Daten durchgeführt werden. Durch eine Validierung wird z.B. geprüft, ob ein eingegebener Wert innerhalb eines vorgegebenen Intervalls liegt oder eine Zeichenkette eine maximale Länge nicht überschreitet. Ein Validator, der eine Validierung eines Datums durchführt, kann z.B. ein Element einer JSF-Seite sein, das mit einer Komponente derselben JSF-Seite (z.B. mit einem Texteingabefeld) verknüpft ist. JSF stellt eine Menge von Standard-Validatoren zur Verfügung.

14.1.7. Konvertierung

Validatoren übernehmen die Validierung von Daten, die von Benutzern eingegeben werden. Doch in welcher Form liegen diese Daten vor? In einer JSF-Anwendung werden auf dem Web-Client alle Daten als Zeichenketten verarbeitet, Server-seitig jedoch als Java-Typen. Bei der Kommunikation zwischen Client und Server müssen die Daten somit in einen geeigneten Typ konvertiert werden. Ein *Konverter* ist für derartige Konvertierungen zuständig und kann – wie ein Validator – durch ein entsprechendes Tag einer Komponente einer JSF-Seite zugewiesen werden. Er kann z.B. die Eingabe in einem Texteingabefeld (die als Zeichenkette vorliegt) in eine Zahl oder einen Wahrheitswert umwandeln. JSF stellt eine Menge von Standard-Konvertern zur Verfügung.

14.1.8. Ereignis

Ein zentraler Bestandteil jeder Web-Anwendung ist die Behandlung von Aktionen der Benutzer. Dies kann z.B. die Eingabe eines Wertes in ein Texteingabefeld sein oder die Betätigung einer Schaltfläche. In JavaServer Faces werden für die Verarbeitung sogenannte *Ereignisse* (Events) verwendet. Ein Ereignis beschreibt die Aktion und wird, sobald es eintritt, von einem sogenannten Event Listener abgefangen. Der Event Listener ruft daraufhin eine Ereignisbehandlungsmethode auf. Dafür muss ihr Bezeichner mit der Komponente, die das Ereignis ausgelöst hat, verknüpft werden. Diese Verknüpfung wird auch *Registrierung* genannt. So kann z.B. festgelegt werden, dass immer dann, wenn sich der Wert in einem Texteingabefeld ändert, eine Methode aufgerufen wird, die die Gültigkeit des neuen Wertes prüft. Die Ereignisbehandlung in JSF-Anwendungsprogrammen wird in Kapitel 19 aufgegriffen.

14.1.9. Navigation

Möchte eine Benutzerin von einer Seite zu einer anderen Seite wechseln, so hat dieser Ablauf immer bestimmte Vorbedingungen und bestimmte Nachbedingungen und wird durch ein bestimmtes Ereignis ausgelöst. Diese Bedingungen und Ereignisse können in sogenannten Navigationsregeln festgehalten werden. JSF bietet eine implizite und eine explizite Navigation an. In Kapitel 18 werden beide Navigationstypen betrachtet.

15. Der JSF-Lebenszyklus

Für die Erstellung von JSF-Seiten sind auch Kenntnisse über die interne Verarbeitung durch das JSF-Framework wichtig. Wie bei JavaServer Pages ist der Lebenszyklus (Lifecycle) eines Requests in einer JSF-Anwendung durch mehrere Schritte beschrieben, die im Folgenden näher betrachtet werden.

Durch die Spezifikation von JavaServer Faces wird die Bearbeitung eines Requests von einem Client genau vorgegeben. Beginnend bei einer HTTP-Anfrage, die vom Client an einen Server gesendet wird, bis zum Zurücksenden des Antwortdokumentes an den Client, werden insgesamt sechs Verarbeitungsschritte durchgeführt, wie Abbildung 15.1 zeigt.

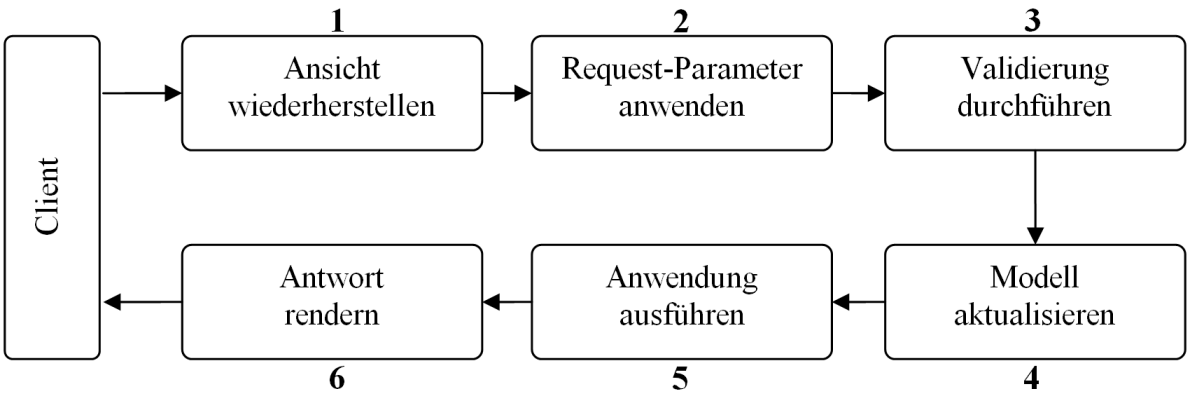

Abbildung 15.1.: JavaServer Faces - Lebenszyklus

In diesem Kapitel werden diese sechs Schritte des Lebenszyklus beschrieben.

Schritt 1 - Ansicht wiederherstellen

Wird von einem Client ein Request an einen Server gesendet, z.B. nachdem eine Schaltfläche auf einer JSF-Seite in einem Formular betätigt wurde, so wird im ersten Schritt die JSF-Seite intern wiederhergestellt. Alle Elemente bzw. Komponenten einer JSF-Seite werden in einem sogenannten *Komponentenbaum* organisiert, der die Struktur und die Zusammenhänge aller Komponenten definiert (z.B. ist die Schaltfläche zum Absenden eines Formulars der Formular-Komponente untergeordnet). Wird eine JSF-Seite zum ersten Mal von einem Client aufgerufen, so muss der Komponentenbaum für diese Seite neu erstellt werden. Unmittelbar nach dem initialen Aufbau des Komponentenbaums wird Schritt 6 des Lebenszyklus durchgeführt, die Ansicht wird *gerendert*[1]. Der Begriff „rendern“ bezeichnet hier die Erstellung des Ausgabedokumentes auf Basis

[1] rendern = engl. to render „machen, leisten, erbringen“

des Komponentenbaumes. Wurde die angefragte JSF-Seite bereits einmal von diesem Client aufgerufen, wird der Komponentenbaum aus dem Zustand wiederhergestellt, der auf dem Server gespeichert ist.

Schritt 2 - Request-Parameter anwenden

Nachdem der Komponentenbaum geladen oder neu erzeugt wurde, enthält er nur Informationen über die Komponenten generell, die in den Komponenten enthaltenen Werte wie z.B. der Text in einem Texteingabefeld oder das selektierte Element in einem DropDown-Menü sind den Komponenten noch nicht zugewiesen. Die zweite Phase des Lebenszyklus führt diese Zuweisungen aus. Hierfür wird eine Methode auf dem Wurzelelement des Komponentenbaumes aufgerufen, die rekursiv Methoden ihrer Kindelemente aufruft. Diese Methoden prüfen, ob für die jeweilige Komponente ein Wert in der Anfrage hinterlegt wurde. Ist dies der Fall, wird dieser Wert der Komponente zugewiesen.

Schritt 3 - Validierung durchführen

Nachdem der Komponentenbaum mit allen Komponenten und allen Request-Parametern erstellt wurde, werden die Werte der Request-Parameter konvertiert und validiert. Sofern keine benutzerdefinierten Konverter implementiert wurden, werden Standard-Konverter verwendet, die bereits durch das JSF-Framework bzw. den Application Server implementiert sind. Nach der Konvertierung aller Werte werden diese durch entsprechende Validatoren validiert. Nach welchen Regeln eine Validierung für eine Komponente stattfindet, wird durch die entsprechenden Tags, die einen Validator an eine Komponente bindet, festgelegt, die im Komponentenbaum Kindelemente der Komponente sind. Treten bei der Konvertierung oder Validierung Fehler auf, wird direkt mit Schritt 6 fortgefahren und die aktuelle Seite mit entsprechenden Fehlermeldungen ausgegeben. Tritt kein Fehler auf, wird jeder konvertierte und validierte Wert in seiner Komponente gespeichert.

Schritt 4 - Model aktualisieren

Sind alle Werte der Request-Parameter validiert und im Komponentenbaum eingetragen worden, müssen – zur weiteren Verarbeitung der Werte durch die Anwendungslogik – alle Managed Beans, die mit der JSF-Seite verknüpft sind, aktualisiert werden. Die Verknüpfung von Eigenschaften einer Managed Bean mit den Komponenten einer JSF-Seite ist durch Ausdrücke der Expression Language für die `value`-Attribute der Komponenten gegeben. Mit Hilfe der Setter-Methoden werden die Eigenschaften einer Managed Bean mit den entsprechenden Werten belegt.

Schritt 5 - Anwendung ausführen

Wurden die Werte der Request-Parameter in den mit der JSF-Seite verknüpften Managed Beans hinterlegt, wird die Anwendungslogik ausgeführt. Dies beinhaltet die Ausführung von Ereignisbehandlungsmethoden mit eventuellen Aufrufen von Methoden des Anwendungskerns. Auch wird festgelegt, welche JSF-Seite als nächstes aufgerufen wird, entweder durch Rückgabewerte

aufgerufener Methoden oder durch definierte Navigationsregeln.

Schritt 6 - Antwort rendern

Der letzte Schritt des Lebenszyklus besteht aus dem Rendern des Antwortdokumentes. Intern wird in dieser Phase wieder ein Komponentenbaum erzeugt, und Werte von Properties entsprechender Managed Beans werden an die Komponenten im Komponentenbaum der neuen JSF-Seite übergeben. Nachdem der Komponentenbaum der JSF-Seite erzeugt wurde, muss dieser gerendert werden, d.h. es wird durch den Komponentenbaum traversiert, und jede Komponente wird in ein XHTML-konformes Element übersetzt. Zudem wird der Status dieses Komponentenbaums gespeichert, sodass die Informationen bei einem erneuten Aufruf dieser Seite direkt verfügbar sind und kein neuer Baum konstruiert werden muss. Nach der Erstellung des Ausgabedokumentes wird dieses an den Client gesendet. Der Lebenszyklus der Anfrage ist beendet.

Diese Seite bleibt aus technischen Gründen frei!

16. Facelets als Seitendeklarationssprache

Mit einer Seitendeklarationssprache können Internetseiten definiert werden. Zur Entwicklung bzw. Definition von JSF-Seiten wird seit der JSF-Version 2.0 Facelets als Seitendeklarationssprache eingesetzt. Der Einsatz von JavaServer Pages wird zwar noch unterstützt, JSP wurde jedoch offiziell von JSF abgelöst.

In diesem Kapitel wird die JSF-Deklarationssprache (wir beziehen uns nun immer auf Facelets) näher betrachtet.

Eine JSF-Seite ist ein XHTML-Dokument, das aus Standard-XHTML-Tags und JSF-spezifischen Tags besteht. Um JSF-spezifische Tags nutzen zu können, müssen die entsprechenden Tag-Bibliotheken in das XHTML-Dokument eingebunden werden.

In einer JSP-Seite wurde die Direktive `taglib` genutzt, um eine Tag-Bibliothek in die Seite einzubinden. Da eine JSF-Seite nur aus XHTML-Tags besteht, wird die Einbindung von Tag-Bibliotheken mit Hilfe von *Namensräumen* (namespaces) umgesetzt. In einer XML- oder XHTML-Datei werden Namensräume dafür verwendet, mehrere XML-Sprachen innerhalb eines Dokumentes zu verwenden, in unserem Fall mehrere Tag-Bibliotheken. Enthalten zwei Tag-Bibliotheken Tags mit dem gleichen Bezeichner, so müssen diese Tags voneinander unterschieden werden können.

Die Standard-Tag-Bibliotheken in JSF sind zum einen die HTML-Tag-Bibliothek, für die der Namensraum `http://xmlns.jcp.org/jsf/html` und das Präfix `h` vorgesehen sind, zum andern die Core-Tag-Bibliothek, für die der Namensraum `http://xmlns.jcp.org/jsf/core` und das Präfix `f` vorgesehen sind. Die Namensräume in einer XHTML-Datei werden innerhalb des öffnenden `html`-Tags angegeben. Codeausschnitt 16.1 zeigt den Inhalt eines XHTML-Dokumentes mit der Einbindung beider JSF-Tag-Bibliotheken. Zu beachten ist, dass in einer XHTML-Datei auch der Namensraum für XHTML angegeben werden muss.

```
1  <html xmlns="http://www.w3.org/1999/xhtml"
2        xmlns:h="http://xmlns.jcp.org/jsf/html"
3        xmlns:f="http://xmlns.jcp.org/jsf/core">
4    <head>
5      <title>
6        Einbindung der JSF-Tag-Bibliotheken
7      </title>
8    </head>
9    <body>
10
11   </body>
12 </html>
```

Listing 16.1: Einbindung der JSF-Tag-Bibliotheken

16.1. Die HTML-Tag Library

Wie in Abschnitt 14.1 beschrieben wurde, werden die JSF-spezifischen Elemente, die durch die entsprechenden Tag-Paare in eine XHTML-Seite eingefügt werden, Komponenten genannt.

Die Menge aller Komponenten, die durch die HTML-Tag-Bibliothek für JSF gegeben ist, lässt sich in folgende Gruppen aufteilen:

- Eingabekomponenten
- Ausgabekomponenten
- Befehlskomponenten
- Auswahlkomponenten
- Formularkomponenten
- DataTable-Komponenten
- Bildausgabekomponenten
- Panel-Komponenten
- Nachrichtenkomponenten

Tabelle 16.1 zeigt eine Auswahl dieser Komponenten mit einer kurzen Beschreibung.

Alle Komponenten aus der JSF-HTML-Tag-Bibliothek werden in ein oder mehrere HTML-Elemente übersetzt, wenn die entsprechende JSF-Seite verarbeitet wird.

`h:body`	stellt den Rumpfbereich (Body) der Seite dar.
`h:head`	stellt den Kopfbereich (Head) der Seite dar.
`h:commandButton`	stellt eine HTML-Schaltfläche dar.
`h:commandLink`	stellt einen HTML-Link dar.
`h:inputText`	stellt ein HTML-Texteingabefeld dar.
`h:inputSecret`	stellt ein HTML-Passworteingabefeld dar (Zeichen sind nicht sichtbar).
`h:outputLabel`	stellt eine Bezeichnung <label>dar, die in der Regel an eine Eingabekomponente gebunden wird. Durch einen Klick auf die Bezeichnung (den Text), wird der Fokus auf die Eingabekomponente gesetzt (z.B. ein Texteingabefeld).
`h:outputText`	stellt einen einfachen Text auf der Seite dar.
`h:form`	stellt ein HTML-Formular dar.
`h:panelGrid`	Diese Komponente wird zur Gruppierung von anderen Komponenten genutzt.
`h:graphicImage`	stellt ein Bild dar und entspricht dem HTML-Bildelement.
`h:dataTable`	stellt eine Struktur dar, die in eine HTML-Tabelle übersetzt wird. Diese Struktur besteht aus Spalten und Zeilen.
`h:selectBooleanCheckbox`	stellt eine HTML-Kontrollbox dar, die entweder aktiviert oder deaktiviert werden kann.
`h:selectOneMenu`	stellt eine aufklappbare HTML-Liste dar. Aus dieser Liste kann genau ein Wert ausgewählt werden.
`h:selectOneListbox`	stellt eine HTML-Liste dar, aus der genau ein Element ausgewählt werden kann. Im Unterschied zur aufklappbaren Liste, sind mehrere Listenelemente gleichzeitig sichtbar. Ggfs. kann durch die Liste gescrollt werden.
`h:message`	stellt ein HTML-Element dar und wird verwendet, um eine Nachricht einer Komponente darzustellen. Über das `for`-Attribut kann das `message`-Elemente mit einer Komponente verbunden werden. Wird diese Komponente mit einem Validator validiert, wird im Falle einer nicht erfolgreichen Validierung eine Informationsnachricht des Validators über das `message`-Element ausgegeben.

Tabelle 16.1.: Komponenten der HTML-Tag-Library

16.2. Die Core-Tag Library

Die Core-Tag-Library des JSF-Frameworks beinhaltet Elemente, die Basisfunktionalitäten von JSF zur Verfügung stellen. Im Folgenden werden nur einige der 27 verfügbaren Elemente besprochen, die für diesen Kurs relevant sind.

Tabelle 16.2 zeigt diese Auswahl mit den entsprechenden Beschreibungen

Auf die beiden Tags `<f:valueChangeListener>` und `<f:actionListener>` wird in Kapitel 19 (Ereignisbehandlung in JavaServer Faces) noch einmal eingegangen.

Codeausschnitt 16.2 zeigt den Inhalt einer JSF-Seite, die eine Liste mit Ländernamen und eine Schaltfläche zum Absenden des selektierten Landes enthält. Codeausschnitt 16.3 zeigt die Ergebnisseite mit dem aus der Liste ausgewählten Land und Codeausschnitt 16.4 zeigt die Implementierung der Managed Bean, in der eine Liste der verfügbaren Länder und das ausgewählte Land gespeichert werden. In den Codeausschnitten werden nur die Rümpfe der JSF-Seiten dargestellt.

```
1  <h:body>
2      <h:form>
3          <h:selectOneListbox id="myList"
4                              value="#{listBean.selection}">
5              <f:selectItems value="#{listBean.countries}" />
6          </h:selectOneListbox>
7
8          <h:commandButton value="Absenden" action="result.xhtml" />
9      </h:form>
10 </h:body>
```

Listing 16.2: Beispiel für eine Listenkomponente: Auswahlseite

```
1  <h:body>
2      <h:form>
3          Folgendes Land wurde ausgewählt:
4         <h:outputLabel value="#{listBean.selection}" />
5      </h:form>
6  </h:body>
```

Listing 16.3: Beispiel für eine Listenkomponente: Ergebnisseite

In der Managed Bean sind zwei Properties deklariert: `countries` und `selection`. Die Property `selection` verfügt über einen Getter und einen Setter und speichert die Auswahl aus der Liste. Der Wert des `value`-Attributes der Listenkomponente enthält einen EL-Ausdruck, der den ausgewählten Wert an die Property bindet. Die Property `countries` beinhaltet die Liste aller Länder. Die Liste der Länder wird im Getter der Property erzeugt, falls sie noch nicht erzeugt wurde. Ein Setter wird hier nicht benötigt. Die Bindung der Länderliste an die Listenkomponente wird durch Angabe des `value`-Attributes des `selectItems`-Element erreicht

f:selectItem	Dieses Tag stellt einen Listeneintrag dar und wird zusammen mit einer Liste verwendet (z.B. <h:selectOneListbox>). Der sichtbare Text dieses Listeneintrags ist durch das Attribut itemLabel gegeben, der Wert, der durch diesen Eintrag repräsentiert wird, ist durch das Attribut itemValue gegeben.
f:selectItems	Dieses Tag stellt eine Menge von Listeneinträgen dar und wird zusammen mit einer Liste verwendet (z.B. <h:selectOneListbox>). Der Wert des value-Attributes verweist auf eine Liste einer Managed Bean (z.B. eine ArrayList, die eine Menge von Zeichenketten speichert).
f:ajax	Dieses Tag fügt einer Komponente asynchrones Verhalten hinzu. Auf die Anwendung der AJAX-Technologie in einer JSF-Anwendung wird in Kapitel 20 eingegangen.
f:valueChangeListener	Dieses Tag verbindet eine Komponente mit einer Klasse, die die Schnittstelle ValueChangeListener implementiert. Diese Klasse muss eine Methode zur Verfügung stellen, die den Wert der Komponente verarbeitet, sobald das übergeordnete Formular abgesendet wurde.
f:actionListener	Dieses Tag verbindet eine Komponente mit einer Klasse, die die Schnittstelle ActionListener implementiert.Diese Klasse muss eine Methode zur Verfügung stellen, die das durch die Komponente ausgelöste Ereignis verarbeitet. Das Tag <f:actionListener> wird zusammen mit Befehlskomponenten (z.B. eine Schaltfläche oder ein Link) verwendet.
f:attribute	Dieses Tag fügt einer Komponente ein Attribut hinzu. Der Name des Attributes wird durch das Tag-Attibut name, der Wert durch das Tag-Attribut value angegeben. Der Wert eines Attributes kann ein beliebiges Objekt sein.
f:param	Dieses Tag fügt einer Komponente einen Parameter hinzu. Der Name des Parameters wird durch das Attribut name, der Wert durch das Attribut value angegeben. Der Wert eines Parameters kann nur eine Zeichenkette sein.

Tabelle 16.2.: Elemente der Core-Tag-Library

```java
package kurseinheit5;

import java.util.ArrayList;
import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

@Named
@RequestScoped
public class ListBean {
    private ArrayList<String> countries;
    private String selection;

    public ArrayList<String> getCountries() {
        if (countries == null) {
            countries = new ArrayList<>();

            countries.add("Deutschland");
            countries.add("Italien");
            countries.add("Frankreich");
            countries.add("Spanien");
            countries.add("Griechenland");
        }

        return countries;
    }

    public String getSelection() {
        return selection;
    }

    public void setSelection(String selection) {
        this.selection = selection;
    }
}
```

Listing 16.4: Beispiel für eine Listenkomponente: ManagedBean

16.3. Validatoren

JavaServer Faces stellt mehrere Typen von Validatoren zur Verfügung, die die Eingaben von Benutzern auf Korrektheit prüfen. Ein Beispiel einer Komponente, deren Wert durch einen Validator validiert werden kann, ist eine Texteingabekomponente, die durch das Tag-Paar `<h:inputText> ... </h:inputText>` definiert wird.

Ein Validator wird durch entsprechende Tags einer Komponente zugewiesen. Codeausschnitt 16.5 zeigt eine Texteingabekomponente, der ein Validator zugewiesen wird, der überprüft, ob ein Wert in das Textfeld eingegeben wurde.

```
1 <h:inputText id="input" >
2    <f:validateRequired />
3 </h:inputText>
```

Listing 16.5: Angabe eines Validators für eine Texteingabekomponente

Folgende Tags stehen in JSF unter anderem zur Verfügung, um einer Komponente einen Validator zuzuweisen:

- `<f:validateRequired />`
  Der entsprechende Validator überprüft, ob überhaupt ein Wert in die Komponente eingegeben wurde.
- `<f:validateLength minimum="5" maximum="10" />`
  Der entsprechende Validator überprüft, ob sich die Länge der entsprechenden Zeichenkette in dem durch die Attribute minimum und maximum definierten Intervall befindet.
- `<f:validateLongRange minimum="20" maximum="1000" />`
  Der entsprechende Validator überprüft, ob sich die Ganzzahl, gegeben durch den Wert der Komponente, in dem durch die Attribute minimum und maximum definierten Intervall befindet.
- `<f:validateDoubleRange minimum="1.5" maximum="3.7" />`
  Der entsprechende Validator überprüft, ob sich die Fließkommazahl, gegeben durch den Wert der Komponente, in dem durch die Attribute minimum und maximum definierten Intervall befindet.
- `<f:validateRegex pattern="[0-9]" />`
  Der entsprechende Validator überprüft, ob die angegebene Zeichenkette der Komponente dem im Attribut pattern angegebenen regulären Ausdruck entspricht. Der hier angegebene *reguläre Ausdruck*[1] lässt z.B. nur Zahlen mit Ziffern von 0 bis 9 zu.
- `<f:validator validatorID="myCustomValidator" />`
  Dieses Element weist der Komponente eine Instanz eines benutzerdefinierten Validators zu. Der Wert des `validatorID`-Attributes verweist auf diese Instanz.

Codeausschnitt 16.6 zeigt eine JSF-Seite mit drei Texteingabefeldern und einer Schaltfläche zum Validieren aller Eingaben. Jedem Texteingabefeld wird ein Validator zugewiesen. Für das erste Textfeld muss eine Eingabe erfolgen, die Länge des zweiten Textfeldes muss zwischen vier und acht Zeichen liegen und in das dritte Textfeld dürfen nur Buchstaben eingegeben werden (kleine und große). Für die Schaltfläche wird in diesem Beispiel kein `action`-Attribut angegeben. Durch einen Klick auf die Schaltfläche wird das Formular verarbeitet und die Seite neu geladen. Zur Vereinfachung wurde in diesem Codeausschnitt nur der Inhalt des Rumpfes angegeben.

[1] Ein regulärer Ausdruck beschreibt eine Menge von Zeichenketten, die auch von einem Endlichen Automaten beschrieben oder erkannt werden kann.

```
<h:body>
<h:form>
<h:outputLabel for="input" value="Eingabe: " />
   <h:inputText id="input" validatorMessage="Falsche Eingabe.">
      <f:validateRequired />
   </h:inputText>
   <h:inputText id="input" validatorMessage="Falsche Eingabe.">
      <f:validateLength minimum="4" maximum="8" />
   </h:inputText>
   <h:inputText id="input" validatorMessage="Falsche Eingabe.">
      <f:validateRegex pattern="[a-zA-Z]+" />
   </h:inputText>
   <h:message for="input" />

   <h:commandButton value="Validieren" />
</h:form>
</h:body>
```

Listing 16.6: Einsatz von Validatoren für Texteingabefelder

Wird eine Eingabe durch einen Validator geprüft und ist diese Überprüfung nicht erfolgreich, d.h. der eingegebene Wert entspricht nicht einem vom Validator erwarteten Wert, so wird eine Informationsnachricht durch das JSF-Framework erzeugt.

Eine JSF-Anwendung kann sich in unterschiedlichen *Stufen* befinden, z.B. in der *Entwicklungsstufe* (Development) oder in der *Produktionsstufe* (Production). In welcher Stufe sich die Web-Anwendung befindet, kann im Deployment Descriptor `web.xml` festgelegt werden. Standardmäßig (keine Angabe im Deployment Descriptor) ist die Entwicklungsstufe aktiviert. In dieser Stufe werden Informationsnachrichten, die durch Validatoren erzeugt werden, auf der entsprechenden JSF-Seite angezeigt, ohne dass ein Entwickler dies explizit veranlasst hat. Befindet sich die Web-Anwendung in der Produktionsstufe, die dem Endprodukt entspricht, werden diese Nachrichten nicht angezeigt. Um dennoch Informationen über falsche Eingaben zu erhalten, wird das `message`-Element verwendet. Dieses ist mit der Komponente verknüpft, dessen Wert durch einen Validator überprüft wird und enthält nach der Überprüfung die Informationsnachricht des Validators als Zeichenkette. Das JSF-Framework erstellt diese Informationsnachricht automatisch. Um den Text der Informationsnachricht anzupassen, muss im öffnenden Tag der Komponente das Attribut `validatorMessage` hinzugefügt werden. Der Wert dieses Attributes entspricht der anzuzeigenden Informationsnachricht. In Codeausschnitt 16.6 wurde die Informationsnachricht angepasst.

17. Managed Beans

Im vorherigen Kapitel wurde die Seitendeklarationssprache Facelets vorgestellt. Mit ihr können die Inhalte von JSF-Seiten in XHTML-Schreibweise definiert werden. Eine JSF-Seite besteht generell aus Standard-HTML-Tags und JSF-spezifischen Tags.

Im Rahmen der MVC-Architektur stellt die Menge der definierten JSF-Seiten die View (Ansicht) dar. Der Controller ist durch das Faces Servlet gegeben.

Managed Beans sind ein zentraler Bestandteil von JavaServer Faces und sind innerhalb einer Java EE-Anwendung Teil des Models, sie dienen als Schnittstelle zwischen der Benutzungsoberfläche und den Methoden der Geschäftslogik. Für die Trennung der Präsentation und der Programmlogik spielen Managed Beans daher eine wichtige Rolle. Eine Managed Bean ist eine einfache Java-Klasse, die den Anforderungen einer Java-Bean genügt. Außerdem muss diese Klasse durch die Annotation `@Named` definiert sein.

17.1. Implementierung einer Managed Bean

Im folgenden kleinen Beispiel wird gezeigt, wie eine Managed Bean definiert wird. Codeausschnitt 17.1 zeigt eine Managed Bean `Login`, die den Benutzernamen und das Passwort eines Benutzers speichern soll.

```
1  @Named
2  @SessionScoped
3  public class Login
4  {
5     private String username;
6     private String password;
7
8     public void setUsername(String username) {
9        this.username = username;
10    }
11
12    public String getUsername() {
13       return this.username;
14    }
15
16    public void setPassword(String password) {
17       this.password = password;
18    }
19
```

```
20      public String getPassword() {
21          return this.password;
22      }
23  }
```

Listing 17.1: Defintion einer Managed Bean

Um die Klasse `Login` als Managed Bean auszuzeichnen, wird die Annotation `@Named` in Zeile 1 verwendet. In Zeile 2 wird der Scope der Klasse angegeben. In diesem kleinen Beispiel soll die Managed Bean `Login` im Session-Scope verfügbar sein. Zeilen 5 und 6 deklarieren die Properties `username` und `password` mit der Sichtbarkeit `private`, wie es von einer JavaBean verlangt wird. Ab Zeile 8 werden die Getter und Setter der beiden definierten Properties aufgeführt.

Um die Annotationen `@Named` und `@SessionScoped` verwenden zu können, müssen die entsprechenden Annotationstypen importiert werden. Dies geschieht durch die folgenden zwei `import`-Anweisungen, die an den Anfang der Datei geschrieben werden müssen, die die Klassendefinition enthält:

```
import javax.inject.Named;
import javax.enterprise.context.SessionScoped;
```

17.2. Wichtige Scopes für Managed Beans

Im Codeausschnitt 17.1 wurde die Managed Bean Login für die Nutzung innerhalb des Session Scope definiert.

Das Java-Paket `javax.enterprise.context` stellt sechs Scope-Annotationen zur Verfügung, mit denen Managed Beans näher beschrieben werden können. Im Abschnitt „Scoped“ der Kurseinheit 3 wurden bereits die Scopes Request Scope, Session Scope und Application Scope für die Verwendung mit Servlets beschrieben. Diese drei Scopes können auch für Managed Beans verwendet werden. Dieser Abschnitt behandelt noch einen weiteren Scope, den Conversational Scope, der Normal Scope und der Scope Dependent werden hier nicht weiter erläutert.

Eine Managed Bean innerhalb des *Request Scope* existiert nur für die Dauer eines Request-Durchlaufs. Stellen wir uns eine JSF-Seite mit einer verknüpften Managed Bean vor (weiteres zu dieser Verknüpfung in Abschnitt 17.3). Auf der JSF-Seite befindet sich ein Formular, in das Daten eingetragen werden. Wird dieses Formular abgesendet, erzeugt dies einen Request. Durch den Schritt 4 des JSF-Lebenszyklus (siehe Abschnitt 15) werden den Properties der Managed Bean die eingegebenen Daten im Formular als Werte übergeben. Die Weiterleitung an eine zweite JSF-Seite (z.B. zur Anzeige der in das Formular eingegebenen Daten) ist noch Teil des Requests. Diese JSF-Seite kann wieder mit Hilfe der Expression Language auf die Properties der Managed Bean im Request Scope zugreifen und kann die gespeicherten Daten darstellen. Mit Beendigung des Requests wird auch die Instanz der Managed Bean im Request Scope zerstört und kann nicht weiter verwendet werden.

Eine Managed Bean innerhalb des *Conversation Scope* existiert für die Dauer einer *Conversation* (Konversation). Eine Konversation ist als Menge von mehreren Requests zu verstehen. Diese Namensgebung wird verständlich, wenn man sich einen Assistenten innerhalb einer Web-Anwendung vorstellt, der die Benutzer durch mehrere Bearbeitungsstufen führt. So könnte z.B. eine Umfrage aus drei Stufen (drei JSF-Seiten) bestehen, von der ersten Seite gelangt man zur zweiten, von der zweiten zur dritten und auf der dritten Seite befindet sich eine Schaltfläche zum Beenden der Umfrage. Jede JSF-Seite enthält ein Formular, in das Daten eingetragen werden sollen. Die Konversation startet mit dem Aufruf der ersten Seite und wird mit dem Betätigen der Beenden-Schaltfläche beendet. Eine Managed Bean im Conversation Scope hat den Vorteil, dass nicht für jeden Request bzw. für jede JSF-Seite eine Instanz einer entsprechenden Managed Bean erzeugt werden muss. Vielmehr können alle Daten der Umfrage in einer einzigen Managed Bean gespeichert und beim Beenden der Konversation bearbeitet werden. Die Lebenszeit des Conversation Scope wird vom Entwickler im Code gesteuert.

Eine Managed Bean innerhalb des *Session Scope* existiert für die Dauer der aktuellen Session. In der Regel definiert diese den Zeitraum, für den eine Benutzerin die Web-Anwendung kontinuierlich nutzt. Ein anschauliches Beispiel ist die Internetseite eines Online Shops, der für jede Benutzerin einen Warenkorb zur Verfügung stellt, in den die gewünschten Waren abgelegt werden können. Der Inhalt des Warenkorbs soll während des gesamten Zeitraums des Einkaufs bestehen bleiben. Die Speicherung des Warenkorbs kann eine Managed Bean übernehmen, die alle Waren in einer Liste speichert. Erst nachdem die Benutzerin den Einkauf tätigt, sich abmeldet oder z.B. den Internetbrowser schließt, sollen die Waren aus dem Warenkorb gelöscht werden, d.h. die Instanz der Managed Bean wird zerstört.

Der letzte Scope, den wir betrachten ist der *Application Scope* . Befindet sich eine Managed Bean in diesem Scope, so wird eine Instanz von ihr beim Starten der Web-Anwendung auf dem Server erzeugt, die erst beim Beenden der Web-Anwendung zerstört wird. Die Besonderheit ist, dass diese Instanz für alle Benutzer verfügbar ist. Als einfaches Beispiel stelle man sich eine Managed Bean vor, die eine Zählervariable beinhaltet, die speichert, wie oft die Internetseite insgesamt aufgerufen wurde. Diese Zählervariable muss für alle Benutzer zur Verfügung stehen und wird immer dann inkrementiert, wenn die Internetseite aufgerufen wird.

Die Wahl des Scope für eine Managed Bean sollte stets gut überlegt und auf das jeweilige Szenario angepasst sein. Gerade in Bezug auf den verfügbaren Arbeitsspeicher des Servers und auf Web-Anwendungen mit sehr vielen Benutzern sollte sorgsam auf den Speicherverbrauch geachtet werden, um schlechte Performanz oder gar den Absturz der Web-Anwendung zu vermeiden.

17.3. Zusammenspiel von Managed Beans und Facelets

Nachdem in Kapitel 16 Facelets als Seitendeklarationssprache und in den vorherigen Abschnitten dieses Kapitels Managed Beans als Server-seitige Klassen und somit als Teil des Model in der MVC-Architektur vorgestellt wurden, betrachten wir in diesem Abschnitt die Verknüpfung

von Facelets und Managed Beans. Wie in Abschnitt 14.1 bereits angesprochen wurde, wird für diese Verknüpfung die Expression Language verwendet. Wir betrachten ein einfaches Beispielprogramm, das aus zwei JSF-Seiten und einer Managed Bean besteht. Die Seite `start.xhtml` enthält ein Formular mit einem Texteingabefeld und einer Schaltfläche zum Absenden des Formulars an den Server. Der im Texteingabefeld eingegebene Wert wird in einer Property der Managed Bean gespeichert. Das Absenden des Formulars leitet die Benutzerin auf die Seite `result.xhtml` weiter, die den in der Managed Bean gespeicherten Wert der Property ausliest und anzeigt.

```
1  ...
2  <html xmlns="http://www.w3.org/1999/xhtml"
3        xmlns:h="http://xmlns.jcp.org/jsf/html">
4      <h:head>
5          <title>Facelets, Managed Bean und EL</title>
6      </h:head>
7      <h:body>
8          <h:form>
9              <h:outputLabel for="input" value="Eingabe: "/>
10             <h:inputText id="input" value="#{myManagedBean.input}" />
11             <h:commandButton action="result.xhtml" value="Absenden" />
12         </h:form>
13     </h:body>
14 </html>
```

Listing 17.2: Inhalt der JSF-Seite start.xhtml

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
3    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
4  <html xmlns="http://www.w3.org/1999/xhtml"
5        xmlns:h="http://xmlns.jcp.org/jsf/html">
6      <h:head>
7          <title>Facelets, Managed Bean und EL</title>
8      </h:head>
9      <h:body>
10         <h:outputLabel value="Der eingegebene Text ist: " />
11         <h:outputLabel value="#{myManagedBean.input}" />
12     </h:body>
13 </html>
```

Listing 17.3: Inhalt der JSF-Seite result.xhtml

Bei Texteingabe- und der Textausgabekomponente der Codeausschnitte 17.2 und 17.3 wird der Wert für das `value`-Attribut durch einen Ausdruck der Expression Language definiert.

```
1  package kurseinheit5;
2
3  import javax.inject.Named;
4  import javax.enterprise.context.RequestScoped;
5
6  @Named
```

```
7  @RequestScoped
8  public class MyManagedBean
9  {
10     private String input;
11
12     public String getInput() {
13         return input;
14     }
15
16     public void setInput(String input) {
17         this.input = input;
18     }
19 }
```

Listing 17.4: Inhalt der Datei MyManagedBean.java

In einem EL-Ausdruck wird auf eine Instanz einer Managed Bean (zur Vereinfachung sprechen wir im Folgenden von Zugriffen auf eine Managed Bean und nicht von Zugriffen auf ihre Instanz) durch ihren Bezeichner zugegriffen. In Codeausschnitt 17.4 ist dieser Bezeichner durch „MyManagedBean“ gegeben. Implizit kann mit der EL auf diese Managed Bean zugegriffen werden, indem der erste Buchstabe klein geschrieben wird: „myManagedBean“. Alternativ kann der Managed Bean explizit ein Name zugewiesen werden, mit dem in der EL auf die Managed Bean zugegriffen wird. Der Name wird dabei als Attribut der Annotation `@Named` angegeben: `@Named("einAndererName")`.

Um auf eine Property (Eigenschaft) einer Mananged Bean zugreifen zu können, muss der Bezeichner der Managed Bean, gefolgt von einem Punkt und dem Bezeichner der Property, angegeben werden. Auch der Bezeichner der Property muss mit einem Kleinbuchstaben beginnen. Wird auf eine Property einer Managed Bean durch die EL lesend zugegriffen, wird implizit die `get`-Methode verwendet. Wird schreibend auf die Property zugegriffen, wird implizit die `set`-Methode mit dem entsprechenden Parameterwert ausgeführt. Es ist nicht notwendig diese Methoden explizit aufzurufen.

In dem Beispielprogramm gibt die Benutzerin einen Wert in das Texteingabefeld ein. Das Absenden des Formulars bewirkt durch den EL-Ausdruck `myManagedBean.input` die Speicherung des eingegebenen Wertes in der Property der Managed Bean.

Neben der Bindung von Komponenten einer JSF-Seite an Properties einer Managed Bean können mit Hilfe der Expression Language auch Methoden von Managed Beans aufgerufen werden. Die jeweils aufzurufende Methode muss dafür öffentlich in der Managed Bean deklariert werden (Schlüsselwort: `public`).

Soll z.B. der anzuzeigende Wert einer Textausgabekomponente durch den Rückgabewert einer Methode der Managed Bean `myManagedBean` festgelegt werden, ist dies folgendermaßen zu erreichen:

```
<h:outputLabel id="text" value="#{myManagedBean.getWert()"} />
```

Die Methode `getWert()` in der Managed Bean ist dann folgendermaßen zu definieren:

```
public String getWert()
{
   return "Das ist der Wert";
}
```

18. Navigation in JSF

Web-Anwendungsprogramme generieren meist nicht nur eine einzige Internetseite, sondern vielmehr eine Menge von Internetseiten. In JSF-Anwendungen entspricht jede Seite einer XHTML-Datei, deren Komponenten durch die Seitendeklarationssprache beschrieben sind (Facelets in JSF-Anwendungen). Um von einer Seite auf eine andere Seite wechseln zu können, müssen Seiten Befehlskomponenten enthalten. Eine Befehlskomponente ist eine Schaltfläche oder ein Link (siehe Kapitel 16)[1].

Unter *Navigation* versteht man innerhalb einer JSF-Anwendung den Wechsel von einer Internetseite zu einer anderen. Dieser Wechsel wird meist durch eine Aktion, z.B. die Betätigung einer Schaltfläche oder eines Links, ausgelöst. Vor Version 2.0 von JavaServer Faces gab es nur eine Art von Navigation: die *explizite Navigation*. Seit Version 2.0 ist die *implizite Navigation* hinzugekommen.

Spezifiziert wird eine Navigation durch vier Elemente: eine *Vorbedingung*, ein *Ereignis*, eine *Bedingung* und eine *Nachbedingung*. Die Vorbedingung drückt aus, von wo aus die Navigation gestartet wird, das Ereignis löst eine Navigation erst aus, durch die Prüfung der Bedingung wird festgestellt, ob die Navigation beendet wird oder nicht, und die Nachbedingung spezifiziert das Ziel der Navigation.

Bei der expliziten Navigation werden diese Elemente direkt angegeben, als Regeln der Art: „Wenn ich mich auf der Seite `start.xhtml` befinde und die Schaltfläche senden betätige, die eine Methode aufruft, und der Rückgabewert dieser Methode die angegebene Bedingung erfüllt, leite mich auf die Seite `result.xhtml` weiter“.

Eine Navigation kann zudem *statisch* oder *dynamisch* sein. Eine statische Navigation liegt dann vor, wenn der Navigationspfad (z.B. zu einer Internetseite) zur Entwicklungszeit angegeben wurde und sich während der Laufzeit des Anwendungsprogramms nicht ändert. Bei einer dynamischen Navigation wird der Nagivationspfad erst zur Laufzeit festgelegt, z.B. durch den Aufruf einer Methode, die einen Navigationspfad erstellt und ihn als Wert zurückgibt.

In diesem Kapitel werden beide Navigationsarten betrachtet, beginnend mit der expliziten Navigation in Abschnitt 18.1.

[1] Neben dem Wechsel von einer Internetseite zu einer anderen mit Hilfe von Befehlskomponenten ist eine Umsetzung der Navigation durch JavaScript möglich. Durch die Verwendung von JavaScript (und ggf. weiteren JavaScript-Bibliotheken, wie z.B. jQuery) kann jede Komponente einer HTML-Seite als Befehlskomponente fungieren.

18.1. Explizite Navigation

Um die Navigation in einer JSF-Anwendung explizit zu spezifizieren, müssen sogenannte Navigationsregeln definiert werden. Die Menge aller Navigationsregeln wird in einer Konfigurationsdatei festgehalten, die standardmäßig die Bezeichnung `faces-config.xml` trägt. Hierbei handelt es sich um eine XML-Datei, die spezielle Konfigurationen für das JSF-Framework beinhaltet. Sie ist zu vergleichen mit dem Deployment Descriptor `web.xml` einer Java EE-Anwendung. Neben der Angabe der Navigationsregeln gibt es noch eine Reihe weiterer Konfigurationen, die in der Datei `faces-config.xml` angegeben werden können, auf die hier jedoch nicht eingegangen wird.

Wie in Kapitel 14.1 erläutert wurde, stellt das Faces Servlet den Controller innerhalb der MVC-Architektur von JavaServer Faces dar. In einer JSF-Anwendung wird im Deployment Descriptor `web.xml` durch die Angabe eines Servlet Mapping festgelegt, welche Anfragen von diesem Servlet verarbeitet werden. Die Steuerung dieser Anfragen, die Navigation, wird durch die Navigationsregeln bestimmt.

Codeausschnitt 18.1 zeigt das Gerüst der Datei `faces-config.xml` ohne Konfigurationselemente.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <faces-config version="2.2"
3                xmlns="http://xmlns.jcp.org/xml/ns/javaee"
4                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5                xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
6            http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
7      .
8      .
9      .
10 </faces-config>
```

Listing 18.1: Gerüst der Konfigurationsdatei faces-config.xml

An die Stelle der drei Punkte werden die Navigationsregeln der Web-Anwendung eingesetzt.

Eine Navigationsregel wird innerhalb der Konfigurationsdatei durch das Tag `<navigation-rule>` eingeleitet und durch das Tag `</navigation-rule>` abgeschlossen. In der Konfigurationsdatei können mehrere dieser Navigationsregeln definiert werden.

Um die Vorbedingung anzugeben, d.h. um festzulegen von welcher Internetseite aus die Navigation gestartet werden soll, werden die Tags `<from-view-id> ... </from-view-id>` verwendet. Innerhalb dieses Tag-Paares steht der Pfad der Ausgangsseite, z.B.

`<from-view-id>/start.xhtml</from-view-id>`.

Neben der Ausgangsseite für die Navigation kann eine Navigationsregel mehrere sogenannte Navigationsfälle beinhalten. Ein Navigationsfall wird durch die Tags `<navigation-case> ... </navigation-case>` eingeschlossen und kann (unter anderem) drei Elemente enthalten:

- `<from-action> ... </from-action>`
  gibt eine Methode durch einen Ausdruck der Expression Language an, die eine Zeichenkette zurückgibt. Dieses Element ist meist nur dann notwendig, wenn eine Navigation nicht eindeutig ist. Der Aufruf dieser Methode kann als Teil des Ereignisses einer Navigation angesehen werden.

- `<from-outcome> ... </from-outcome>`
  gibt eine Zeichenkette an, die entweder mit dem Rückgabewert der angegebenen Methode oder dem `action`-Attribut einer Befehlskomponente verglichen wird. Durch die Angabe dieses Elementes wird die Bedingung der Navigation spezifiziert.

- `<to-view-id> ... </to-view-id>` gibt den Pfad der Seite an, die Ziel der Navigation und somit Nachbedingung ist.

Zwei für die Navigation in JSF-Anwendungen wichtige Befehlskomponenten sind die Schaltfläche (`commandButton`) und der Link (`commandLink`). Beide Befehlskomponenten ermöglichen das Auslösen einer Navigation (z.B. durch einen Mausklick auf die Komponente). Das Attribut `action`, das für beide Komponenten zur Verfügung steht, spielt hierbei eine zentrale Rolle.

Als Wert für das `action`-Attribut muss eine Zeichenkette angegeben werden. Codeausschnitt 18.2 zeigt eine Schaltfläche und einen Link in einem Formular.

```
1 <h:form>
2     <h:commandButton value="Senden" action="goToPage2" />
3     <h:commandLink value="Senden" action="goToPage3" />
4 </h:form>
```

Listing 18.2: Die Befehlskomponenten Schaltfläche und Link

Angenommen, das Formular aus Codeausschnitt 18.2 befindet sich in einer JSF-Seite mit dem Namen `index.xhtml`. Durch die Schaltfläche soll eine Navigation gestartet werden, die zur Seite `page2.xhtml` führt, und durch den Link soll eine Navigation gestartet werden, die zur Seite `page3.xhtml` führt. Dann ist eine Navigationsregel mit zwei Navigationsfällen in der Datei `faces-config.xml` anzugeben. Codeausschnitt 18.3 zeigt den Inhalt dieser Datei.

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <faces-config version="2.2"
3                xmlns="http://xmlns.jcp.org/xml/ns/javaee"
4                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5                xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
6            http://xmlns.jcp.org/xml/ns/javaee/web-facesconfig_2_2.xsd">
7
8      <navigation-rule>
9          <from-view-id>/index.xhtml</from-view-id>
10         <navigation-case>
11             <from-outcome>goToPage2</from-outcome>
12             <to-view-id>/page2.xhtml</to-view-id>
13         </navigation-case>
14         <navigation-case>
```

```
15              <from-outcome>goToPage3</from-outcome>
16              <to-view-id>/page3.xhtml</to-view-id>
17          </navigation-case>
18      </navigation-rule>
19  </faces-config>
```

Listing 18.3: Eine Navigationsregel mit zwei Navigationsfällen in faces-config.xml

Für das Tag `<from-view-id>` wurde die Ausgangsseite `index.xhtml` angegeben, die das Formular und die beiden Befehlskomponenten enthält. Der erste Navigationsfall bezieht sich auf die Schaltflächen-Komponente, der zweite Navigationsfall auf die Link-Komponente. Der Inhalt der Tags `<from-outcome>` muss mit den Werten der `action`-Attribute der Befehlskomponenten übereinstimmen, damit der jeweilige Navigationsfall eintritt. Die Tags `<to-view-id>` geben jeweils die Seite an, zu der navigiert werden soll, in diesem Fall `page2.xhtml` bei Betätigung der Schaltfläche und `page3.xhtml` bei Betätigung des Links.

Die hier vorgestellte Navigationsart wird als *statische Navigation* (als Unterart der expliziten Navigation) bezeichnet, da alle Navigationspfade bereits zur Entwicklungszeit feststehen. Bei einer statischen Navigation wird das Tag `<from-action>` nicht benötigt.

Soll eine JSF-Anwendung den Navigationspfad zur Laufzeit *dynamisch* bestimmen, kann der Wert für das *action*-Attribut einer Befehlskomponente durch einen Ausdruck der Expression Language ersetzt werden, der eine Methode einer Java-Klasse referenziert. Dies kann z.B. eine Methode einer Managed Bean sein. Die Methode muss eine Zeichenkette zurückgeben, die mit dem Wert des Tag `<from-outcome>` der Datei `faces-config.xml` verglichen wird.

Codeausschnitt 18.4 zeigt die Methode der Managed Bean `NavigationBean`. Diese gibt je nach Programmablauf eine entsprechende Zeichenkette zurück. Die Konfigurationsdatei muss für dieses Beispiel nicht geändert werden Allerdings wird die Link-Komponente nicht mehr benötigt, da wir unter der entsprechenden erfüllten Bedingung beide Seiten durch Betätigen der Schaltfläche erreichen können. Codeausschnitt 18.5 zeigt das angepasste Formular-Element.

```
1  public String navigate() {
2      if (...) {
3          return "goToPage2";
4      }
5      else {
6          return "goToPage3";
7      }
8  }
```

Listing 18.4: Methode zur dynamischen Erzeugung eines Navigationspfades

```
1  <h:form>
2      <h:commandButton value="Senden" action="#{navigationBean.navigate()}"
           />
3  </h:form>
```

Listing 18.5: Dynamische Navigation durch eine Schaltfläche

Wird die Schaltfläche betätigt, wird durch den Rückgabewert der aufgerufenen Methode bestimmt, welcher Navigationsfall eintritt und zu welcher Seite navigiert wird.

18.2. Implizite Navigation

Mit der Version 2.0 von JavaServer Faces wurde die *implizite Navigation* eingeführt. Die implizite Navigation soll die vorher notwendige Erstellung von Navigationsregeln umgehen und die Implementierung von Navigationen in einer JSF-Anwendung vereinfachen.

In vorherigen Codeausschnitten wurde oft eine Schaltfläche mit einem `action`-Attribut definiert. Der Wert dieses Attributes enthielt den Pfad einer JSF-Seite.

```
<h:commandButton value="Senden" action="seite2.xhtml" />
```

Hier wurde bereits das Konzept der impliziten Navigation genutzt. Für das `action`-Attribut wird zwar wieder (wie bei der expliziten Navigation) eine Zeichenkette angegeben, das JSF-Framework erkennt jedoch, dass es eine Seite mit diesem Dateinamen gibt, und die Navigation kann ohne explizite Angabe einer Navigationsregel durchgeführt werden. Dies erleichtert die Schreibarbeit für Entwickler, denn es wird keine Konfigurationsdatei (`faces-config.xml`) benötigt, sofern keine anderen Konfigurationen vorgenommen werden müssen. Der Nachteil dieser Methode ist, dass alle Angaben zu Navigationen dezentral definiert sind. Einige Entwicklungsumgebungen stellen eine grafische Darstellung der Navigationsregeln der `faces-config.xml` zur Verfügung, wofür die XML-Datei ausgelesen wird.

Wird für das `action`-Attribut der Pfad einer Seite direkt angegeben, spricht man im Rahmen der impliziten Navigation, wie auch bei der expliziten Navigation von statischer Navigation. Um eine dynamische Navigation durchzuführen, kann der Navigationspfad durch einen Methodenaufruf zur Laufzeit bestimmt werden. Die Methode, die dafür mit Hilfe der Expression Language aufgerufen wird, muss einen gültigen Navigationspfad zurückgeben und nicht nur eine einfache Zeichenkette.

Für die Schaltfläche

```
<h:commandButton value="Senden" action="#{navigationBean.getPage()}"/>
```

zeigt Codeausschnitt 18.6 die Managed Bean MyBean mit der Methode `getPage()`, die einen Dateinamen als Navigationspfad zurückgibt.

```java
package kurseinheit5;

import javax.inject.Named;
import javax.enterprise.context.RequestScoped;

@Named
@RequestScoped
public class NavigationBean {
    public String getPage() {
        return "seite2.xhtml";
    }
}
```

Listing 18.6: Dynamische Navigation im Rahmen der impliziten Navigation

19. Ereignisbehandlung in JSF

Ein essentieller Bestandteil von Web-Anwendungen ist die Behandlung von Benutzereingaben. In den letzten Jahren wurden immer mehr Techniken eingeführt, die die einfache Entwicklung von Web-Anwendungen ermöglichen, deren grafische Benutzungsoberflächen denen von Desktop-Anwendungen ähneln. Basiselemente wie Schaltflächen, Textfelder oder Listen werden bereits von HTML unterstützt. Durch die gestalterischen Möglichkeiten durch Hinzunahme von Cascading Style Sheets und das Hinzufügen von Dynamik durch JavaScript können web-basierte grafische Benutzungsoberflächen komplexer und benutzerfreundlicher gestaltet werden.

Steht einer Benutzerin einer Web-Anwendung eine grafische Benutzungsoberfläche zur Verfügung, gibt es meist eine Vielzahl von Aktionen, die sie durchführen kann. Z.B. kann eine Schaltfläche betätigt, ein Text eingegeben oder geändert, ein Kontrollkästchen aktiviert oder deaktiviert oder ein Listeneintrag ausgewählt werden.

Um mit diesen Interaktionen umgehen zu können, wurde das sogenannte *„Event/Listener"-Modell* entwickelt. Jede Interaktion bzw. jede Aktion einer Benutzerin löst ein sogenanntes *Ereignis* (engl. event) aus. Das Element, für das die Aktion ausgeführt wird, wird als *Auslöser* bezeichnet. Darüber hinaus können Ereignisse auch durch den Application Server ausgelöst werden, ohne dass eine Benutzerin eine Aktion durchführen muss. Jedes Ereignis muss von der Web-Anwendung auf geeignete Art und Weise behandelt werden. Die Behandlung von Ereignissen führen sogenannte *Ereignis-Abhörer* bzw. *Interessenten* (engl. listener, observer) durch. Ein Interessent stellt eine *Ereignisbehandlungsroutine* (engl. event handler) zur Verfügung, die immer dann aufgerufen wird, wenn ein entsprechendes Ereignis eintritt. Um eine Verbindung zwischen einem Ereignis und einer Ereignisbehandlungsroutine zu erstellen, muss der Interessent mit seiner Ereignisbehandlungsroutine bei diesem Ereignis registriert werden.

Ein Ereignis beschreibt somit „was und wem etwas passiert ist" und eine Ereignisbehandlungsroutine beschreibt „was macht das Anwendungsprogramm, wenn das Ereignis eintritt". Im Folgenden werden die englischen Begriffe Event für Ereignis, Listener für Interessent und Event Handler für Ereignisbehandlungsroutine verwendet.

JavaServer Faces unterstützt die ereignisorientierte Programmierung. Im weiteren Verlauf dieses Kapitels geben wir einen Überblick über die in JSF unterstützten Ereignisse und die dafür zur Verfügung stehenden Ereignisbehandlungsroutinen.

19.1. Events und Listeners in JSF

Die *Event-Modellarchitektur* von JavaServer Faces übernimmt Konzepte aus der Java-Beans Spezifikation. Dort stellen Events einen Mechanismus dar, der Informationen über Zustandsänderungen ausgehend von *Quellenobjekten* an *Zielobjekte* verbreitet. Ein Quellenobjekt kann z.B. eine Schaltfläche sein, ein Zielobjekt ist ein Listener. Durch die Betätigung der Schaltfläche wird ein Event-Objekt erzeugt, das Informationen über das Event selber und Informationen über das Quellenobjekt enthalten kann. Dieses Event-Objekt wird anschließend an das am Quellenobjekt registrierte Zielobjekt, den Listener, gesendet. Der Event Handler des Listener bearbeitet dann das Event.

JavaServer Faces kennt zwei Hauptkategorien von Events:

- *Anwendungs-Events* (application events)
- *Lebenszyklus-Events* (lifecycle events)

Anwendungs-Events sind Events, die durch Interaktionen der Benutzer mit der Anwendung ausgelöst werden, wie z.B. die Betätigung einer Schaltfläche. Lebenszyklus-Events sind Events, für deren Auslösung keine direkte Benutzerinteraktion notwendig ist. Diese Events beziehen sich auf die einzelnen Phasen des Lebenszyklus eines Request innerhalb einer JSF-Anwendung. Diese Kategorie unterteilt sich weiter in *Phasen-Events* (phase events) und *System-Events* (system events). Phasen-Events werden durch die Phasen eines Request-Lebenszyklus ausgelöst, wie sie in Kapitel 15 vorgestellt wurden (z.B. Validierung durchführen). System-Events haben eine feinere Granularität. Zu den System-Events gehören die *Komponenten-System-Events* (component system events), die sich auf die Phasen des Lebenszyklus von einzelnen Komponenten beziehen.

Im Folgenden betrachten wir nur die Menge der Anwendungs-Events innerhalb eines JavaServer Faces-Anwendungsprogrammes. Sie stellt die wichtigste Event-Kategorie dar und enthält zwei Event-Typen:

- Events vom Typ: `ActionEvent` (Klasse)
- Events vom Typ: `ValueChangeEvent` (Klasse)

Bei allen Event-Typen des JSF-Frameworks handelt es sich um Java-Klassen, die die entsprechenden Events repräsentieren. Die Klasse `ActionEvent` repräsentiert die Aktivierung einer Komponente der Benutzungsoberfläche (Betätigen einer Befehlskomponente). Die Klasse `ValueChangeEvent` repräsentiert eine Benachrichtigung, dass sich der Wert einer Komponente der Benuzungsoberfläche als Resultat einer Aktion einer Benutzerin geändert hat (z.B. Änderung des Wertes einer Texteingabekomponente).

Um ein Action Event in einer JSF-Anwendung zu verarbeiten, stehen den Entwicklern zwei Möglichkeiten zur Verfügung. Zum einen kann eine eigene Klasse erstellt werden, die einen Listener repräsentiert (durch Implementierung der Schnittstelle `ActionListener`), zum anderen besteht die Möglichkeit lediglich eine sogenannte Aktionsmethode (action method) zu imple-

mentieren, auf die in dem `action`-Attribut einer Befehlskomponente durch einen Ausdruck der Expression Language referenziert wird. In diesem Fall wird das Action Event durch einen vom JSF-Framework zur Verfügung gestellten Listener verarbeitet.

Neben der Angabe einer Ereignisbehandlungsroutine durch das `action`-Attribut ist die Angabe der Routine für das Attribut `actionListener` möglich. Worin besteht der Unterschied zwischen diesen beiden Attributen? Das `action`-Attribut wird dann genutzt, wenn nach dem Aufruf der Methode eine Navigation durchgeführt werden soll. Hierzu kann die aufgerufene Methode Parameter (auch über EL) haben und muss eine Zeichenkette zurückgeben, die im Falle einer impliziten Navigation einen Navigationspfad beschreibt und im Falle einer expliziten Navigation einen in der Datei `faces-config.xml` spezifizierten Navigationsfall auslöst. Das Attribut `actionListener` wird dann verwendet, wenn nach der Ausführung der Methode keine Navigation vorgesehen ist. Die Methode gibt keinen Wert zurück (Rückgabetyp `void`) und hat genau einen Parameter des Typs `ActionEvent`. Eine Entwicklerin hat somit die Möglichkeit mit Hilfe dieses Parameters nähere Informationen über das ausgelöste Ereignis zu erhalten.

Codeausschnitt 19.1 zeigt den Rumpfbereich einer JSF-Seite mit einem Formular und zwei Schaltflächen, eine mit einem `action`-Attribut und eine mit einem `actionListener`-Attribut. Codeausschnitt 19.2 zeigt zwei Methoden zur Verarbeitung der durch die Schaltflächen ausgelösten Events. Auf die Darstellung der gesamten Managed Bean mit dem Namen `MyActionBean` wird hier verzichtet.

In diesem Beispiel wurden die runden Klammern hinter den Methodenbezeichnern ausgelassen. Solange eine Methode keine Argumente erwartet, ist dies möglich. Beim Aufruf einer `actionListener`-Methode, die ein Argument erwartet, wird dieses jedoch implizit durch das JSF-Framework übergeben. Die Angabe von leeren runden Klammern würde das JSF-Framework dazu veranlassen eine Methode mit der Bezeichnung `method2` zu suchen, die über keine Parameter verfügt. Dies wäre hier falsch[1].

```
1 <h:body>
2     <h:form>
3         <h:commandButton value="Senden" action="#{myActionBean.method1}" />
4         <h:commandButton value="Senden" actionListener="#{myActionBean.
              method2}" />
5     </h:form>
6 </h:body>
```

Listing 19.1: Verwendung der Attribute `action` und `actionListener` in einer JSF-Seite

[1] Enthält eine Managed Bean eine Property mit dem Bezeichner Test mit einem entsprechenden Getter und Setter und eine weitere Methode mit dem gleichen Bezeichner Test, die eine Zeichenkette zurückgibt, so wählt das JSF-Framework je nach Komponente den Getter der Property oder den Aufruf der Methode. Für das `action`-Attribut einer Schaltfläche wird bevorzugt die Methode aufgerufen, für das `value`-Attribut einer Texteingabekomponente wird bevorzugt der Getter der Property aufgerufen.

```
1  public String method1()
2  {
3      return "seite2.xhtml";
4  }
5
6  public void method2(ActionEvent event)
7  {
8      // Anweisungen ausführen und ggf. auf den
9      // event-Parameter zugreifen
10 }
```

Listing 19.2: Event handler für ein Action Event

Die Verarbeitung eines Value-Change Event in einer JSF-Anwendung ist mit der Vorgehensweise bei einem Action Event vergleichbar. Ein Value-Change Event wird von Komponenten ausgelöst, die einen Wert aufnehmen können, der sich bei einem Request verändert. Bei einem Texteingabefeld handelt es sich um eine solche Komponente. Hier wird das Attribut `valueChangeListener` verwendet, um mit einem Ausdruck der Expression Language einen Event Handler zu referenzieren (alternativ gibt es die Möglichkeit, eine eigene Klasse zur Ereignisbehandlung zu erstellen).

Ein Value-Change Event wird für eine Komponente nur beim Absenden des übergeordneten Formulars ausgelöst, d.h. es wird nicht immer dann ausgeführt, wenn die Benutzerin den Wert einer Komponente ändert. Die Eingabe eines Zeichens oder einer Ziffer in eine Texteingabekomponente löst z.B. kein Value-Change Event aus, dies erfolgt erst bei der Absendung des Formulars mit Hilfe einer Befehlskomponente.

Codeausschnitt 19.3 zeigt den Rumpfbereich einer JSF-Seite mit einem Formular sowie einer Texteingabekomponente mit zugehörigem Text und einer Schaltfläche. Für die Texteingabekomponente wurde im `valueChangeListener`-Attribut eine Referenz auf den Event Handler `handleChangeEvent` gesetzt. Codeausschnitt 19.4 zeigt die Methode zur Verarbeitung des Value-Change Event der Texteingabekomponente und die Methode, die durch Betätigen der Schaltfläche aufgerufen wird. Auf die Darstellung der gesamten Managed Bean mit dem Namen `MyActionBean` wird verzichtet.

```
1 <h:body>
2     <h:form>
3         <h:outputLabel for="input" />
4         <h:inputText id="input" value="Senden" valueChangeListener="#{
              myActionBean.handleChangeEvent}" />
5         <h:commandButton value="Senden" actionListener="#{myActionBean.
              handleActionEvent}" />
6     </h:form>
7 </h:body>
```

Listing 19.3: Verwendung des Attributes valueChangeListener in einer JSF-Seite

```
1 public void handleChangeEvent(ValueChangeEvent event)
2 {
3     System.out.println("ValueChangeEvent wurde ausgelöst.");
4 }
5
6 public void handleActionEvent(ActionEvent event)
7 {
8     System.out.println("ActionEvent wurde ausgelöst.");
9 }
```

Listing 19.4: Ereignisbehandlungsroutine für ein ValueChange-Event

Diese Seite bleibt aus technischen Gründen frei!

20. Verwendung von Ajax in JSF

Wie bereits in Kapitel 2 der Kurseinheit 2 beschrieben wurde, ist Ajax (Asynchronous JavaScript and XML) eine Technologie bzw. eine Sammlung von Technologien, die eine flüssigere Interaktion zwischen dem Browser und dem Web-Server ermöglicht, indem Daten vom Server asynchron geladen und nur Teile einer Web-Seite aktualisiert werden.

Bereits vor der Version 2.0 von JSF gab es Komponentenbibliotheken und Ajax-Frameworks für JSF-Anwendungen, die für sich genommen gut funktionierten. Eine Spezifikation für eine Ajax-Unterstützung in JSF-Anwendungen gab es jedoch noch nicht. Die vorhandenen Produkte waren oft nicht miteinander kompatibel und basierten teilweise auf sehr unterschiedlichen technischen Umsetzungen.

Seit Version 2.0 beschreibt die Spezifikation von JavaServer Faces einen Standard zur Ajax-Unterstützung. Da Ajax auf der Skriptsprache JavaScript basiert, wurde auch in diesem Standard eine JavaScript-Bibliothek definiert, die grundlegende Ajax-Funktionalitäten zur Verfügung stellt.

Um Ajax in einer JSF-Anwendung zu nutzen, gibt es zwei Möglichkeiten. Zum einen kann eine Entwicklerin die Funktionen der JavaScript-Bibliothek direkt aufrufen, zum anderen kann das `ajax`-Tag der Core-Tag-Bibliothek von JSF genutzt werden, das in diesem Kapitel vorgestellt wird.

Betrachten wir zum Einstieg ein einfaches Beispiel. Codeausschnitt 20.1 zeigt den Rumpfbereich einer JSF-Seite, die eine Textausgabekomponente, eine Texteingabekomponente und eine Schaltfläche beinhaltet. In diesem Beispiel wird auf die Speicherung des eingegebenen Textes in einer Property einer Managed Bean verzichtet. Um auf den eingegebenen Wert zuzugreifen, wird das implizite Objekt `requestScope` der Expression Language verwendet, das in Kapitel 3 der Kurseinheit 3 vorgestellt wurde.

```
1 <h:body>
2     <h:form>
3         <h:outputLabel id="output" value="Der eingegebene Text ist: #{
              requestScope.myValue}" /><br/>
4
5         <h:inputText value="#{requestScope.myValue}" />
6         <h:commandButton value="Senden" />
7     </h:form>
8 </h:body>
```

Listing 20.1: JSF-Seite ohne Ajax-Funktionalität

Wird nun ein Text in die Texteingabekomponente geschrieben und die Schaltfläche betätigt, wird ein Request an den Server gesendet. Im Request Scope wird der eingegebene Wert in der Variablen `myValue` gespeichert. Der Request wird vom Server bearbeitet, eine Response wird erstellt und an den Client gesendet. Diese Response beinhaltet immer noch den Wert der Variablen `myValue`, der als Wert für die Textausgabekomponente (`outputLabel`) übernommen wird. Da dieses Beispiel keine Ajax-Funktionalität nutzt, wurde die gesamte Seite neu geladen. Bei einer Seite mit sehr wenig Inhalt mag dies kein Problem darstellen. Bei größeren und komplexeren Seiten sind Ladezeiten aber erheblich länger, und durch ein ständiges Neuladen der gesamten Seite nimmt die Benutzerfreundlichkeit ab.

Codeausschnitt 20.2 zeigt das Beispiel von oben mit Ajax-Funktionalität. Um zu verdeutlichen, dass die Seite nicht neugeladen wird bzw. keine Navigation vorgenommen wird, wurde die Schaltfläche entfernt. Der Texteingabekomponente wurde das `ajax`-Tag zugewiesen. Der Wert des Attributes `render` verweist auf die ID der Textausgabekomponente und veranlasst, dass diese aktualisiert wird. Die Aktualisierung wird dann ausgeführt, wenn der Fokus der Texteingabekomponente verlassen wird, d.h. wenn z.B. ein Mausklick in einem Bereich außerhalb der Komponente durchgeführt wird. Es wird dann nicht die gesamte Seite neu geladen, sondern nur die Textausgabekomponente aktualisiert.

```
1 <h:body>
2     <h:form>
3         <h:outputLabel id="output" value="Der eingegebene Text ist: #{
              requestScope.myValue}" /><br/>
4
5         <h:inputText value="#{requestScope.myValue}">
6             <f:ajax render="output" />
7         </h:inputText>
8     </h:form>
9 </h:body>
```

Listing 20.2: JSF-Seite mit Ajax-Funktionalität

20.1. Das Attribut ajax

In diesem Abschnitt werden das Attribut ajax und seine Optionen näher betrachtet.

Das `ajax`-Tag kann auf eine oder auf mehrere Komponenten angewendet werden und ihnen so Ajax-Funktionalität zuweisen. Dafür kann es entweder in eine Komponente eingebettet werden:

```
<Komponente>
   <f:ajax />
</Komponente>
```

oder eine oder mehrere Komponenten in sich einbetten:

```
<f:ajax>
   <Komponente1 />
```

```
   <Komponente2 />
   <Komponente3 />
</f:ajax>
```

Die Kombination von umgebenden `ajax`-Tags und eingebetteten `ajax`-Tags wird dann angewendet, wenn die Attribute der Tags für spezifische Nutzungen angepasst werden müssen. Im folgenden Ausschnitt werden zwei `ajax`-Tags genutzt (umgebend und eingebettet):

```
<f:ajax attribut="Mausbewegung">
   <Komponente1>
      <f:ajax attribut="Mausklick"/>
   </Komponente1>
   <Komponente2 />
</f:ajax>
```

Durch diese Kombination wird der `Komponente2` lediglich Ajax-Verhalten für eine Mausbewegung hinzugefügt, während Komponente1 über Ajax-Verhalten für eine Mausbewegung und einen Mausklick verfügt. Die Attribute werden kombiniert.

Die Attribute, die für das `ajax`-Tag angegeben werden können, gehören zu zwei Gruppen. Zum einen gibt es Attribute, die die Server-seitige Verarbeitung konfigurieren. Zum anderen gibt es Attribute, die die Client-seitige Verarbeitung konfigurieren. Alle Attribute sind optional und müssen somit nicht angegeben werden.

Tabelle 20.1 gibt eine Übersicht über die wichtigsten Attribute.

`execute`	Für dieses Attribut kann eine Liste von IDs der Komponenten (getrennt durch Leerzeichen) angegeben werden, die in einen Ajax-Request einbezogen werden sollen. Alternativ können die folgenden Schlüsselwörter angegeben werden: • @all : Alle Komponenten werden mit einbezogen • @none : Keine Komponenten werden mit einbezogen • @this : Die Komponente, die den Ajax-Request ausgelöst hat, wird mit einbezogen • @form : Alle Komponenten des Formulars, aus dem der Ajax-Request ausgelöst wird, werden mit einbezogen Wird das Attribut nicht angegeben, wird nur der Wert der Komponente übertragen, auf den das `ajax`-Tag angewendet wurde.
`render`	Für dieses Attribut kann eine Liste von IDs der Komponenten (getrennt durch Leerzeichen) angegeben werden, die nach einem Ajax-Request aktualisiert werden sollen. Alternativ können die gleichen Schlüsselwörter wie bei `execute` angegeben werden, bezogen auf die Aktualisierung von Komponenten.
`listener`	Der Wert dieses Attributes gibt den Bezeichner einer Ereignisbehandlungsroutine an, die aufgerufen wird, wenn ein Ajax-Verhaltensereignis eintritt, z.B. ein Mausklick oder die Aktivierung einer Kontrollbox.
`disabled`	Durch den Wert dieses Tags kann die Ajax-Funktionalität des `ajax`-Tags deaktiviert (`true`) oder aktiviert werden (`false`). Standardmäßig ist der Wert auf false gesetzt. Eine JSF-Anwendung kann so z.B. die Ajax-Funktionalität abschalten, wenn ein Browser kein JavaScript unterstützt oder JavaScript deaktiviert hat.
`event`	Dieses Tag spezifiziert das Ereignis, durch das ein Ajax-Request ausgelöst wird. Die Menge der möglichen Ereignisse ist durch die Ereignisse gegeben, die JavaScript verarbeiten kann, z.B. `click`, `mousedown`, `keydown`, `mousemove`, `scroll` usw. Wird das `event`-Attribut nicht angegeben, wird ein Standard-Event für die jeweilige Komponente ausgeführt. Für Befehlskomponenten ist dies ein Action-Event, für Komponenten, in denen ein Wert eingegeben oder ausgewählt wird (Text, Listen, Kontrollbox, usw.) ist dies ein ValueChange-Event.
`onevent`	Der Wert dieses Attributes gibt den Namen einer JavaScript-Funktion an, die bei dem ausgeführten Ajax-Request auf dem Client aufgerufen wird.
`onerror`	Der Wert dieses Attributes gibt den Namen einer JavaScript-Funktion an, die bei dem ausgeführten Ajax-Request auf dem Client aufgerufen wird, sollte während des Requests ein Fehler auftreten.

Tabelle 20.1.: Die wichtigsten Ajax-Attribute

Codeausschnitt 20.3 zeigt den gesamten Inhalt einer JSF-Seite, in der das `ajax`-Tag mit allen in Tabelle 20.1 vorgestellten Attributen genutzt wird.

```
<?xml version='1.0' encoding='UTF-8' ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
          "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core">
    <h:head>
        <title>Facelet Title</title>
        <script type="text/javascript">
            function jProcessEvent(data) {
                if (data.status === "begin") {
                    alert("JavaScript-Funktion aufgerufen.");
                }
            }
            function jRequestError(data) {
                if (data.status === "begin") {
                    alert("Fehler im Ajax-Request aufgetreten.");
                }
            }
        </script>
    </h:head>
    <h:body>
        <h:form>
            <h:panelGrid columns="2">
               <h:outputLabel for="forename" value="Vorname: " />
               <h:inputText id="forename" value="#{myAjaxBean.forename}" />
               <h:outputLabel for="surname" value="Nachname: " />
               <h:inputText id="surname" value="#{myAjaxBean.surname}" />
               <h:outputLabel value="Name: " />
               <h:outputLabel id="name" value="#{myAjaxBean.forename} #{
                  myAjaxBean.surname}" />
            </h:panelGrid>

            <h:commandButton id="submit" value="Senden" >
                <f:ajax
                    execute="forename surname"
                    render="name"
                    listener="#{myAjaxBean.processAction}"
                    disabled="false"
                    event="mouseover"
                    onevent="jProcessEvent"
                    onerror="jRequestError">
                </f:ajax>
            </h:commandButton>
        </h:form>
    </h:body>
</html>
```

Listing 20.3: Nutzung des Ajax-Tags mit allen wichtigen Attributen

Der Rumpf der Seite besteht aus einem zweispaltigen Panel-Grid, das unter anderem Texteingabekomponenten für den Vor- und Nachnamen einer Benutzerin enthält, sowie eine Textausgabekomponente, in der der vollständige Name angezeigt werden soll. Im Hintergrund der Seite liegt eine Managed Bean mit dem Namen `MyAjaxBean`, die den Vornamen und den Nachnamen speichert sowie eine Methode `processAction` zur Verfügung stellt. Unterhalb des Panel-Grid wird eine Schaltfläche ohne `action`-Attribut definiert. In die Schaltflächenkomponente ist das `ajax`-Tag eingebettet. Das `execute`-Attribut beinhaltet die IDs beider Texteingabekomponenten, sodass die eingegebenen Werte innerhalb des Ajax-Request verarbeitet werden können. Für das `render`-Attribut wird die ID der Textausgabekomponente angegeben, die den gesamten Namen anzeigen soll. Dies ist notwendig, damit diese Komponente durch den Ajax-Request aktualisiert wird. Der Wert des `listener`-Attributes ist ein Ausdruck der Expression Language, der auf die Methode `processAction` der Managed Bean verweist. Das Attribut `disabled` wird auf *false* gesetzt, damit das `ajax`-Tag aktiv ist. Das Attribut `event` legt durch den Wert `mouseover` fest, dass der Ajax-Request dann ausgeführt wird, wenn die Maus in die Schaltfläche hinein bewegt wird, ein Mausklick auf die Schaltfläche ist nicht notwendig. Das `onevent`-Attribut verweist auf die JavaScript-Methode `jProcessEvent` und das `onerror`-Attribut auf die JavaScript-Methode `jRequestError`. Beide JavaScript-Methoden werden in dem JavaScript-Block im Kopfbereich der Seite definiert und geben lediglich eine Informationsnachricht an die Benutzerin aus (diese konkrete Nachricht würde eine tatsächliche Benutzerin natürlich kaum interessieren). Die in einem `ajax`-Tag angegebene JavaScript-Methode für das Attribut `onevent` wird bei einem Ajax-Request insgesamt dreimal ausgeführt, einmal bevor der Request ausgeführt wird, einmal nachdem der Request ausgeführt wurde und einmal, wenn alle HTML-Elemente aktualisiert wurden. Die aktuelle Phase wird in der Variable `data` gespeichert. Durch einen Zeichenkettenvergleich mit `begin`, `complete` und `success` können Operationen für jede Phase individuell ausgeführt werden.

Kurseinheit 6
Java EE: Enterprise JavaBeans

Nachdem in den Kurseinheiten 3 und 5 Technologien für die Realisierung der Web-Schicht behandelt wurden, widmen sich die letzten beiden Kurseinheiten der Umsetzung des Anwendungskerns, der aus Anwendungslogik und (persistenten) Anwendungsobjekten besteht.

Diese Kurseinheit beginnt mit einer Einführung in die Enterprise JavaBeans Technologie, die von der Java EE-Spezifikation für die Realisierung von Anwendungslogik vorgesehen ist. Im Gegensatz zu reinen Java-Klassen werden Enterprise JavaBeans in einem Java EE-Container ausgeführt und können von diesem eine Fülle von Leistungen in Anspruch nehmen.

Diese Seite bleibt aus technischen Gründen frei!

21. Einstieg in Enterprise JavaBeans und Entities

21.1. Einstieg

Nachdem in den Kurseinheiten 3 und 5 Technologien für die Realisierung der *Web-Schicht* der 5-Schichten-Architektur für Web-Anwendungen, bzw. der *View* und des *Controller* der Model-2-Architektur, behandelt wurden, widmen sich die vorliegende und die nächste Kurseinheit Technologien zur Umsetzung des *Anwendungskerns* (*Model*), der aus *Anwendungslogik* und (persistenten) *Anwendungsobjekten* besteht.

Diese Kurseinheit behandelt die *Enterprise JavaBeans Technologie* (EJB), die von der Java EE-Spezifikation für die Realisierung von Anwendungslogik vorgesehen ist. Wir beziehen uns dabei auf die EJB-Spezifikation [EJB/Spec] in Version 3.2. Gegenüber reinen Java-Klassen, wie z.B. den in [SE I] verwendeten Kontrollklassen, werden Instanzen von EJB-Klassen in einem Container des Java EE-Servers ausgeführt (s. Abbildung 21.1) und können von diesem eine Fülle von Leistungen in Anspruch nehmen, z.B. Transaktions- und Sicherheitsmanagement oder Lastausgleich in Netzwerken. Wenn aus dem Kontext hervorgeht, ob es sich um eine Instanz einer EJB-Klasse oder um die EJB-Klasse an sich handelt, bezeichnen wir Instanz bzw. Klasse abkürzend als EJB.

In Kurseinheit 7 folgt eine Einführung in die so genannten *Entities* als Realisierung von (persistenten) Anwendungsobjekten. Entities sind vergleichbar mit Instanzen von Entitätsklassen aus [SE I]. Ihre Persistierung[1] erfolgt über die *Java Persistence API*, die in [JPA/Spec] spezifiziert wird. Die Java Persistence API stellt eine vom EJB-Container (s. Abb. 21.1) implementierte Schnittstelle dar, die von einem konkreten Persistenzframework wie z.B. Hibernate [Hiber] abstrahiert, und damit auch vom verwendeten Datenbanksystem wie z.B. MySQL [MySQL]. Ein Persistenzframework kapselt den Zugriff auf ein Datenbanksystem, bildet Klassen auf Datenbanktabellen ab und ermöglicht so das Speichern und Laden von Objektzuständen. Das über die Persistence API angebundene Persistenzframework wird in [EJB/Spec] als *Persistence Provider* bezeichnet.

[1] Unter Persistierung versteht man den Vorgang, Persistenz von Daten herzustellen. Persistenz bezeichnet die Eigenschaft, das Ende der Laufzeit eines Programmes oder Systems zu überdauern, persistente Daten bleiben also auch nach Programmende gespeichert.

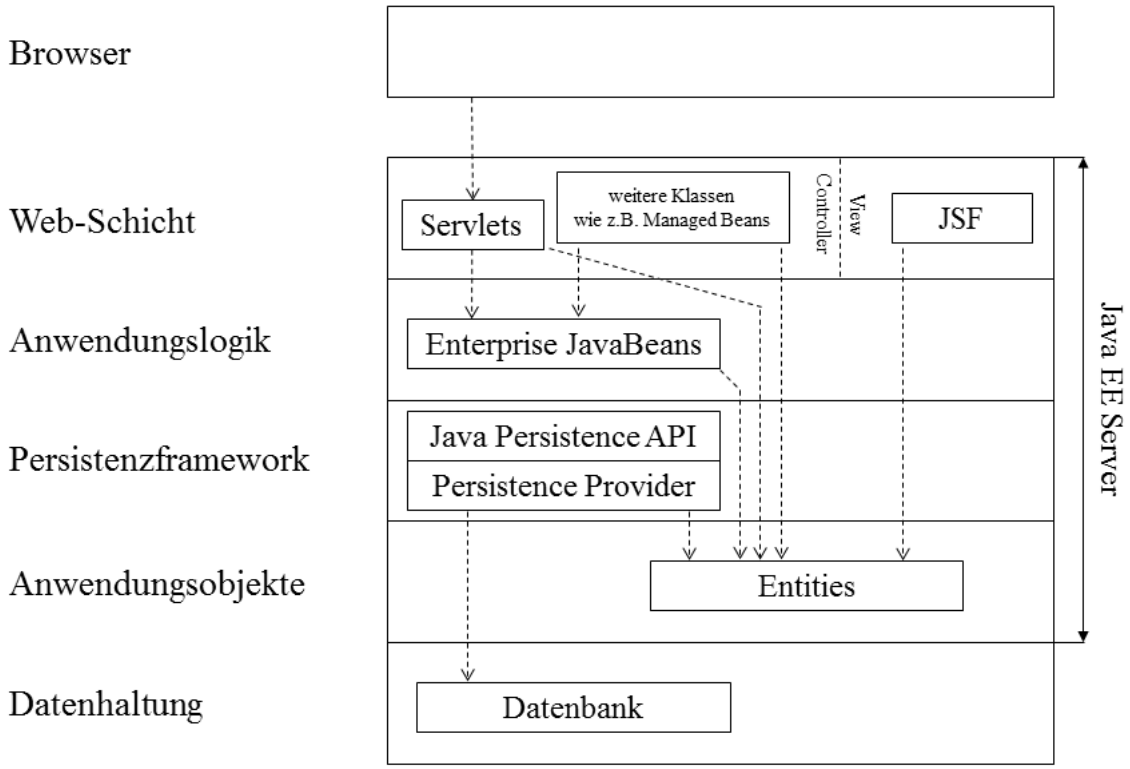

Abbildung 21.1.: 6-Schichten-Architektur mit Java EE und JSF

In diesem Kurs werden EJBs und Entities lediglich im Kontext von Web-Anwendungen betrachtet, sie können aber auch für die Realisierung des Anwendungskerns anderer (Client-Server-)Anwendungen verwendet werden.

Zur Übersicht ordnen wir in Abbildung 21.1 die bereits behandelten sowie in dieser Kurseinheit neu eingeführten Komponenten einer Java EE-Anwendung in eine Schichtenarchitektur ein. Zur Einordnung der Java Persistence API und ihres Persistence Providers fügen wir jedoch eine zusätzliche, sechste Schicht hinzu. Um eine weitergehende Trennung von Verantwortlichkeiten zu erreichen, unterteilen wir außerdem die Web-Schicht in die Teilkomponenten View und Controller (gemäß dem MVC-Muster).

Web-Schicht und Anwendungskern samt Persistenzframework werden im Web- bzw. EJB-Container des Java EE-Application Server ausgeführt. Das vom Persistenzframework verwendete Datenbanksystem gehört logisch nicht zum Application Server, jedoch werden Application Server häufig zusammen mit einem geeigneten Datenbanksystem gebündelt ausgeliefert. Zum Beispiel liegt dem Oracle GlashFish Application Server das Datenbanksystm *Java DB* [JDB] bei, das auf *Apache Derby* [Derby] basiert[2].

Entities werden in der nächsten Kurseinheit beschrieben. Als Erläuterung der Beziehungen aus Abb. 21.1 sei jedoch vorweggenommen, dass nur das Persistenzframework (auf Veranlassung

[2] Apache Derby ist ein Java-basiertes, relationales Datenbank-Management-System

der Anwendungslogik) Entities in die Datenbank schreiben oder aus der Datenbank lesen kann. Komponenten der Web-Schicht können lediglich neue Entites instanziieren und zur Persistierung an die Anwendungslogik weiterleiten oder von der Anwendungslogik transiente[3] Kopien persistenter Entities[4] z.B. zur Präsentation übergeben bekommen.

Den Abschluss des Kurstextes bildet eine Vorstellung ausgewählter Entwurfsmuster, insbesondere für eine sinnvolle Anbindung des Anwendungskerns an die Web-Schicht.

21.2. Annotationen für Metadaten

Annotationen wurden bereits in Kurseinheit 3 vorgestellt. Hier sollen die Auswirkungen von Annotationen noch einmal kurz angesprochen und ein neuer Begriff eingeführt werden.

Annotationen haben keinen direkten Einfluss auf den Code, in dem sie eingefügt sind. Sie enthalten Informationen, die in der Regel für einen gewissen Adressaten bestimmt sind. Die Auswirkungen der Annotationen hängen ab von der Reaktion des Adressaten auf diese Informationen und können dementsprechend sehr unterschiedlich sein.

Die in Codeausschnitt 1.4 der Kurseinheit 3 vorgestellte Override-Annotation ist ein Beispiel für eine Annotation, die keinerlei Auswirkung auf das Verhalten der Implementierung hat. Ihr Adressat ist der Compiler, der durch die Annotation zusätzliche Informationen bekommt, um erweiterte Gültigkeitsprüfungen zu ermöglichen. Zudem können Annotationen helfen, Programmierfehler aufzudecken.

Wie nützlich die Override-Annotation trotz ihrer scheinbaren Wirkungslosigkeit sein kann, sei an einem Beispiel erläutert: Angenommen, man möchte die `doGet`-Methode eines Servlet implementieren und vertippt sich versehentlich bei ihrer Signatur (nennt sie z.B. `doget` oder vergisst einen Parameter), so wird hierdurch eine neue Methode neben der geerbten eingeführt. Ohne Annotation wäre die Servlet-Klasse trotz dieses Tippfehlers problemlos compilierbar. Erst beim Zugriffsversuch zur Laufzeit würde der Webserver melden, dass das Servlet keine GET-Requests unterstützt. Annotiert man dagegen die Methode mit `@Override`, so wird der Compiler sofort bei der Übersetzung feststellen und melden, dass man nicht, wie beabsichtigt, eine geerbte Methode überschreibt. Das Servlet ist dann nicht compilierbar, bis der Fehler behoben wurde. Wir werden daher die Override-Annotation auch in den folgenden Fallbeispielen einsetzen.

[3] transient: flüchtig, vorübergehend
[4] Vgl. Begriffsdefinitionen in Abschnitt 23.1

```
1  import javax.ejb.EJB;
2  import ...
3
4  public class MeinServlet extends HttpServlet {
5
6     @EJB private MeineEJB bean;
7
8     protected void doGet(HttpServletRequest request, HttpServletResponse
          response) throws ServletException, IOException {
9        bean.meineAnwendungslogik();
10       ...
11    }
12 }
```

Listing 21.1: Dependency Injection mittels EJB-Annotation

Eine weitere Anwendung für Annotationen wird in Codeausschnitt 21.1 demonstriert. In diesem Beispiel möchte ein Servlet die Anwendungslogik einer EJB nutzen. Es besitzt ein Attribut `bean` vom gewünschten EJB-Typ und deklariert seine Abhängigkeit von einer EJB dieses Typs durch Annotation des Attributs mit `@EJB`. Der Web-Container durchsucht Servlets bei deren Deployment nach derartigen Annotationen und sorgt in diesem Beispiel zur Laufzeit dafür, dass im Attribut `bean` rechtzeitig, d.h. bevor im Code des Servlet darauf zugegriffen wird, eine Referenz auf eine passende EJB-Instanz abgelegt wird. Man sagt, dass der Container die benötigte Referenz in die Variable *injiziert*.

Es gibt neben `javax.ejb.EJB` noch verschiedene weitere Annotationstypen zur Deklaration gegenseitiger Abhängigkeit (Dependency) von Komponenten oder Ressourcen. Die damit angeforderte Injektion zur Laufzeit durch den Container wird als *Dependency Injection* oder *Resource Injection* bezeichnet.

22. Enterprise JavaBeans

22.1. Einführung

Mit Enterprise JavaBeans stellt Java EE eine leistungsfähige Technologie zur Realisierung von Anwendungslogik bereit.

Eine Enterprise JavaBean ist eine spezielle Java-Klasse, deren Methoden Anwendungslogik implementieren. Dank der mit EJB 3.0 eingeführten Simplified API soll dabei die Implementierung von Anwendungslogik in EJBs gegenüber der Implementierung mit simplen Java-Klassen (wie mit den in [SE I] verwendeten Kontrollklassen) kaum schwieriger oder aufwendiger sein, wobei die EJBs jedoch unter anderem folgende zusätzliche Möglichkeiten bieten:

- wahlweise synchroner oder asynchroner Zugriff
- Client-spezifischer Zustand
- Verteilung mit transparentem[1] Remote-Zugriff und Lastausgleich
- konfigurierbares automatisches Transaktionsmanagement
- Sicherheitsmanagement

Anwendungsentwickler sollen sich auf die Implementierung der Anwendungslogik konzentrieren, während oben genannte Dienste vom EJB-Container übernommen werden.

Im Folgenden gehen wir auf die oben genannten Punkte näher ein und stellen dabei die verschiedenen Arten von EJBs vor.

22.1.1. Synchroner und asynchroner Zugriff: Session Bean und Message-Driven Bean

Es gibt zwei Arten von EJBs: Session Beans und Message-Driven Beans.

Vor der EJB Version 3.1 unterstützten Session Beans ausschließlich synchrone Methodenaufrufe: der aufrufende Thread[2] wartet bis zur Abarbeitung der aufgerufenen Methode und bekommt nach deren Terminierung ggf. den Rückgabewert übermittelt. Seit der EJB Version 3.1 unterstützen

[1] Als transparent wird in einem Computersystem ein Teil genannt, der vorhanden und in Betrieb ist, von Benutzern jedoch nicht wahrgenommen wird.

[2] Ein Thread (eng. Faden, Thema) bezeichnet in der Informatik einen Ausführungsstrang innerhalb eines Programms.

Session Beans auch asynchrone Methodenaufrufe: der Aufrufer muss nicht warten, bis die Session Bean die Methode abgearbeitet hat, sondern erhält direkt die Ablaufkontrolle zurück.

Message-Driven Beans unterstützen ausschließlich asynchrone Methodenaufrufe und werden stets bei einer *Nachrichtenwarteschlange* angemeldet. Über einen vom Application Server bereitgestellten *Nachrichtendienst* kann ein Client so Nachrichten an die Message-Driven Bean senden, welche darauf mit der Ausführung von Anwendungslogik reagiert. Diese Reaktion erfolgt asynchron, d.h. der Aufrufer sendet lediglich seine Nachricht und arbeitet dann parallel weiter, ohne auf eine Rückmeldung der Bean zu warten.

Asynchrone Aufrufe sind insbesondere für langwierige Abläufe sinnvoll, während derer der aufrufende Thread nicht blockiert bleiben soll. Bei Web-Anwendungen kann so z.B. ein langes Warten des Browsers auf die Response vermieden werden.

22.1.2. Clientspezifischer Zustand: Stateful und Stateless Session Bean

Es gibt zwei Varianten von Session Beans: Stateless und Stateful Session Beans.

Eine Instanz einer *Stateless Session Bean* wird einem Client lediglich zur Ausführung einer Methode zugeordnet. Nach der Abarbeitung der Methode wird diese Zuordnung zum Client wieder gelöst: die Instanz verbleibt in einem Pool[3], bis sie wieder zur Bearbeitung der Anfrage irgendeines Clients benötigt oder zur Freigabe von Ressourcen zerstört wird. Eine solche Stateless Session Bean könnte zwar, wie jedes Java-Objekt, einen Zustand haben. Dies ist jedoch in der Regel nicht empfehlenswert, da mehrere parallel existierende Instanzen einer sonst derartigen Session Bean unterschiedliche Zustände annehmen könnten, ohne fest einem Client zugeordnet zu sein. Folglich wird man in einer Stateless Session Bean im Normalfall lediglich Methoden, jedoch keine Attribute implementieren.

Eine Instanz einer *Stateful Session Bean* ist dagegen immer exklusiv einem bestimmten Client zugeordnet: für jeden neuen Client wird eine neue Instanz erzeugt und bleibt bis zu ihrer Zerstörung diesem Client zugeordnet. Falls die Bean-Klasse Attribute besitzt, ihre Instanzen also einen Zustand haben, so ist auf diese Weise sichergestellt, dass zu jedem Client ein spezifischer Zustand, auch als *Conversational State* bezeichnet, nachgehalten wird.

Eine Session Bean sollte nur dann als *stateful* realisiert werden, wenn das Speichern eines Clientspezifischen Zustands auch tatsächlich benötigt wird. Beans ohne Zustand sollten dagegen stets *stateless* sein, da durch die Wiederverwendung der Instanzen für mehrere Clients deutliche Ressourcen-Einsparungen erzielt werden können. Das Pooling[4] von gerade nicht benötigten Instanzen von Stateless Beans reduziert darüber hinaus den ansonsten durch Zerstören und Neu-Erzeugen von Instanzen verursachten Aufwand, was sich positiv auf die Performanz auswirkt.

[3] Ein Pool bezeichnet eine Menge von Ressourcen (z.B. Instanzen von Klassen), die jederzeit einsatzbereit sind und bei Bedarf verwendet werden können. Nach der Verwendung werden sie nicht zerstört, sondern in den Pool zurückgelegt.

[4] Die Verwendung eines Pool wird als Pooling bezeichnet.

Eine Message-Driven Bean ist grundsätzlich stateless: Alle Nachrichten in der Warteschlange werden an die angemeldeten Instanzen der Bean-Klasse verteilt, dabei findet keine Client-spezifische Zuordnung statt.

22.1.3. Verteilung mit transparentem Remote-Zugriff und Lastausgleich – Remote- und Local Interfaces für Session Beans

EJBs müssen nicht alle in demselben EJB-Container auf demselben Server betrieben werden, sondern können auf mehrere Server verteilt sein. Damit können verschiedene Ziele verfolgt werden, z.B. redundante Verteilung auf mehrere Server zur Erhöhung der Ausfallsicherheit und/oder zum Verteilen der Rechenlast (Lastausgleich, Load Balancing). Dabei regelt der EJB-Container die transparente Zuteilung von Anfragen der Clients an Instanzen passender EJBs auf erreichbaren, möglichst niedrig ausgelasteten Servern. Ein Client fordert lediglich eine EJB-Instanz an, deren Anwendungslogik er anschließend ausführt (bzw. er sendet eine Nachricht). Und selbst wenn (zunächst) die EJBs nur auf *einem* Server installiert werden, so ist das System skalierbar durch die Möglichkeit, im Falle steigender Auslastung nachträglich eine Verteilung vorzunehmen.

Message-Driven Beans eignen sich also grundsätzlich zur Verteilung, bei Session Beans dagegen wird zwischen Remote-Zugriff[5] und lokalem Zugriff unterschieden: Eine Session Bean musste vor der EJB Version 3.1 ihre Routinen für die Anwendungslogik grundsätzlich über eine Schnittstelle, das so genannte *Business Interface*, anbieten. Das heißt, jede Session Bean musste zwangsläufig entweder ein *Local Interface* (für einen lokalen Zugriff) oder ein *Remote Interface* (für einen Remote-Zugriff) implementieren und dadurch die Anwendungslogik-Methoden exportieren.

Spricht ein Client eine Session Bean über ein Remote Interface an, wird der Aufruf über das Netzwerk gesendet. Dies hat den Vorteil, dass der oben beschriebene Remote-Zugriff auf Beans ermöglicht wird, die auf anderen Servern installiert sind. Nachteilig ist, dass dazu alle Parameter *serialisiert* werden müssen, d.h. es dürfen ausschließlich serialisierbare Typen verwendet werden, und die Parameter werden grundsätzlich als Kopie übergeben („pass-by-value“). Referenzübergaben („pass-by-reference“) sind über Remote Interfaces dagegen nicht möglich. Durch die Serialisierung fällt ein gewisser Aufwand an, der sich nur lohnt, wenn die Bean tatsächlich auf einem anderen Server liegt.

Ist eine Bean dagegen direkt auf demselben Application Server installiert wie der sie benutzende Client, so ist ein lokaler Aufruf effizienter. Dann verläuft die Kommunikation nicht über das Netzwerk, sondern ein herkömmlicher Java-Methodenaufruf findet statt. Vor der EJB Version 3.1 war ein lokaler Aufruf einer Methode einer EJB nur möglich, indem der Client die EJB über ein Local Interface ansprach. Seit der EJB Version 3.1 wird für lokale Aufrufe kein Local Interface mehr zwingend benötigt, hier spricht man von einer *No-Interface view.* Dem Client

[5] engl. remote = fern, entfernt

stehen alle öffentlichen Methoden der Bean zur Verfügung, und die Bean kann direkt, anstatt über ein Interface, angesprochen bzw. referenziert werden.

Eine Session Bean kann sowohl ein Local Interface als auch ein Remote Interface implementieren und die No-Interface view unterstützen, sodass alle drei Zugriffsmöglichkeiten zugleich bestehen. Implementiert die Bean ausschließlich ein Local Interface, oder wird die No-Interface view verwendet, so ist es für die oben angesprochenen Skalierbarkeit ratsam, keine Spezifika des lokalen Aufrufs (wie z.B. Referenzübergaben oder die Möglichkeit, nicht serialisierbare Typen für Parameter zu verwenden) auszunutzen. Im Falle nachträglicher Verteilung kann dann problemlos auf ein Remote Interface ohne Modifikation der implementierten Anwendungslogik umgestellt werden.

22.1.4. Transaktionsmanagement

Unter einer Transaktion versteht man in der Informatik eine Folge von Ausführungsschritten, die eine logische Einheit bilden. Möchte z.B. eine Kundin ein Brot in einer Bäckerei kaufen, so erhält sie das Brot als Gegenleistung zu einem gezahlten Geldbetrag. Die Bezahlung und die Übergabe des Brotes als Ausführungsschritte bilden eine logische Einheit. Eine Bezahlung ohne den Erhalt des Brotes ist für die Kundin nicht zufriedenstellend. Insbesondere bei Operationen auf Datenbanken haben Transaktionen eine große Bedeutung.

Ein System, das Transaktionen durchführen kann, wird Transaktionssystem genannt. Eine Transaktion muss die sogenannten ACID-Eigenschaften erfüllen.

- A - Atomicity (Atomarität)
  Eine Folge von Ausführungsschritten wird als atomar bezeichnet, wenn sie nicht in Teilfolgen unterteilbar ist. Eine Transaktion wird entweder ganz ausgeführt, oder sie wird nicht ausgeführt.
- C - Consistency (Konsistenz)
  Wird eine Transaktion ausgeführt, so muss der Datenbestand nach der Ausführung konsistent sein, sofern er vor der Ausführung der Transaktion bereits konsistent war. Es dürfen somit durch die Transaktion keine in sich fehlerhaften Daten eingefügt werden, wie zum Beispiel unauflösbare Referenzen.
- I - Isolation (Isolation)
  Transaktionen müssen voneinander isoliert sein. Das bedeutet, dass zwei Transaktionen, die nebenläufig ausgeführt werden, sich gegenseitig nicht beeinflussen.
- D - Durability (Langlebigkeit, Persistenz)
  Daten müssen nach der erfolgreichen Durchführung einer Transaktion dauerhaft gespeichert sein. Fällt ein Computersystem während oder nach einer Transaktion aus, so ist evtl. nur ein Teil der Schreiboperationen in einer Datenbank ausgeführt worden. Durch ein sogenanntes Transaktionslog kann gespeichert werden, welche Schreiboperationen bereits durchgeführt wurden und welche noch durchgeführt werden müssen.

Der EJB-Container bietet ein automatisches Transaktionsmanagement. Auch um das Starten und Beenden von Transaktionen müssen sich Entwickler im Regelfall nicht kümmern. Das übernimmt vielmehr der Container, indem er automatisch (für den Client transparent) eine Transaktion startet, sobald eine EJB-Methode von einem Client aufgerufen wird, und beim Beenden der Methode die Transaktion wieder beendet. Das Transaktionsmanagement ist (über Annotationen oder Deployment-Deskriptoren) konfigurierbar, sodass vom Standardverhalten abgewichen werden kann.

Sollte dieses konfigurierbare *implizite Transaktionsmanagement* einmal nicht flexibel genug sein, besteht auch die Möglichkeit, auf ein *explizites Transaktionsmanagement* zurückzugreifen, bei dem Transaktionen durch Entwickler über entsprechende Anweisungen gesteuert werden können.

Auf das Transaktionsmanagement werden wir später im Kontext der Verwendung persistenter Anwendungsobjekte durch EJBs noch weiter eingehen.

22.1.5. Sicherheitsmanagement

Sicherheit spielt bei Web-Anwendungen eine große Rolle. Angefangen bei einem Anmeldeformular, über verschiedene Rechte und Rollen für Benutzer, bis hin zu Konzepten zur Verhinderung von Angriffen auf die Web-Anwendung oder den Server, ist es stets die Aufgabe von Entwicklern, entsprechende Sicherheitskonzepte in ihre Anwendungesprogramme zu integrieren.

Java EE bietet ein umfangreiches Konzept zur Absicherung von EJBs und allgemein von Web-Anwendungen. Analog zum Transaktionsmanagement gibt es die Möglichkeit, das Sicherheitsmanagement weitestgehend dem Container zu überlassen und es lediglich deklarativ durch Annotationen oder Deployment-Deskriptoren zu konfigurieren. So kann z.B. durch Annotationen festgelegt werden, wer Zugriffsrechte für eine EJB oder eine bestimmte Methode haben soll. Wo das nicht ausreicht, gibt es auch hier weitergehende Möglichkeiten, in der Anwendungslogik selbst steuernd einzugreifen.

Das Sicherheitsmanagement ist ein wichtiges und umfangreiches Thema, das hier aber nicht vertieft behandelt wird.

22.1.6. Rollenverteilung der Java EE-Plattform (Platform Roles)

Die Java EE-Spezifikation [JEE/Spec] unterscheidet verschiedene so genannte Platform Roles, Rollen im Prozess der Entwicklung der Plattform selbst, der Werkzeugentwicklung, der Anwendungsentwicklung für die Plattform und für den Anwendungsbetrieb. Tabelle 22.1 stellt die für die Anwendungsentwicklung und den Anwendungsbetrieb zuständigen Rollen knapp zusammen (Details finden sich in [JEE/Spec], Abschnitt 2.11). Es können mehrere dieser Rollen von derselben Person eingenommen werden.

Rolle	Aufgaben
Enterprise Bean Provider	Erstellung einer oder mehrerer *Java Enterprise Beans*. Bereitstellung der Enterprise Java Beans für den Application Assembler.
Application Assembler	Zusammenstellung einer *Enterprise Application* aus einzelnen Anwendungskomponenten. Bereitstellung für den Deployer.
Deployer	Installation von Web-Anwendungen und EJBs in einem Container inkl. der Konfiguration.
System Administrator	Einrichtung, Betrieb und Überwachung des Systems.

Tabelle 22.1.: Java EE Platform Roles (Auszug)

Im Folgenden werden wir nur dann auf diese Rollen Bezug nehmen, wenn es um die Abgrenzung von Verantwortlichkeiten geht.

22.1.7. Annotationen vs. Deployment-Deskriptoren

Wie bereits in Kurseinheit 3 im Zusammenhang mit Servlets erläutert, lassen sich gewisse Einstellungen von Java EE Web-Anwendungen wahlweise durch Annotationen im Quelltext oder mit Hilfe von Deployment-Deskriptoren in Form von XML-Dateien vornehmen. Ähnliches gilt für die Konfiguration von Java Enterprise Beans. Die Verwendung der Annotationen ist dabei einfacher und komfortabler, da die Meta-Informationen direkt an die Programmtext-Teile geheftet werden, auf die sie sich beziehen, und nicht in eine separat zu erstellende Datei ausgelagert werden, in der noch explizit der Bezug zum Code hergestellt werden muss. Durch die Gültigkeitsprüfungen des Compilers ist die Verwendung der Annotationen darüberhinaus weniger fehleranfällig.

Andererseits wird die Alternative der Deployment-Deskriptoren nicht allein aus Abwärtskompatibilitätsgründen angeboten, sondern kann durchaus auch Vorteile aufweisen: Im folgenden Abschnitt erklären wir, wie man z.B. ein Business Interface durch eine Annotation als Local oder Remote Interface markiert. Möchte man den Interface-Typ durch Austausch der Annotation nachträglich ändern, ist das Programm neu zu kompilieren. Nimmt man diese Zuordnung jedoch in einer XML-Datei extern vor, so kann sie jederzeit nachträglich geändert werden, ohne dass neu kompiliert werden muss.

Die Java EE-Spezifikation legt fest, dass falls eine Meta-Information sowohl als Annotation als auch in einem Deployment Descriptor vorliegt, diejenige aus dem Deskriptor bevorzugt wird, also die Annotation überstimmt. Das soll insbesondere dem Application Assembler bzw. dem Deployer ermöglichen, bei Bedarf die von der Komponenten-Entwicklerin per Annotationen vorgenommenen Voreinstellungen einfach über Deskriptoren zu modifizieren.

Wir werden im Folgenden ausschließlich die Verwendung der Annotationen erklären und an Beispielen demonstrieren und nicht näher auf die Deployment-Deskriptoren eingehen.

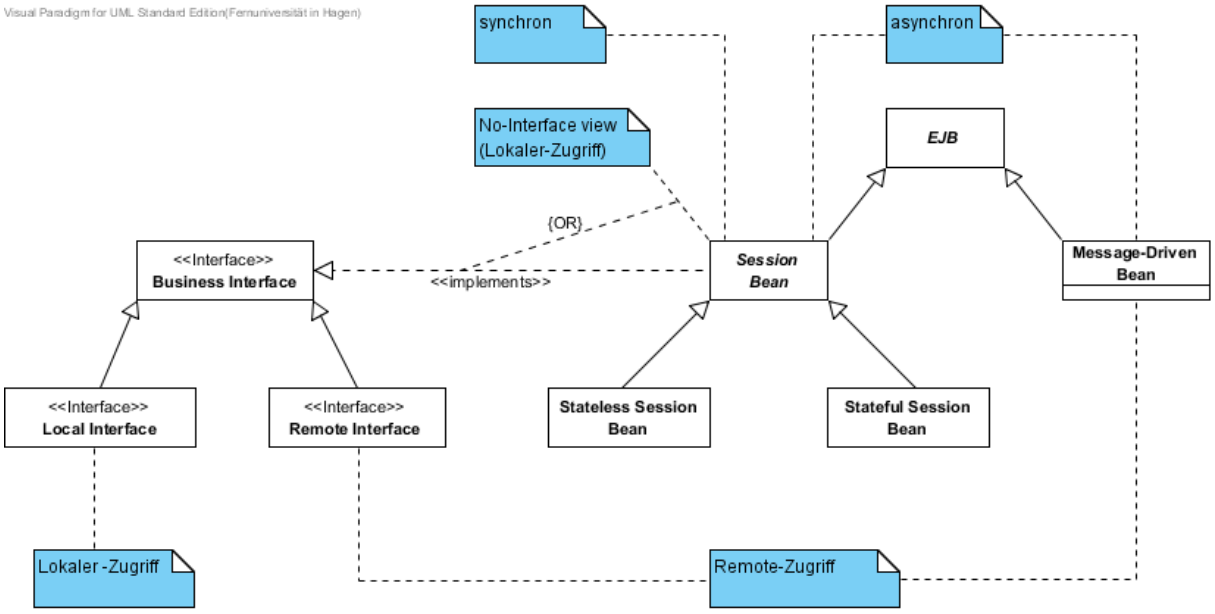

Abbildung 22.1.: Überblik über die Arten von Enterprise JavaBeans

22.1.8. Überblick über die EJB-Arten

Abbildung 22.1 stellt noch einmal die zu Beginn dieser Einführung vorgestellten verschiedenen Arten von EJBs zusammen.

In den folgenden Abschnitten wird genauer auf die Erstellung und Verwendung von Stateless Session Beans, Stateful Session Beans und Message-Driven Beans eingegangen. Außerdem vertiefen wir die No-Interface view sowie den asynchronen Aufruf von Session Beans. Als begleitendes Anwendungsbeispiel dient ein kleines Anwendungsprogramm zur Währungsumrechnung.

22.2. Implementierung einer Session Bean

22.2.1. Business Interface

Die Entwicklung einer Session Bean beginnt üblicherweise mit dem Festlegen der Schnittstelle, des Business Interface (vgl. Kapitel 21). Dabei handelt es sich um ein gewöhnliches Java-Interface, das über eine Annotation als Business Interface ausgezeichnet wird: ein Local Interface wird durch den Annotationstyp `javax.ejb.Local`, ein Remote Interface durch `javax.ejb.Remote` deklariert. In gewissen Fällen kann die Local-Annotation zwar auch entfallen (wenn z.B. die EJB-Klasse nur ein einziges Interface implementiert und dieses keine der beiden Annotationen aufweist, wird es automatisch als Local Interface interpretiert). Es ist jedoch besserer Programmierstil, immer eine explizite Auszeichnung vorzunehmen – schon aus Gründen der Lesbarkeit.

Listing 22.1 zeigt beispielhaft ein Local Interface für eine EJB mit einer einzigen Business-Methode. Für ein Remote Interface müsste lediglich die Annotation von Local zu Remote geändert werden.

```
1 package kurs01796.ke6;
2 import javax.ejb.Local;
3
4 @Local
5 public interface MeineAL {
6    public int anwendungslogik(int eingabe);
7 }
```

Listing 22.1: Local Business Interface

Die Signaturen der Business-Methoden sind beinahe beliebig, die Methodenbezeichner dürfen jedoch nicht mit `ejb` beginnen, da es dabei zu Konflikten mit internen Methoden kommen könnte. Es gibt noch weitere kleinere Einschränkungen. So können Business-Methoden z.B. nicht statisch sein, alle Methoden müssen als `public` und dürfen nicht als `final` deklariert sein, und die Parameter von Business-Methoden eines Remote Interface müssen serialisierbar sein.

22.2.2. Session Bean

Nach der Implementierung eines Business Interface kann die eigentliche Bean-Klasse erstellt werden. Dabei handelt es sich um eine gewöhnliche Java-Klasse, die das Business Interface implementiert. Diese Klasse ist wahlweise als Stateless oder Stateful Session Bean auszuzeichnen, wozu die Annotationstypen `javax.ejb.Stateless` bzw. `javax.ejb.Stateful` dienen. Die von Oracle vorgeschlagene Konvention zur Benennung von Bean-Klassen sieht vor, den Bezeichner eines Business Interface mit dem Zusatz Bean zu verwenden.

Codeausschnitt 22.2 zeigt den Aufbau einer Stateless Session Bean passend zum Business Interface aus Codeausschnitt 22.1.

```
1 package kurs01796.ke6;
2 import javax.ejb.Stateless;
3
4 @Stateless
5 public class MeineALBean implements MeineAL {
6    public int anwendungslogik (int eingabe) {
7       ...Implementierung der Anwendungslogik...
8    }
9 }
```

Listing 22.2: Stateless Session Bean

Es fällt auf, dass eine EJB-Klasse nicht als Subklasse einer (abstrakten) Framework-Klasse abgeleitet werden muss. Abgesehen von der Annotation handelt es sich um eine einfache, fast

frei formulierbare Java-Klasse, oft auch also „POJO“ (Plain Old Java Object) bezeichnet. Dies ist eine der wesentlichen in EJB Version 3.0 eingeführten Neuerungen und hat den Vorteil, dass so auch Klassenhierarchien von EJBs gebildet werden können. Diese Freiheit wurde durch Einführung von Annotationen und Dependency Injection möglich: Das Framework erkennt EJB-Klassen an ihren Annotationen, nicht an ihrem Typ. Benötigt eine EJB ein Objekt vom Framework (z.B. zur Transaktionskontrolle), deklariert sie diesen Bedarf durch eine Annotation und lässt sich eine Referenz auf das benötigte Objekt vom Container in eine Variable injizieren. Bei den einfachen EJBs, die wir in diesem Kapitel vorstellen, benötigen wir zwar noch keine solchen Injektionen, aber in der nächsten Kurseinheit werden wir davon Gebrauch machen.

Wie bereits in Abschnitt 22.1.3 beschrieben, muss seit der EJB Version 3.1 eine lokale Session Bean nicht unbedingt ein Local Interface implementieren. Stattdessen kann eine herkömmliche Java Klasse durch die Annotation `javax.ejb.LocalBean` als lokale Session Bean deklariert werden. In Codeausschnitt 22.3 ist fast die gleiche Stateless Session Bean wie in Codeausschnitt 22.2 dargestellt. Der Unterschied ist, dass die Session Bean in Codeausschnitt 22.3 kein Local Interface implementiert, dafür aber mit der Annotation `@LocalBean` versehen ist.

```
1  package kurs01796.ke6;
2  import javax.ejb.LocalBean;
3  import javax.ejb.Stateless;
4
5  @Stateless
6  @LocalBean
7  public class MeineALBean {
8     public int anwendungslogik (int eingabe) {
9        ...Implementierung der Anwendungslogik...
10    }
11 }
```

Listing 22.3: Stateless Session Bean ohne Local Business Interface

EJBs können neben Business-Methoden auch so genannte *Lifecycle Callback Methods* und *Interceptor Methods* implementieren, die vom Container zur Ereignisbehandlung vor der Zerstörung einer Bean-Instanz oder vor jeder Ausführung einer Business-Methode ausgelöst werden. Für Stateful Session Beans gibt es weiterhin die Möglichkeit, eine Remove-Methode zum Anfordern einer sofortigen Zerstörung der einem bestimmten Client zugeordneten Instanz (und somit des Conversational State) zu veranlassen. In den folgenden Kurseinheiten werden wir stets nur Stateless Session Beans ohne weitere Besonderheiten benötigen, der Vollständigkeit halber geben wir aber bezüglich dieser Punkte einige Beispiele.

22.2.3. Zerstörung von Stateful Session Beans

Wie in Abschnitt 22.1.2 bereits erklärt, bleibt eine Stateful Session Bean bis zu ihrer Zerstörung fest einem Client zugeordnet. Da für jeden neuen Client eine neue Instanz erzeugt wird, müssen nicht mehr benötigte Instanzen irgendwann auch wieder zerstört werden.

Der EJB-Container bietet hierzu einerseits einen Timeout-Mechanismus, der jede für eine gewis-

se Mindestzeitspanne nicht mehr von ihrem Client benutzte Stateful Session Bean automatisch entsorgt. Andererseits ist es natürlich dennoch wünschenswert, den Ressourcenverbrauch durch schon längst nicht mehr benötigte Instanzen so weit wie möglich zu vermeiden, indem man dem Client die Möglichkeit gibt, sich abzumelden und somit seine Stateful Session Bean zum sofortigen Zerstören freizugeben. Dazu kann eine Business-Methode, die im Folgenden kurz als *Remove-Methode* bezeichnet wird, mit einer Annotation des Typs `javax.ejb.Remove` versehen werden. Die Annotation wird an der Implementierung der Methode in der EJB-Klasse angebracht, als Methodenbezeichner bietet sich `remove` an. Falls ein Business-Interface verwendet wird, muss auch die Remove-Methode im Business Interface exportiert werden, damit der Client sie aufrufen kann.

Nach der Ausführung einer so annotierten Methode wird die Bean-Instanz vom EJB-Container zerstört, d.h. durch den Aufruf einer Remove-Methode gibt ein Client seine Bean frei. Im Standardfall findet die Zerstörung unabhängig davon statt, ob die Ausführung der Remove-Methode fehlerfrei verlief. Über den optionalen Boolean-Parameter `retainIfException` des Remove-Annotationstyps kann jedoch (durch Zuweisung von *true*, Default ist *false*) eingestellt werden, dass der Container die Instanz nur zerstören darf, wenn beim Aufruf der Remove-Methode kein Fehler (Exception) aufgetreten ist, andernfalls wird die Instanz behalten (s. Codeausschnitt 22.4).

```
1  import ...
2
3  @Stateful
4  public class AbcBean implements Abc
5  {
6     ...
7     @Remove(retainIfException=true)
8     public void remove() throws Exception {
9       ...
10    }
11 }
```

Listing 22.4: Remove-Methode einer Stateful Session Bean

Die Remove-Methode kann komplett leer bleiben, sodass ihr Aufruf lediglich den Container über die Freigabe zur Zerstörung informiert, sie kann aber auch Anwendungslogik enthalten, nach deren Ausführung die Zerstörung der Bean erfolgen soll. Beispielsweise könnte eine im Rahmen eines Bestellprozesses in einem Online-Shop benutzte Stateful Session Bean nach dem Absenden der Bestellung überflüssig werden, sodass die Business-Methode zum Abschließen der Bestellung als Remove-Methode annotiert wird. Für die Freigabe von im Zustand der Bean gespeicherten Ressourcen ist die Remove-Methode jedoch *nicht* der geeignete Ort, da eine solche Freigabe auch bei einer automatischen Zerstörung im Falle eines Timeouts stattfinden sollte, die Remove-Methode jedoch nur bei manuellem Aufruf durch den Client ausgeführt wird. Für solche Aufräumzwecke ist vielmehr die im Folgenden vorgestellte PreDestroy-Callback-Methode zu verwenden.

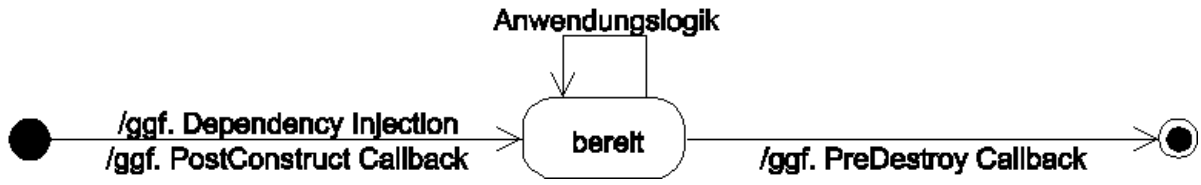

Abbildung 22.2.: Lebenszyklus einer Stateless Session Bean

22.2.4. Callback-Methoden & Lebenszyklus einer Session Bean

Neben Business-Methoden (einschließlich der soeben vorgestellten Remove-Methode) können in einer Session Bean noch so genannte *Callback-Methoden*[6] (*Lifecycle Callback Interceptor Methods*) implementiert werden. Eine Callback-Methode wird nicht vom Client, sondern vom Container beim Auftreten eines gewissen Ereignisses im Lebenszyklus der Bean aufgerufen, so dass die Bean auf dieses Ereignis reagieren kann. Die EJB-Spezifikation bezeichnet dies als *Lifecyle Event Callback*. Ein Ereignis, über das der EJB-Container die Bean durch einen solchen Callback informieren kann, wird als Callback-Ereignis bezeichnet. Im Folgenden werden zunächst Lebenszyklus und *Callback-Ereignisse* jeweils zu Stateless und Stateful Session Beans vorgestellt, bevor abschließend die Implementierung der Callback-Methoden behandelt wird.

Eine Instanz einer *Stateless Session Bean* kennt nur einen Zustand: Sie ist bereit und wartet auf Aufrufe ihrer Business-Methoden. Der Container verwaltet die Menge der im Pool bereit liegenden Stateless Session Beans automatisch, indem er je nach Bedarf weitere Instanzen erzeugt oder zerstört. In diesem Lebenszyklus gibt es zwei Callback-Ereignisse: das Eintreten in den Bereit-Zustand nach Konstruktion (*PostConstruct*) und das Verlassen des Bereit-Zustands vor Zerstörung (*PreDestroy*).

Abbildung 22.2 stellt den Lebenszyklus einer *Stateless Session Bean* als Zustandsdiagramm dar: Sobald der Container eine neue Bean-Instanz erzeugt hat, führt er zunächst die Dependency Injection für alle entsprechenden Attribute (sofern vorhanden) durch und ruft danach die PostConstruct-Callback-Methode der Bean (sofern vorhanden) auf. Danach ist die Bean bereit. Vor ihrer Zerstörung wird die PreDestroy-Callback-Methode (sofern vorhanden) ausgeführt.

Etwas komplizierter ist der Lebenszyklus einer *Stateful Session Bean*, der in einer vereinfachten Form in Abbildung 22.3 dargestellt ist (ohne Berücksichtigung des Transaktionsmanagements, ein ausführlicheres Zustandsdiagramm findet sich in [EJB/Spec]). Auch eine Stateful Session Bean hat einen Bereit-Zustand, in dem sie direkt Anwendungslogik ausführen kann, und auch hier gilt, dass bei der Instanziierung zunächst die Dependency Injection und danach der PostConstruct-Callback ausgeführt werden. Bei ihrer Zerstörung findet ebenfalls ein PreDestroy-Callback statt. Die Zerstörung findet, wie bereits beschrieben, entweder nach einem Timeout statt oder nachdem eine mit `Remove` annotierte Methode ausgeführt wurde.

Da für jeden Client eine eigene Instanz der Stateful Session Bean angelegt wird und sich durch

[6] engl. callback-method = Rückruffunktion. Eine Rückruffunktion ist eine Funktion, die einer anderen Funktion als Parameter übergeben wird. Von dieser wird sie dann unter bestimmten Bedingungen aufgerufen.

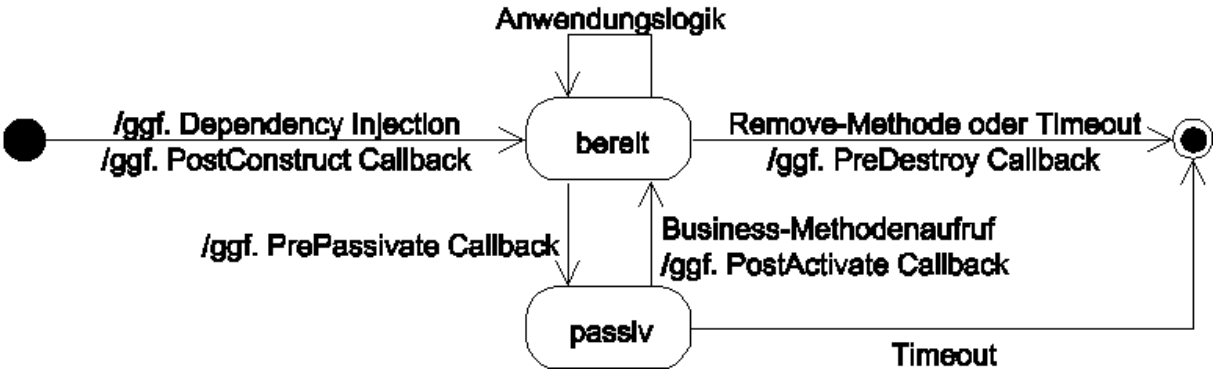

Abbildung 22.3.: Lebenszyklus einer Stateful Session Bean (vereinfacht)

lange Timeout-Intervalle mitunter recht viele Karteileichen unter den Instanzen finden, kann die Situation eintreten, dass die Systemressourcen eines Servers nicht für alle zu haltenden EJB-Instanzen ausreichen. In diesem Fall kann der EJB Container einige – typischerweise nach dem LRU-Schema (Least Recently Used) ausgewählte – Stateful Session Beans *passivieren*, d.h. in einen Sekundärspeicher auslagern (vgl. Zustand „passiv“ in Abbildung 22.3). Ruft der Client einer passivierten Bean eine Business-Methode auf, so wird die Bean zunächst vom Container wieder aktiviert, und anschließend wird die Business-Methode ausgeführt.

Mit der Passivierung und der Reaktivierung kommen bei Stateful Session Beans zwei weitere Callback-Ereignisse hinzu: Ein *PrePassivate*-Callback findet statt, bevor die Passivierung eingeleitet wird, ein *PostActivate*-Callback wird nach Abschluss der Reaktivierung ausgelöst, aber noch vor der Ausführung der Business-Methode, deren Aufruf die Reaktivierung veranlasst hat.

Bei der Passivierung findet eine Serialisierung und Auslagerung des gesamten Conversational State der Stateful Session Bean statt. Deshalb kommt diesen beiden Callbacks vor allem dann eine Bedeutung zu, wenn vor der Passivierung gewisse Maßnahmen ergriffen werden müssen, um die Serialisierung zu ermöglichen. So können insbesondere im Conversational State gehaltene so genannte `open resources` wie z.B. offene Socketverbindungen zur TCP/IP-Kommunikation nicht passiviert und später wieder hergestellt werden. Eine solche Verbindung muss vielmehr vor der Passivierung geschlossen und kann nach der Reaktivierung wieder geöffnet werden. Ist der Zustand einer Stateful Session Bean nach dem PrePassivate-Callback nicht serialisierbar[7], so darf der EJB Container die Bean zerstören statt sie zu passivieren!

Um eine Callback-Methode zu einem der vier Ereignisse zu schreiben, ist in der EJB-Klasse eine öffentliche Methode der Signatur

```
void methodenBezeichner()
```

zu erstellen und mit einer Annotation eines der folgenden Typen zu versehen (wie in Codeausschnitt 22.5 demonstriert):

```
javax.annotation.PostConstruct
```

[7] Wann genau ein Conversational State passivierbar ist, erklärt [EJB/Spec] in Abschnitt 4.2

```
javax.annotation.PreDestroy
javax.ejb.PrePassivate
javax.ejb.PostActivate
```

PrePassivate- oder PostActivate-Callbacks finden natürlich höchstens bei Stateful Session Beans statt. In anderen EJBs werden entsprechende Callback-Methoden, sofern vorhanden, niemals im Rahmen eines Callbacks ausgeführt. PostConstruct- und PreDestroy-Methoden können dagegen in allen EJBs (auch Message-Driven Beans) eingesetzt werden.

Im Gegensatz zur oben vorgestellten Remove-Methode wird eine PreDestroy-Callback-Methode grundsätzlich beim Zerstören einer Session Bean aufgerufen. Wird auf einer Stateful Session Bean die Remove-Methode aufgerufen, so wird der PreDestroy-Callback nach deren Abarbeitung ausgelöst.

```
1  import ...
2
3  @Stateless
4  public class AbcBean implements Abc
5      ...
6      @PreDestroy
7      public void vorZerstoerung() {
8          ...
9      }
10 }
```

Listing 22.5: PreDestroy-Callback-Methode

22.2.5. Interceptors

Viele Dienste des EJB-Containers, wie z.B. das transparente Transaktions- oder Sicherheitsmanagement oder die Zerstörung einer Stateful Session Bean nach Ausführung einer Remove-Methode, sind über so genannte Interceptors realisiert. Interceptors sind Objekte, die – bildlich gesprochen – einen Methodenaufruf abfangen[8] und vor und/oder nach der aufgerufenen Methode weitere Aktionen einfügen. Wann die Nutzung von Interceptors sinnvoll ist, soll an einem einfachen Beispiel gezeigt werden. Nehmen wir an, wir haben mehrere EJBs, die jeweils über mehrere Methoden (Business Methoden) verfügen. Nun soll zu Analysezwecken festgehalten werden, wie lange es dauert eine dieser Methoden bei einem Aufruf abzuarbeiten. In dieser Methode kann zu Beginn ein Zeitnehmer gestartet und am Ende der Methode dieser Zeitnehmer gestoppt und die benötigte Zeit ausgegeben werden. Soll dies für jede Methode erfolgen, führt die Integration dieser Codeschnipsel unweigerlich zu redundantem Code. Zudem beinhalten die entsprechenden Business-Methoden nun Code, der nicht zur eigentlichen Aufgabe der Methoden gehört. An dieser Stelle kann ein Interceptor eingesetzt werden, der durch eine einfache Java-Klasse realisiert wird. Zur Laufzeit wird automatisch eine Instanz dieser Klasse erzeugt. Dieses Interceptor-Objekt muss über eine (oder mehrere) Methoden verfügen, die an die Business-Methoden von EJBs angehängt oder ihnen vorangestellt werden.

[8] engl. to intercept = abfangen

Aufrufe von Business-Methoden werden auch durch Container-interne Interceptors abgefangen, die z.B. für das Starten einer Transaktion vor der Ausführung der Businessmethode oder für den Abschluss der Transaktion nach ihrer Ausführung sorgen können. Auch die oben behandelten Callback-Methoden zum Abfangen von Ereignissen und Einschieben von Aktionen werden laut [EJB/Spec] zu den Interceptors gezählt (*Interceptors for Lifecycle Event Callbacks*), auch wenn diese – streng genommen – nicht selbst aktiv Ereignisse abfangen, sondern passiv warten, bis sie vom Container aufgerufen werden.

Anstatt eine Interceptor-Klasse zu implementieren, ist es ebenso möglich eine Interceptor-Methode direkt in eine EJB-Klasse zu integrieren. Eine solche Methode wird dann jeden Aufruf einer beliebigen Business-Methode dieser EJB-Klasse abfangen (genau genommen wartet auch eine solche Interceptor-Methode, wie eine Callback-Methode, passiv auf einen Aufruf durch den Container.)

Hierzu ist in der Bean-Klasse eine Methode der Signatur

```
Object methode(javax.interceptor.InvocationContext) throws Exception
```

zu erstellen und mit einer Annotation des Typs

```
jaxax.interceptor.AroundInvoke
```

zu markieren. Die Methode muss nicht öffentlich sein, sondern darf einen beliebigen Access Modifier[9] tragen, auch `private`. Sie darf lediglich weder `static` noch `final` deklariert sein. Damit die so erstellte Interceptor-Methode nicht anstelle einer Business-Methode, sondern – wie die Annotation `AroundInvoke` es ausdrückt – „um sie herum“ ausgeführt wird, muss die Stelle im Code der Interceptor-Methode, an der die Business-Methode eingeschoben werden soll, durch einen Aufruf der Methode `proceed()` des als Parameter übergebenen `InvocationContext`-Objekts markiert werden. Alle Anweisungen der Interceptor-Methode, die vor diesem `proceed`-Aufruf stehen, werden also vor der Anwendungslogik eingeschoben, alle Anweisungen nach dem `proceed`-Aufruf danach. Da die eingeschobene Business-Methode ein Funktionsergebnis zurückgeben könnte, gibt die `proceed`-Methode ein `Object` aus, das seinerseits von der Interceptor-Methode zurückzugeben ist.

Codeausschnitt 22.6 demonstriert dies am Beispiel einer einfachen Logging-Methode, die jeweils vor und nach der Ausführung einer Business-Methode der Bean einen Eintrag ins Server-Log schreibt. Die Zeile mit dem Proceed-Aufruf ist farblich hinterlegt.

```
1 import javax.interceptor.AroundInvoke;
2 import javax.interceptor.InvocationContext;
3 import javax.ejb.LocalBean;
4 import ...;
5
6 @Stateless
7 @LocalBean
8 public class MeineALBean {
```

[9] Ein Access Modifier ist ein Schlüsselwort in einer objektorientierten Sprache, das die Sichtbarkeit von Klassen, Methoden oder Attributen bestimmt, z.B. in Java: `public`, `private`, `protected`.

```
9     ... Business-Methoden ...
10    @AroundInvoke
11    private Object log(InvocationContext ctx) throws Exception {
12       String methode = ctx.getMethod().getName();
13
14       System.out.println("[MeineALBean] Anwendungslogik in " + "Methode "+
             methode+" wird gestartet.");
15
16       Object rueckgabe = ctx.proceed();
17
18       System.out.println("[MeineALBean] Methode " + methode + " beendet.");
19
20       return rueckgabe;
21    }
22 }
```

Listing 22.6: Interceptor-Methode mit AroundInvoke-Annotation

Neben dieser `AroundInvoke`-Annotation gibt es noch weiterführende Möglichkeiten, Interceptors umzusetzen. So können z.B. methodenspezifische Interceptors erstellt werden, die nicht die Aufrufe sämtlicher Business-Methoden einer EJB abfangen, sondern nur bestimmter Methoden. Es können separate Interceptor-Klassen erstellt werden, in denen Interceptors für Business-Methoden und/oder Lifecycle Callback Interceptor-Methoden implementiert werden. Jede Interceptor-Klasse kann dann von mehr als nur einer EJB-Klasse benutzt werden. Mit so genannten *Default Interceptors* besteht die Möglichkeit, Interceptor-Methoden global allen EJBs aus einem EJB-JAR-Modul zuzuordnen (über einen Deployment Descriptor).

Das Interceptor-Muster ermöglicht es auf diese Weise sowohl dem Framework selbst als auch der Bean-Autorin, auf relativ einfache Weise Code z.B. für Ereignisbehandlung oder Logging in ein Anwendungsprogramm einzuschieben. Die Interceptors können auch außerhalb der Bean-Klasse implementiert sein und somit sogar ohne Kenntnis der Bean-Autorin eingefügt werden.

22.3. Verwendung einer Session Bean

Zum Zugriff auf die Anwendungslogik benötigt der Client[10] eine Referenz auf eine Instanz der Bean. Die Instanziierung darf er nicht selbst (mittels `new`) vornehmen, sondern er muss eine Instanz vom EJB-Container anfordern. Nur so sind Verteilung, transparente Zuteilung von Instanzen etc. (vgl. Abschnitt 21.1) realisierbar. Implementiert die Bean ein Business Interface, wird dabei keine direkte Referenz auf die Bean-Instanz erworben (was bei entfernten EJBs auf anderen Servern auch gar nicht möglich wäre). Vielmehr bekommt der Client eine Referenz auf ein lokales *Stub-Objekt*[11], dessen Klasse ebenfalls das Business Interface implementiert und

[10] Im Zusammenhang mit der Verwendung einer EJB bezeichnen wir mit *Client* diejenige Softwarekomponente, die die EJB verwendet, also z.B. der Controller der Web-Schicht. Der Begriff ist nicht mit dem Client der Web-Anwendung (im Folgenden zur Abgrenzung ggf. als *Web-Client* bezeichnet) zu verwechseln.

[11] engl. stub = Stummel, Stumpf. Ein Stub bezeichnet Programmcode, der einen anderen Programmcode ersetzt, z.B. wenn sich dieser Programmcode auf einem anderen Computer befindet. Unter einem Stub-Objekt versteht

das z.B. für den Aufruf der Interceptor-Methoden oder bei Remote Interfaces für die Serialisierung der Parameter und die Weiterleitung des Aufrufs an die eigentliche Bean-Klasse sorgt. Dieser ganze Mechanismus wird vom EJB-Container bereitgestellt und ist für den Client transparent: Aus seiner Sicht wird ein Objekt vom Typ des Business Interfaces bezogen, auf welchem die Business-Methoden aufgerufen werden können. Wir werden im Folgenden diese abstrakte Sicht beibehalten und meist vereinfachend vom Beziehen einer EJB (statt vom Beziehen einer Referenz auf einen Stub) sprechen.

Es gibt für einen Client nun zwei Möglichkeiten, eine EJB vom Container anzufordern: mit Hilfe von Annotationen zur Dependency Injection oder durch Lookup-Anweisungen im Programm-Code.

22.3.1. Injektion einer Session Bean

Dependency Injection ist die einfachste Möglichkeit, eine Session Bean zu beziehen (siehe auch Abschnitt 21.2).

Hierzu ist in der Client-Klasse zunächst ein Attribut vom Typ des Business Interface zu deklarieren, bzw. bei No-Interface (wenn also kein Business Interface verwendet wird) ein Attribut vom Typ der Bean. Wird ein Business Interface verwendet, bestimmt dessen Art (Local oder Remote) die Zugriffsart. Falls also zu einer Session Bean sowohl ein Remote als auch ein Local Interface existiert, wird über die Typdeklaration dieses Attributs bestimmt, welche der beiden angebotenen Zugriffsarten zu verwenden ist.

Das Attribut wird mit einer Annotation des Typs `javax.ejb.EJB` versehen, einer für den (den Client ausführenden) Container bestimmten Meta-Information, die den Bedarf an einer EJB des entsprechenden Typs ausdrückt. Der Container sorgt dann zur Laufzeit dafür, dass rechtzeitig vor einem Zugriff des Clients auf dieses Attribut eine Instanz der EJB erzeugt oder aus einem Pool ungenutzter Instanzen entnommen wird und eine Referenz darauf in das Attribut eingetragen wird. Codeausschnitt 22.7 zeigt als Beispiel einen als Servlet implementierten Client, der die Stateless Session Bean aus Codeausschnitt 22.3 anspricht.

```
1  import javax.ejb.EJB;
2  import kurs01796.ke6.MeineALBean;
3  import ...
4
5  public class Controller extends HttpServlet {
6
7     @EJB private MeineALBean al;
8
9     protected void doGet(HttpServletRequest request,
10                          HttpServletResponse response)
11                          throws ServletException, IOException {
12       int i = ...;
```

man einen (Client-seitigen) Stellvertreter für ein entferntes Objekt, das auf einem Server liegt, mit dem der Client kommuniziert.

```
13          int j = al.anwendungslogik(i); //Anwendungslogik aufrufen
14          ...
15      }
16  }
```

Listing 22.7: Injektion einer Session Bean

Damit die Dependency Injection funktionieren kann, muss der Client in einem Java EE-Container ausgeführt werden, der seine Komponenten introspiziert[12] und dabei (unter anderem) nach solchen EJB-Annotationen sucht. Diese Container sind Web Container, EJB Container und Application Client Container (vgl. Abb. 1.1 in Kurseinheit 3).

Die Injektion von EJBs kann also insbesondere in Servlets, in (anderen) EJBs, in Interceptor-Klassen oder auch in Java ServerFaces verwendet werden[13].

22.3.2. Lookup einer Session Bean

Da Java EE-Ressourcen über mehrere Server verteilt sein können, verwendet Java EE einen Server-übergreifend arbeitenden Namens- und Verzeichnisdienst zu deren Referenzierung. Um diesen Dienst anzusprechen, wird die Schnittstelle JNDI (*Java Naming and Directory Interface*, vgl. [JNDI]) eingesetzt. Ressourcen werden unter einem so genannten *JNDI-Namen* verzeichnet und können anhand dieses Namens über einen so genannten *Lookup* lokalisiert werden. Namenszuordnungen werden in so genannten Kontexten organisiert: Ein *Kontext* ist eine Menge von Paaren aus jeweils einem Namen und einer an diesen Namen gebundenen Ressource. Dabei kann eine solche Ressource wieder ein Kontext (Subkontext) sein, d.h. die Kontexte können hierarchisch organisiert werden. Als Ausgangspunkt für JNDI-Lookups dient der so genannte *InitialContext*.

Um im Rahmen einer Managed Bean eine Session Bean über einen JNDI-Lookup beziehen zu können, müssen zwei Voraussetzungen erfüllt sein:

1. Die Session Bean muss überhaupt in einem der Client-Komponente (hier der Web-Anwendung) zugänglichen Kontext verzeichnet sein
2. Der Name, unter dem sie verzeichnet ist, muss bekannt sein

Zu Remote Interfaces legt z.B. der Glassfish Application Server bei deren Deployment automatisch einen JNDI-Eintrag an und verwendet den qualifizierten Klassennamen als JNDI-Namen, sodass der Client sie darüber direkt mit einem Lookup ansprechen kann. Auch bei der oben vorgestellten Dependency Injection findet ein solcher Lookup statt, nur wird er – transparent für den Client – durch den Container durchgeführt.

[12] Unter Introspektion (oder Reflexion) versteht man in der Informatik, dass ein Programm seine eigene Struktur kennt. In der objektorientierten Programmierung erlaubt Introspektion die Sammlung von Informationen über Klassen oder deren Instanzen zur Laufzeit des Programms.

[13] Eine tabellarische Auflistung von Komponenten, die Injektion unterstützen, findet sich in [JEE/Spec] in Table EE.5-1.

Etwas anders verhält es sich beim Verwenden einer Session Bean über ein Local Interface. Obwohl der Bezug per Dependency Injection für Remote und Local Interface syntaktisch gleich aussieht und aus Client-Sicht auch scheinbar das Gleiche passiert, arbeitet der Container hinter den Kulissen jeweils nach einem anderen Prinzip. Insbesondere wird zu einem Local Interface nicht automatisch ein JNDI-Eintrag angelegt. Um in diesem Fall einen JNDI-Lookup vornehmen zu können, muss der Client eine – beim Deployment durch seinen Container ausgewertete – Deklaration vornehmen, welche EJB-Klasse er benötigt und unter welchem JNDI-Namen er sie im Namensverzeichnis erwartet. Dies kann auf zweierlei Weise erfolgen:

- Falls Teile des Clients introspiziert werden (bei einer Web-Anwendung z.B. Servlets oder Managed Beans), kann dort die EJB per Dependency Injection bezogen werden. Dabei kann in der EJB-Annotation über den Parameter name ein JNDI-Name festgelegt werden, über den sich die so bezogene Bean dann auch von anderen, nicht introspizierten Teilen per Lookup beziehen lässt.
  Beispiel: `@EJB(name="ejb/meineALBean") private MeineALBean al;`
- Bei einer Web-Anwendung kann ein Eintrag im Deployment Decriptor `web.xml` angelegt werden, der insbesondere die EJB-Klasse und den gewünschten JNDI-Namen spezifiziert (vgl. Codeausschnitt 22.8).

Da wir in den weiteren Beispielen JavaServer Faces sowie Managed Beans als Schnittstelle zwischen der View und der eigentlichen Programmlogik (EJBs) benutzen werden, wird hier nicht weiter auf die Konfigurationen innerhalb des Deployment Descriptor eingegangen. Innerhalb von Managed Beans kann einfach die Annotation `@EJB` genutzt werden.

22.4. Beispiel: Währungsumrechner mit Stateless Session Bean

Das folgende Beispiel illustriert die Implementierung des Anwendungskerns einer Web-Anwendung mit Hilfe einer Session Bean. Wir werden zunächst eine Stateless Session Bean verwenden und anschließend in einem weiteren Beispiel eine erweiterte Version der Beispiel-Anwendung unter Einsatz einer Stateful Session Bean entwickeln.

Als Anwendung haben wir einen Währungsumrechner für die Konvertierung von Beträgen zwischen Dollar und Euro gewählt, denn dieser hat eine kompakte Anwendungslogik und benötigt nur eine sehr einfache Web-Schnittstelle. In der Web-Anwendung wird ein Betrag eingegeben, der entweder als Dollar-Betrag interpretiert und als solcher in einen Euro-Betrag konvertiert wird, oder umgekehrt als Euro-Betrag interpretiert und in Dollar umgerechnet wird. Abbildung 22.4 zeigt die Benutzungsschnittstelle unserer Beispielanwendung nach einer erfolgten Umrechnung.

Im Folgenden werden der Anwendungskern und Teile der Web-Schicht vorgestellt. In den hier abgedruckten Programmfragmenten verzichten wir aus Platzgründen auf die meisten Quelltext-Kommentare (insbesondere auf JavaDoc-Kommentare).

Abbildung 22.4.: Benutzungsschnittstelle des Dollar-Euro-Währungsumrechners

22.4.1. Anwendungslogik

Der Anwendungskern dieses Währungsumrechners wird durch eine einzige Stateless Session Bean gebildet, die zwei Methoden anbietet: eine zur Umrechnung von Dollar in Euro und eine zur Umrechnung von Euro in Dollar.

Codeausschnitt 22.8 zeigt die Implementierung der Stateless Session Bean. Der von den beiden Umrechnungsmethoden benötigte Wechselkurs wird in einer Konstante der EJB-Klasse hinterlegt. Die Session Bean muss hier kein Business Interface implementieren. Die Annotation LocalBean wird explizit verwendet, da die gesamte Anwendung (Web-Schicht und EJB) auf einem einzigen Application Server betrieben werden soll.

```
1  package kurs1796.ke6;
2
3  import java.math.BigDecimal;
4  import javax.ejb.LocalBean;
5  import javax.ejb.Stateless;
6
7  @Stateless
8  @LocalBean
9  public class Converter
10 {
11     private final static BigDecimal EUROKURS = new BigDecimal(0.725268349);
12
13     public BigDecimal dollarZuEuro(BigDecimal dollar) {
14         return dollar.divide(EUROKURS, 2, BigDecimal.ROUND_HALF_UP);
15     }
16
17     public BigDecimal euroZuDollar(BigDecimal euro) {
18         BigDecimal dollar = euro.multiply(EUROKURS);
19         return dollar.setScale(2, BigDecimal.ROUND_HALF_UP);
20     }
21 }
```

Listing 22.8: Anwendungslogik des Dollar-Euro-Umrechners

22.4.2. Web-Schicht (JavaServer Faces)

Für diese einfache Web-Anwendung kommt man mit einer einzigen JSF-Seite aus. Diese enthält ein paar Beschreibungen, ein Texteingabefeld für den einzugebenen Betrag, eine Schaltfläche, um die Umrechnung zu starten und zwei Textausgabefelder zur Anzeige der berechneten Ergebnisse.

Codeausschnitt 22.9 zeigt den gesamten Inhalt der JSF-Datei.

```
1  <?xml version='1.0' encoding='UTF-8' ?>
2  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
       w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
3  <html xmlns="http://www.w3.org/1999/xhtml"
4        xmlns:h="http://xmlns.jcp.org/jsf/html">
5      <h:head>
6          <title>Euro-Dollar-Konverter</title>
7      </h:head>
8      <h:body>
9          <h2>Euro-Dollar-Konverter</h2>
10
11         <p>
12             Geben Sie einen zu konvertierenden Betrag
13             (Euro oder Dollar) ein:
14         </p>
15         <h:form>
16             <h:inputText value="#{converterManagedBean.eingabe}" />
17             <h:commandButton value="Umrechnen" action="#{
                   converterManagedBean.berechne()}" />
18         </h:form>
19         <hr />
20         <h:panelGrid
21            rendered="#{not empty converterManagedBean.eingabe}">
22             <h:outputLabel value="#{converterManagedBean.eingabe} Euro = #{
                   converterManagedBean.dollar} Dollar" />
23             <h:outputLabel value="#{converterManagedBean.eingabe} Dollar =
                   #{converterManagedBean.euro} Euro" />
24         </h:panelGrid>
25     </h:body>
26 </html>
```

Listing 22.9: JSF-Seite der Web-Schicht

Die `value`-Attribute der Texteingabefelder und der Textausgabefelder werden mit Properties der Managed Bean ConverterManagedBean verbunden, das `action`-Attribut der Schaltfläche mit der Methode `berechne()` dieser Managed Bean. Eine Besonderheit stellt das Attribut `rendered` innerhalb des `panelGrid`-Elementes dar. Dieses erwartet einen Ausdruck, der entweder zu *true* oder zu *false* ausgewertet werden kann. Wird der Ausdruck zu *true* ausgewertet, wird das Element, in dem sich das `rendered`-Attribut befindet, angezeigt (inklusive seiner Kindelemente), wird er zu *false* ausgewertet, wird das Element (inklusive seiner Kindelemente) nicht angezeigt. Im letzteren Fall wird mit Hilfe des EL-Ausdrucks geprüft, ob die Property

`eingabe` der Managed Bean einen Wert enthält oder nicht. Der Ausdruck wird dann zu *true* ausgewertet, wenn die Property einen Wert enthält.

22.4.3. Web-Schicht (Managed Bean)

Als letzter Teil der Web-Anwendung wird die Managed Bean `ConverterManagedBean` implementiert. Da Managed Beans in einem Web Container des Application Servers ausgeführt werden, ist es möglich der Managed Bean Instanzen von EJBs durch Dependency Injection zugänglich zu machen. Hierzu wird die Annotation `@EJB` genutzt.

Codeausschnitt 22.10 zeigt die Implementierung der Managed Bean `ConverterManagedBean`. Aus Gründen der Übersichtlichkeit werden die Getter und Setter hier nicht aufgeführt.

```
1  package kurs1796.ke6;
2
3  import java.math.BigDecimal;
4  import javax.ejb.EJB;
5  import javax.inject.Named;
6  import javax.enterprise.context.RequestScoped;
7
8  @Named
9  @RequestScoped
10 public class ConverterManagedBean
11 {
12     @EJB private Converter bean;
13
14     private BigDecimal eingabe;
15     private BigDecimal dollar;
16     private BigDecimal euro;
17
18     public void berechne() {
19         dollar = bean.euroZuDollar(eingabe);
20         euro = bean.dollarZuEuro(eingabe);
21     }
22
23     ... Getter und Setter ...
24
25     public ConverterManagedBean() {
26
27     }
28 }
```

Listing 22.10: Managed Bean der Web-Schicht

Durch die Annotation `@EJB` wird eine Instanz der EJB-Klasse `Converter` der Variable `bean` zugewiesen. Die Methode `berechne()` ruft beide Umrechnungsmethoden der EJB auf und weist die Rückgabewerte den entsprechenden Variablen zu. Der Anwenderin wird somit bei Betätigung der Schaltfläche „Umrechnen“ die zum eingegebenen Wert entsprechenden Dollar- und Euro-Werte unterhalb des Eingabeformulars angezeigt.

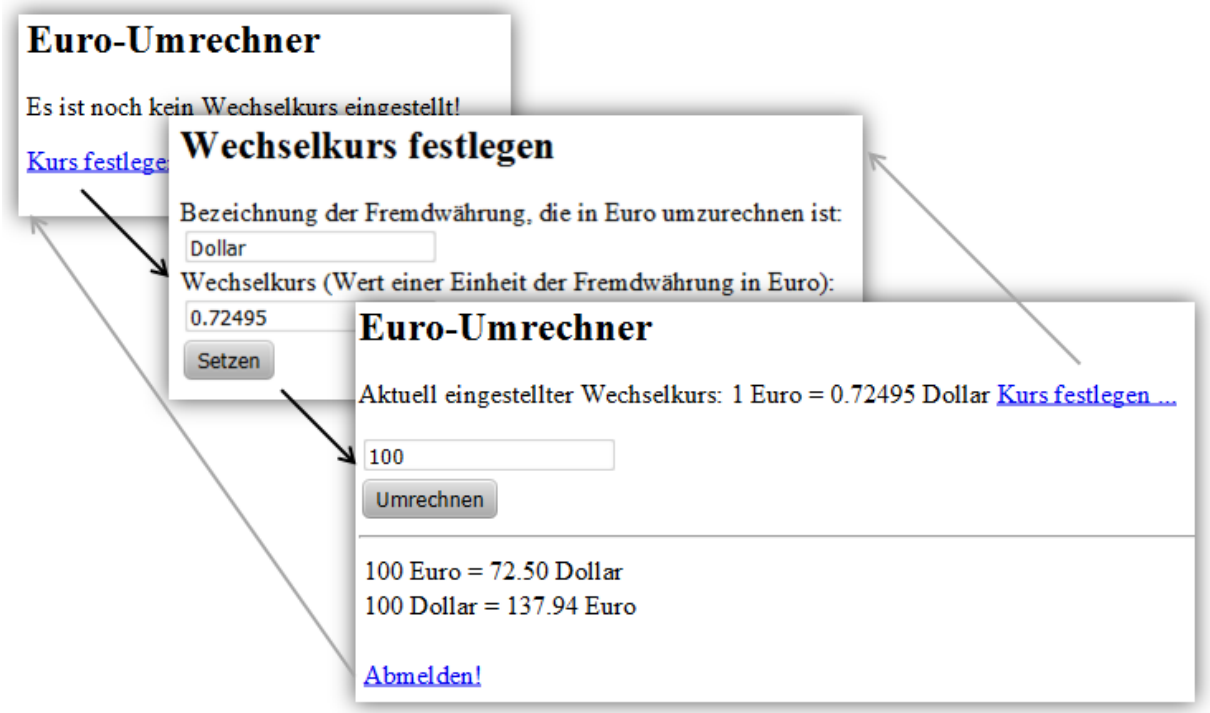

Abbildung 22.5.: Benutzungsschnittstelle des Fremdwährung-EUR-Umrechners

22.5. Beispiel: Währungsumrechner mit Stateful Session Bean

Im Folgenden wird das Beispiel aus Abschnitt 22.5 erweitert: Statt ausschließlich zwischen Dollar und Euro umrechnen zu können, wobei der Wechselkurs als Konstante vorliegt, soll nun eine Fremdwährung-Euro-Umrechnung angeboten werden, bei der die Benutzer eine Bezeichnung für die Fremdwährung sowie den Wechselkurs selbst festlegen können. Zu diesem Zweck arbeitet diese Web-Anwendung sitzungsbasiert.

Beide Einstellungen (Währungsbezeichnung und Kurs) könnten direkt im Session Scope gespeichert werden, jedoch wäre dann bei jedem Aufruf einer Umrechnungsmethode der Umrechnungskurs per Parameter an die Anwendungslogik zu übergeben. Dafür würde auch noch eine Stateless Session Bean ausreichen. Die hier verfolgte Alternative sieht dagegen vor, dass die Währungsbezeichnung und der Kurs in der Session Bean gespeichert werden sollen. Da der Kurs in diesem Beispiel nicht mehr konstant ist, sondern einen Client-spezifischen Zustand darstellt, benötigen wir dazu eine Stateful Session Bean.

Abbildung 22.5 zeigt die Benutzungsschnittstelle des Währungsumrechners und die Navigation zwischen den verschiedenen Seiten. Beim Start einer Sitzung, wenn noch kein Kurs eingestellt wurde, wird nur eine entsprechende Meldung mit einem Link angezeigt. Der Link führt zu einer Einrichtungsseite, auf der Währungsbezeichnung und Kurs festgelegt werden. Danach ist der

Umrechner analog zum Dollar-Euro-Umrechner des letzten Beispiels zu bedienen. Im Vergleich zur Abbildung 22.4 ist jedoch die Anzeige des eingestellten Wechselkurses mit einem Link für den Rücksprung zur Einrichtungsseite hinzugekommen. Um nicht mehr benötigte Instanzen der Stateful Session Bean nicht immer bis zu einem Timeout aufheben zu müssen, bekommt die Benutzerin zusätzlich die Möglichkeit zur Abmeldung, bei der die Bean-Instanz zerstört wird.

22.5.1. Anwendungslogik

Auch die zuletzt besprochene Web-Anwendung soll wieder zentral auf einem einzigen Server betrieben werden, weshalb wir eine No-Interface View wählen. Analog zum Beispiel aus Abschnitt 22.4 sehen wir wieder zwei Umrechnungsroutinen (Euro zu Fremdwährung und umgekehrt) vor, zusätzlich Methoden zum Setzen und Auslesen des Umrechnungskurses, zum Setzen und Auslesen der Beschreibung, sowie eine Remove-Methode zur Zerstörung der Instanz (vgl. Codeausschnitt 22.11).

```
1  package kurs1796.ke6;
2
3  import java.math.BigDecimal;
4  import javax.ejb.LocalBean;
5  import javax.ejb.Remove;
6  import javax.ejb.Stateful;
7  import javax.interceptor.AroundInvoke;
8  import javax.interceptor.InvocationContext;
9
10 @Stateful
11 @LocalBean
12 public class Converter
13 {
14     private BigDecimal fwKurs = new BigDecimal(1.0);
15     private String fwBezeichnung;
16
17     ... Getter und Setter für Kurs fwKurs und fwBezeichnung ...
18
19     public BigDecimal fwZuEuro(BigDecimal fw) {
20         return fw.divide(getFwKurs(), 2, BigDecimal.ROUND_HALF_UP);
21     }
22
23     public BigDecimal euroZuFw(BigDecimal euro) {
24         BigDecimal fw = euro.multiply(getFwKurs());
25         return fw.setScale(2, BigDecimal.ROUND_HALF_UP);
26     }
27
28     @Remove
29     public void remove() {
30         System.out.println("[Converter] remove");
31     }
32
33     @AroundInvoke
34     private Object log(InvocationContext ctx) throws Exception {
```

```
35          String methode = ctx.getMethod().getName();
36
37          System.out.println("[Converter] Anwendungslogik in Methode " +
               methode + " wird gestartet");
38          Object rueckgabe = ctx.proceed();
39          System.out.println("[Converter] Methode " + methode + " beendet.");
40
41          return rueckgabe;
42      }
43  }
```

Listing 22.11: Stateful Session Bean des Fremdwährung-EUR-Umrechners

Zur Demonstration eines AroundInvoke-Interceptor haben wir der Session Bean zusätzlich eine einfache Funktionalität spendiert, die den Aufruf und das Verlassen jeder Methode der Session Bean im Server-Log protokolliert. Zudem gibt auch die Remove-Methode einen Log-Eintrag aus. Sie unterscheidet sich aber von den anderen Logging-Methoden dadurch, dass sie nicht primär zur Log-Ausgabe, sondern zum Anfordern der Zerstörung des Objekts dient. Wichtig bei der Implementierung der Session Bean ist die Annotation `@Stateful`, die die Session Bean erst als eine Stateful Session Bean auszeichnet.

22.5.2. Webschicht (JavaServer Faces)

Da es sich in unserem letzten Beispiel um eine etwas komplexere Web-Anwendung handelt, erstellen wir gemäß Abbildung 22.5 drei JSF-Seiten. Eine Seite wird beim Start der Anwendung angezeigt und enthält lediglich einen Link, der die Benutzerin auf die Seite zum Einstellen der Währungsparameter weiterleitet (index.xhtml). In einer weiteren JSF-Seite können die Bezeichnung und der Umrechnungskurs einer Fremdwährung eingegeben und gespeichert werden (kurs.xhtml). Nachdem die Parameter der Fremdwährung gesetzt wurden, wird die Benutzerin auf eine Berechnungs-Seite weitergeleitet. Auf dieser werden, wie im vorherigen Beispiel, ein Betrag eingegeben und umgerechnet. Zusätzlich kann sich die Benutzerin hier abmelden.

Auf die Darstellung der JSF-Seiten verzichten wir hier.

22.5.3. Webschicht (Managed Bean)

Um die Verbindung zwischen der Benutzungsschnittstelle (den JSF-Seiten) und der Fachlogik (die Stateful Session Bean) herzustellen, wird wieder eine Managed Bean genutzt. Codeausschnitt 22.12 zeigt die Managed Bean `ConverterManagedBean` ohne die Darstellung aller Getter und Setter.

```
1  package kurs1796.ke6;
2
3  import java.math.BigDecimal;
4  import javax.ejb.EJB;
5  import javax.inject.Named;
6  import javax.enterprise.context.SessionScoped;
7
8  @Named
9  @SessionScoped
10 public class ConverterManagedBean
11 {
12     @EJB private Converter bean;
13
14     private BigDecimal eingabe;
15
16     private BigDecimal fwKurs;
17     private String fwBezeichnung;
18
19     private BigDecimal euroWert;
20     private BigDecimal fwWert;
21
22     public String setWaehrung() {
23         bean.setFwKurs(fwKurs);
24         bean.setFwBezeichnung(fwBezeichnung);
25
26         return "berechnung";
27     }
28
29     public void berechne() {
30         fwWert = bean.euroZuFw(eingabe);
31         euroWert = bean.fwZuEuro(eingabe);
32     }
33
34     public void abmelden()
35     {
36         bean.remove();
37     }
38
39     public ConverterManagedBean() {}
40
41    ... Getter und Setter für eingabe, fwKurs, fwBezeichnung, euroWert und
          fwWert ...
42 }
```

Listing 22.12: Managed Bean des Fremdwährung-EUR-Umrechners

In diesem Beispiel wird eine einzige Managed Bean für alle drei JSF-Seiten verwendet. Innerhalb der Managed Bean wird wieder eine Variable `bean` vom Typ `Converter` deklariert, der zur Laufzeit durch Dependency Injection (Annotation `@EJB`) eine Instanz der Klasse `Converter` zugewiesen wird. Die Methode `setWaehrung()` wird aus der JSF-Seite `kurs.xhtml` heraus aufgerufen und greift auf die Setter der Session Bean zu, um die Parameter der Fremdwährung zu

setzen. Weiterhin gibt sie als Rückgabewert die Zeichenkette `berechnung` zurück. Da der Methodenaufruf innerhalb eines EL-Ausdrucks an ein `action`-Attribut einer Schaltfläche gebunden ist, wird eine dynamische Navigation zur Seite `berechnung.xhtml` durchgeführt. Die Methode `berechne()` berechnet, wie im vorherigen Beispiel, den Euro-Wert sowie den Wert der Fremdwährung und wird innerhalb der Seite `berechnung.xhtml` durch die Benutzerin aufgerufen. Die Methode `abmelden()` wird bei Betätigung des Abmelde-Links aufgerufen und ruft die `remove`-Methode der Session Bean auf.

Schließlich soll noch einmal das Zusammenspiel von Managed Beans und Session Beans dargestellt werden, indem wir die in diesem Beispiel verwendeten Annotationen betrachten und uns klar machen, was zur Laufzeit bezüglich der vorhandenen Klassen genau passiert. Die Managed Bean `ConverterManagedBean` wurde in unserem Beispiel mit der Annotation `@SessionScoped` versehen, die Session Bean mit der Annotation `@Stateful`. Hier reicht es nicht aus, nur die Session Bean als `stateful` zu deklarieren und die Managed Bean im Request Scope zur Verfügung zu stellen. Wird die Web-Anwendung aufgerufen, so wird eine Instanz von `ConverterManagedBean` innerhalb des Session Scope erzeugt. Ebenfalls wird eine Instanz der EJB Converter in die Variable `bean` der Managed Bean injiziert. Diese Instanz existiert nur so lange, wie auch die Instanz der Managed Bean existiert, in diesem Fall für die Dauer der Session. Die Deklaration der Session Bean als Stateful Session Bean bedeutet, dass jedem Client eine Instanz der Bean zugewiesen wird.

Angenommen, wir hätten die Session Bean als Stateless Session Bean deklariert. Dann wählt ein Client A eine Instanz aus dem Pool aller Stateless Session Beans aus, um diese zu nutzen. Falls eine solche Instanz nicht existiert, wird sie erzeugt. Wird nun ein Client B aktiv, so wählt dieser ebenfalls zufällig eine Stateless Session Bean aus diesem Pool aus oder erzeugt eine neue. Durch die zufällige Auswahl von im Pool vorhandenen Beans kann es sein, dass Client B eine Bean erhält, in der die von Client A festgelegten Parameter für die Fremdwährung hinterlegt sind. Dies ist für unsere Beispielanwendung unerwünscht. Für Stateful Session Beans existiert ein solcher Pool nicht.

22.6. Implementierung einer Message-Driven Bean

Anders als eine Session Bean verfügt eine Message-Driven Bean nicht über ein Business Interface, sondern wird über einen *Nachrichtendienst* (Message Service) angesprochen. Dazu wird sie bei einer entsprechenden *Nachrichtenwarteschlange* (Message Queue) als Nachrichtenempfänger registriert (vgl. auch Abschnitt 22.1). Es können verschiedene Nachrichtendienste verwendet werden, um Message-Driven Beans anzusprechen. Am gängigsten ist der *Java Message Service* (JMS), denn JMS gehört zu den Java EE Standard Services und steht im Web Container und im EJB Container zur Verfügung.

Die Implementierung des Nachrichtenempfängers, also der Message-Driven Bean, ist von der Wahl des Nachrichtendienstes abhängig. Wir werden im Folgenden JMS verwenden. Eine für JMS implementierte Message-Driven Bean wird in [EJB/Spec] als *JMS Message-Driven Bean* bezeichnet.

Das Arbeiten mit einem Nachrichtendienst wie JMS stellt selbst wieder ein umfangreiches Thema dar, auf das wir hier nicht ausführlich eingehen. Wir konzentrieren uns stattdessen auf die Implementierung einer Message-Driven Bean und zeigen dabei nur beispielhaft die Verwendung des JMS.

22.6.1. Message-Driven Bean als JMS-Nachrichtenempfänger

Als JMS-Nachrichtenempfänger kann jede Klasse fungieren, die das Interface `javax.jms.MessageListener` implementiert. Dieses Interface exportiert eine einzige Methode, die von der Empfängerklasse zu implementieren ist und zur Übergabe der Nachricht durch den Nachrichtendienst dient. Sie hat folgende Signatur:

```
void onMessage(javax.jms.Message)
```

In dieser Methode ist die Reaktion auf die Nachricht zu implementieren. Natürlich darf die Empfängerklasse zusätzlich weitere Methoden implementieren, die ihrerseits durch die Methode `onMessage` aufgerufen werden können.

Um nun einen solchen Nachrichtenempfänger zu einer Message-Driven Bean zu machen, ist im Wesentlichen eine Annotation des Typs `javax.ejb.MessageDriven` an der Klasse anzubringen, wie in Codeausschnitt 22.13 zu sehen ist.

```
1  package kurs01796.ke6;
2  import javax.ejb.MessageDriven;
3  import javax.jms.Message;
4  import javax.jms.MessageListener;
5
6  @MessageDriven
7  public class AsyncALBean implements MessageListener {
8     private void meineAL(String s) {
9        ...
10    }
11    public void onMessage(Message message) {
12       ... Nachrichtenverarbeitung ...
13    }
14 }
```

Listing 22.13: Message-Driven Bean JMS

Wie die Verarbeitung einer JMS-Nachricht in der `onMessage`-Methode aussehen kann, werden wir in Abschnitt 22.7 an einer weiteren Variation des Währungsumrechner-Beispiels demonstrieren.

22.6.2. Anmeldung an einer Message Queue

Was nach der Implementierung der Message-Driven Bean noch fehlt, ist deren Anmeldung an einer Nachrichtenwarteschlange. Diese Aufgabe gehört laut [EJB/Spec] zwar in den Verantwor-

tungsbereich des *Deployers* und nicht des die Bean implementierenden und bereitstellenden *Bean Providers*. Jedoch kann der Bean Provider Vorschläge unterbreiten, und ein Application Server kann (muss jedoch nicht) in der Lage sein, diese Vorschläge auszuwerten und umzusetzen, so lange der Deployer sie nicht überstimmt.

Der Glassfish Application Server sieht hierzu folgende einfache Möglichkeit vor: Der Bean Provider kann im optionalen Parameter `mappedName` des `MessageDriven`-Annotationstyps den JNDI-Namen der Warteschlange angeben, von der die Bean mit Nachrichten versorgt werden soll (vgl. Codeausschnitt 22.14). Es sei aber betont, dass dies eine produktspezifische Notation ist[14] , die nicht von jedem Application Server unterstützt wird.

```
1 @MessageDriven(mappedName="jms/1796/meineQueue")
2 public class AsyncALBean implements MessageListener {
3     ...
4 }
```

Listing 22.14: mappedName-Parameter in einer Message-Driven-Annotation

22.6.3. Lebenszyklus-Callback-Methoden und AroundInvoke-Interceptor

Abgesehen von ihrer asynchronen Benachrichtigung über Nachrichten ist eine Message-Driven Bean in vielerlei Hinsicht mit einer Stateless Session Bean vergleichbar. Von beiden werden durch den Container je nach Bedarf Instanzen erzeugt oder zerstört, die Instanzen werden in einem Pool gehalten und pro Anfrage dynamisch Clients zugeordnet. Dementsprechend gleicht sich ihr Lebenszyklus: Das Zustandsdiagramm aus Abbildung 22.2 beschreibt auch den Lebenszyklus einer Message-Driven Bean.

Aufgrund dieser Vergleichbarkeit können die in Abschnitt 22.2 für Stateless Session Beans vorgestellten Lifecycle Callback Methoden auch für Message-Driven Beans implementiert werden. Auch die für Session Beans vorgestellte AroundInvoke-Interceptor-Methode kann für Message-Driven Beans implementiert werden. Während diese Methode bei Session Beans sämtliche Aufrufe von Business-Methoden (über das Business-Interface exportierte Methoden) abfängt, wird bei JMS Message-Driven Beans jeder Aufruf der `onMessage`-Methode abgefangen, nicht jedoch die durch onMessage ausgelösten Sub-Aufrufe von Methoden, wie z.B. `meineAL` aus Codeausschnitt 22.12.

[14] Genauer: Der Parameter mappedName selbst wird durch den Annotationstyp MessageDriven definiert und ist somit nicht produktspezifisch, wohl aber dessen Auswertung, also ob und ggf. wie der Parameter interpretiert wird.

22.7. Verwendung einer Message-Driven Bean

22.7.1. Aufruf der Anwendungslogik durch Senden einer Nachricht

Jede Message-Driven Bean ist, wie bereits erläutert, ein an einer Nachrichtenwarteschlange registrierter Nachrichtenempfänger. Um sie anzusprechen und somit zur Ausführung ihrer Anwendungslogik zu veranlassen, ist eine Nachricht an diese Warteschlange zu senden. Dabei ist es für den Client irrelevant, ob es sich bei dem Nachrichtenempfänger überhaupt um eine EJB handelt oder um irgendeinen anderen Nachrichtenempfänger.

Falls eine Message-Driven Bean immer dieselbe Aktion ausführen soll und dazu auch keine Eingabeparameter erwartet, genügt eine einfache leere Nachricht, um diese Aktion auszulösen. Erwartet sie jedoch Eingaben oder soll sie gar mehrere alternative Aktionen anbieten, so müssen alle benötigten Informationen als Nachrichteninhalt übermittelt werden. Der vom Empfänger erwartete Aufbau einer Nachricht, also seine Eingabeschnittstelle, sollte gut dokumentiert werden, z.B. im `JavaDoc`-Kommentar der Bean-Klasse.

JMS bietet eine Reihe verschiedener Nachrichtentypen an (Subtypen von `javax.jms.Message`), die z.B. einen String, einen Stream[15] oder ein Objekt zum Inhalt haben können. In Codeausschnitt 22.15 wird eine Nachricht des Typs `TextMessage` versendet, die einen String als Inhalt aufnehmen kann. Dieser Nachrichteninhalt wird als *Message Body* bezeichnet. Neben dem Message Body können den JMS-Nachrichten zusätzlich so genannte *Message Properties* hinzugefügt werden, die jeweils ein Tripel aus einem Namen, einem Datentyp (Basisdatentyp oder String) und einem Wert dieses Typs darstellen. In Codeausschnitt 22.15 wird eine Integer-Property namens `answer` mit Wert 42 zur Nachricht hinzugefügt.

```
1  import javax.annotation.Resource;
2  import javax.jms.*;
3  import ...
4
5  @Named
6  @RequestScoped
7  public class MyManagedBean
8     @Resource(...) Queue queue;
9     @Resource(...) ConnectionFactory factory;
10
11    public void senden() {
12       try {
13          //Vorbereitungen zum Versenden der Nachricht
14          Connection conn = factory.createConnection();
15          Session session = conn.createSession(false, Session.
               AUTO_ACKNOWLEDGE);
16          MessageProducer prod = session.createProducer(queue);
17          //Senden der Nachricht
18          TextMessage msg = session.createTextMessage("The Ultimate Question
                of Life, the " + "Universe and Everything");
```

[15] engl. stream = Strom, Fluss bzw. Datenstrom im technischen Sinne

```
19            msg.setIntProperty("answer", 42);
20            prod.send(msg);
21        } catch (JMSException e) {
22            ...
23        }
24    }
```

Listing 22.15: Versenden einer JMS-Nachricht (Beispiel)

22.7.2. Verbindung zur Message Queue

Codeausschnitt 22.15 zeigt eine Managed Bean als Beispiel für eine Client-Komponente, die eine Message-Driven Bean ansprechen will. Zum Absenden einer JMS-Nachricht an eine JMS Message Queue werden zunächst zwei Ressourcen benötigt: Die Warteschlange selbst sowie eine zur Herstellung der Verbindung zu dieser Warteschlange benötigte Connection Factory. Der Codeausschnitt demonstriert, wie mit Hilfe dieser beiden Ressourcen die Verbindung zur Queue hergestellt und die Nachricht verschickt wird. Auf Details zu JMS gehen wir nicht weiter ein, da das zu weit vom eigentlichen Thema wegführen würde. Referenzen auf diese Ressourcen können wahlweise injiziert werden (sofern Dependency Injection überhaupt möglich ist, wie bei obiger Managed Bean) oder per JNDI-Lookup ermittelt werden. Zur Injektion der Ressourcen dient der Annotationstyp `javax.annotation.Resource`.

Damit die Ressourcen jedoch injiziert oder per Lookup gesucht werden können, müssen sie zunächst vom Administrator des Application Servers eingerichtet und über JNDI verzeichnet und zur Verfügung gestellt werden. Geht man davon aus, dass diese Installation nach der Entwicklung von Bean und Client stattfindet, obliegt es auch hier – wie schon bei der im letzten Abschnitt angesprochenen Anmeldung der Bean an der Message Queue – dem Deployer, die Verbindung des Clients zu Queue und Factory zu konfigurieren. So lässt sich z.B. auch die Dependency Injection über einen Deployment Descriptor steuern. Aber auch hier kann die Entwicklerin direkt im Programmcode die gewünschte Verbindung vorkonfigurieren, wobei die Notation (zumindest bei der Dependency Injection) wieder produktspezifisch ist und im Folgenden in einer Fassung für den Glassfish Application Server demonstriert wird.

Angenommen, die Message Queue sei unter dem JNDI-Namen `jms/meineQueue` und die ConnectionFactory unter `jms/meineCF` verzeichnet. Dann können die beiden Injektionen in Codeausschnitt 22.15 wie folgt komplettiert werden:

```
@Resource(mappedName="jms/meineQueue") Queue queue;
@Resource(mappedName="jms/meineCF") ConnectionFactory factory;
```

Alternativ kann ein JNDI-Lookup nach folgendem Muster verwendet werden:

```
InitialContext ic = new InitialContext();
queue = (Queue)ic.lookup("jms/meineQueue");
factory = (ConnectionFactory)ic.lookup("jms/meineCF");
```

22.7.3. Rückmeldungen von der Bean zum Client

Die einzige Rückmeldung, die ein Client nach dem Senden einer Nachricht direkt erhält, informiert ihn darüber, ob der Nachrichtenversand an die Message Queue funktioniert hat (falls nicht, wird eine Exception[16] ausgelöst). Danach arbeitet der Client parallel weiter, während die Nachricht letztendlich von der Queue an die Message-Driven Bean zugestellt und von dieser verarbeitet wird. Es existiert also kein direkter Rückkanal von der Bean zum Client – das ist der Preis für die asynchrone Ausführung der Anwendungslogik.

Zwar gibt es Beispiele für Anwendungslogik, in denen ein solcher Rückkanal auch gar nicht benötigt wird, eine Message-Driven Bean könnte z.B. eine langwierige Operation auf einer Datenbank oder eine Datensicherung durchführen. Am Ende einer solchen Aktion wird ggf. noch ein Eintrag in ein Log-File geschrieben, der vermerkt, ob der Vorgang erfolgreich beendet wurde oder fehlgeschlagen ist. Sollte der Client aber doch Rückmeldungen benötigen, kann er z.B. zum Sammeln von Informationen parallel dazu die Log-Einträge inspizieren oder den Datenbankstatus überwachen. Für Rückmeldungen, die ausschließlich an den Client gerichtet sind (also nicht in das Server-Log gehören) und die auch nicht in einer Datenbank gespeichert werden sollen, bietet sich nach demselben Prinzip das Anlegen eines speziellen temporären Verzeichnisses an, in das die Bean ihre Rückmeldungen schreibt und aus dem der Client diese Informationen auslesen kann.

Beim Hinterlegen von Informationen für Clients ist zu beachten, dass mehrere Instanzen einer Message-Driven Bean parallel für verschiedene Clients arbeiten können und somit Informationen für mehrere Clients gesammelt werden. Es wird daher ein Mechanismus benötigt, der einem Client ermöglicht, genau die für ihn bestimmten Rückmeldungen abzurufen. Das im folgenden Abschnitt angegebene Beispiel verwendet hierzu einen einfachen (optimistischen) *Zeitstempel*-Mechanismus, der beim Senden einer Nachricht den aktuellen Zeitstempel (Timestamp) im Client speichert und mit in die Nachricht integriert. Die Message-Driven Bean, die auf die Nachricht reagieren soll, erstellt ein Ergebnis-Objekt ihrer Berechnung, das den erhaltenen Zeitstempel enthält. Nun kann der Client anhand des Zeitstempels eines Ergebnis-Objektes überprüfen, ob es sich um das Ergebnis-Objekt seiner Berechnungs-Anfrage handelt. Dieser Mechanismus ist sehr optimistisch, da es passieren kann, dass zwei Clients den gleichen Zeitstempel verwenden.

Man könnte auch auf die Idee kommen, die Rückmeldungen der Bean wieder über den Nachrichtendienst an den Aufrufer zurückzuschicken: Ein Nachrichtenempfänger wie die Message-Driven Bean kann auch selbst wieder Nachrichten an eine Warteschlange senden und auf der Clientseite könnte ein Nachrichtenempfänger vorhanden sein, der die asynchron eintreffenden Antwortnachrichten verarbeitet, indem er z.B. ein Ereignis auslöst. Genau hier zeigt sich aber eines der Probleme dieses Vorschlags: Wir betrachten hier Web-Anwendungen, deren Controller aus Web-Komponenten wie z.B. Servlets oder Managed Beans bestehen, die ggf. das Model (in diesem Fall eine Message-Driven Bean) ansprechen. Der Client einer Message-Driven Bean wäre also im Regelfall eine solche Web-Komponente, und diese kann lediglich auf Requests des Web-Clients reagieren. Hat sie einmal ihre Response an den Web-Client zurückgeschickt, kann

[16] engl. exception = Ausnahme, Ausnahmefall

sie mit nachträglich asynchron eintreffenden Informationen der Message-Driven Bean nichts anfangen, sie kann nicht ihrerseits asynchron den Web-Client nachträglich kontaktieren. Vielmehr erfolgt die Kommunikation zwischen Web-Client und Web-Server ausschließlich synchron (von moderneren DHTML- und Ajax-Ansätzen einmal abgesehen) und Client-gesteuert. Die einzige Möglichkeit des Web-Clients, nachträglich Informationen zu einem früher asynchron gestarteten Vorgang zu bekommen, ist das so genannte Polling, d.h. regelmäßiges aktives Nachfragen, ob inzwischen Ergebnisse (z.B. im Log, einer Datenbank oder einem wie oben beschriebenen Rückmeldungsverzeichnis) vorliegen.

22.8. Beispiel: Währungsumrechner mit Message-Driven Bean

In diesem Abschnitt übertragen wir das frühere Beispiel eines Dollar-zu-Euro-Währungsumrechners (Abschnitt 22.4) auf Message-Driven Beans. Die Anwendungslogik zur Währungsumrechnung soll also nicht mehr in einer Session Bean, sondern in einer Message-Driven Bean implementiert werden. Eine Managed Bean soll eine Nachricht zum Aufruf der Anwendungslogik senden, welche das Berechnungsergebnis in einem zentralen Verzeichnis ablegt, wo es später im Rahmen eines Pollings wieder ausgelesen wird. Um die parallele Ausführung der Anwendungslogik und der Web-Schicht zu demonstrieren, soll eine längerwierige Berechnung simuliert werden, sodass nicht gleich beim ersten Polling-Versuch schon ein Ergebnis vorliegt, sondern wiederholt nachgefragt werden muss. Während der Wartezeit soll eine Information angezeigt werden. Abbildung 22.6 zeigt die Benutzungsschnittstelle dieses Währungsumrechners.

22.8.1. Ergebnisrückgabe

Die Message-Driven Bean bekommt in der eingehenden Nachricht die Eingabe der Nutzerin übergeben und berechnet daraus die beiden Ergebniswerte. Diese Berechnungsergebnisse werden in einer globalen Ergebnisliste hinterlegt, aus der der Client sie per Polling wieder auslesen kann. Da die Liste Ergebnisse mehrerer Clients enthalten kann, werden die Einträge mit einer vom Client generierten Identifikationsnummer versehen, anhand derer der Client die für ihn bestimmten Listeneinträge erkennen kann. In diesem Beispiel wird als Identifikationsnummer ein Timestamp verwendet, den ein Client vor dem Senden seiner Nachricht ermittelt[17].

Zur Umsetzung der Ergebnisliste dienen zwei Klassen: `ErgebnisEintrag` implementiert einen Eintrag in die Ergebnisliste, der den Timestamp, die Eingabe und die beiden Umrechnungsergebnisse aufnimmt. `ErgebnisListe` erstellt eine Liste aus solchen Einträgen und bietet Methoden

[17] Für den nicht sehr wahrscheinlichen Konfliktfall, dass zwei Clients einmal denselben Timestamp verwenden (was z.B. bei späterer Verteilung der Anwendung oder bei Mehrprozessorrechnern vorkommen kann) und somit ein Client das falsche Ergebnis aus der Liste abruft, könnte zu jedem Ergebnis zusätzlich die ursprüngliche Eingabe mit gespeichert werden, sodass das angezeigte Ergebnis in jedem Fall konsistent ist und höchstens nicht zur Anfrage passt.

Abbildung 22.6.: Benutzungsschnittstelle des Message-Driven Währungsumrechners

zum Hinzufügen neuer Einträge und zum Suchen eines Eintrags anhand eines Timestamps als Suchkriterium. Beide Klassen finden sich im Paket `kurs1796.ke6.converter.util`.

22.8.2. Nachrichtenformat

Wie im letzten Abschnitt betont wurde, muss der Aufbau einer Nachricht, wie er von der Bean erwartet wird und somit vom Absender einzuhalten ist, festgelegt und dokumentiert werden. In unserem Beispiel gelten folgende Vereinbarungen:

- Die Nachricht muss vom Typ `ObjectMessage` sein.
- Das den Message Body bildende Objekt muss vom Typ `BigDecimal` sein und die Benutzereingabe enthalten.
- Neben dem Message Body ist eine Message Property vom Typ long mit dem Namen `timestamp` zu übertragen, welche den Timestamp zur Identifikation des Ergebniseintrags enthält.

22.8.3. Message-Driven Bean

Nach den Festlegungen zur Ein- und Ausgabeschnittstelle (Nachrichtenformat und Ergebnisliste) kann die Bean-Klasse implementiert werden. Dazu übernehmen wir zunächst die bereits aus dem ersten Beispiel (Abschnitt 22.4) bekannten Methoden zur Umrechnung. Da diese jedoch nicht mehr über ein Business-Interface exportiert, sondern nur intern von der Nachrichtenbehandlung (Methode `onMessage`) verwendet werden, sind sie nicht mehr öffentlich, sondern als private deklariert. Die `onMessage`-Methode entnimmt der eingegangenen Nachricht die Benutzereingabe und ruft nacheinander beide Umrechnungsmethoden auf. Anschließend werden die Ergebnisse samt Timestamp in der Ergebnisliste abgelegt. Codeausschnitt 22.16 zeigt die Implementierung der Bean-Klasse.

```
1  package kurs1796.ke6.converter.ejb;
2  import kurs1796.ke6.converter.util.ErgebnisEintrag;
3  import kurs1796.ke6.converter.util.ErgebnisListe;
4  import ...
5
6  //Annotation spezifisch für Glassfish Application Server:
7  @MessageDriven(mappedName="jms/1796/converterBeanQueue")
8  public class ConverterBean implements MessageListener {
9     private final static BigDecimal EUROKURS = new BigDecimal(1.3792);
10
11    private BigDecimal dollarZuEuro(BigDecimal dollar) {
12       return dollar.divide(EUROKURS,2,BigDecimal.ROUND_HALF_UP);
13    }
14
15    private BigDecimal euroZuDollar(BigDecimal euro) {
16       BigDecimal dollar = euro.multiply(EUROKURS);
17       return dollar.setScale(2, BigDecimal.ROUND_HALF_UP);
18    }
19
20    public void onMessage(Message message) {
21       try {
22          //Nachricht auslesen.
23          ObjectMessage msg = (ObjectMessage)message;
24          long timestamp = msg.getLongProperty("timestamp");
25          BigDecimal eingabe = (BigDecimal)msg.getObject();
26
27          //Berechnung durchführen:
28          BigDecimal dollar = euroZuDollar(eingabe);
29          BigDecimal euro = dollarZuEuro(eingabe);
30
31          //Langwierige Berechnung simulieren:
32          Thread.sleep(7000);
33
34          //Ergebnis speichern
35          ErgebnisListe.addEintrag(
36                  new ErgebnisEintrag(timestamp,eingabe,dollar,euro));
37       } catch (Exception e) {
38          ...
39       }
```

```
40     }
41 }
```

Listing 22.16: Message-Driven Bean des Dollar-Euro-Umrechners

22.8.4. Web-Schicht

Zur Umsetzung der Web-Schicht wird eine einzige JSF-Seite verwendet, auf der entsprechend des aktuellen Zustandes der Anwendung (Eingabe, Warten auf Berechnungsergebnis, Ergebnis vorhanden) Komponenten dargestellt oder nicht dargestellt werden. In dieser Beispielanwendung realisiert JavaScript das Polling, d.h. den regelmäßigen Abruf von Berechnungsergebnissen. Im Hintergrund der JSF-Seite befindet sich eine Managed Bean, die den von der Benutzerin eingegebenen Wert speichert. In diesem Beispiel enthält die Managed Bean den Programmcode zum Abrufen eventuell vorhandener Berechnungsergebnisse aus der Ergebnisliste.

22.8.5. Installation

Zunächst ist zu beachten, dass dieses Beispiel ausschließlich lokal auf einem Server betrieben werden kann, also die EJB und die Web-Schicht nicht auf verschiedene Server verteilt werden können. Zwar funktioniert der Nachrichtenaustausch durchaus über Servergrenzen hinweg (vgl. Abschnitt 22.1), jedoch wird unser zentrales Rückmeldungsverzeichnis in diesem Beispiel von Bean und Web-Schicht jeweils lokal angesprochen.

Neben dem Deployment der EJB- und Web-Komponenten auf einem Application Server ist außerdem die Bereitstellung der für den Nachrichtenaustausch benötigten Ressourcen (Message Queue und dazugehörige Connection Factory) vonnöten. Falls diese unter den JNDI-Namen `jms/1796/converterBeanQueue` bzw. `jms/1796/queueConnectionFactory` bereitgestellt sind, kann die Beispielanwendung auf einem Glassfish Application Server ohne weitere Konfigurationsarbeit in Betrieb genommen werden, da diese beiden JNDI-Namen in der EJB und in der Managed Bean bereits in den Annotationen hinterlegt sind.

22.9. Verzeichnisstrukturen für das Deployment

In diesem Kapitel wurden erstmals so genannte Enterprise Applications erstellt. Diese sind hier Anwendungen aus einer EJB-Komponente und einer Web-Anwendungs-Komponente, die in getrennten Projekten erstellt werden können.

Das Projekt für die Web-Schicht ist wie jedes allein stehende Webanwendungsprojekt aufgebaut. Die dafür vorgegebene Verzeichnisstruktur, die zum Deployment in einem WAR (Web Archive) gebündelt werden kann, wurde bereits in Kurseinheit 3 vorgestellt.

Zum Deployment von EJBs gibt es ebenfalls eine Vorgabe für eine Verzeichnisstruktur, die wesentlich simpler aufgebaut ist (vgl. Tabelle 22.2): In dem Stammverzeichnis des EJB-Projekts werden die EJB-Klassen und ggf. weitere von diesen verwendeten Klassen zusammengefasst. Deployment-Deskriptoren gehören ggf. ins Verzeichnis META-INF, in unseren Beispielen benötigten wir jedoch keine solchen Deskriptoren, sondern haben alle für das Deployment benötigten Metainformationen in Annotationen hinterlegt.

Ein solches EJB-Teilprojekt kann zum Deployment als JAR-Archiv (dann als EJB-JAR bezeichnet) zusammengefasst werden.

Verzeichnis	*Inhalt*
/Name	Der Name des EJB-Teilprojekts ist zugleich das Wurzelverzeichnis. Bei Distribution als EJB-JAR tritt der Name nicht in der Verzeichnisstruktur auf, sondern wird als Name für das Archiv herangezogen. Die EJB-Klassen sowie ggf. weitere von diesen benutzte Klassen sind entsprechend ihrer Paketstruktur in Unterverzeichnisse einzugliedern.
/Name/META-INF/	Enthält Metadaten über das EJB-JAR-Archiv. Insbesondere kann hier ein gesonderter Deployment Descriptor für das EJB-JAR (namens `ejb-jar.xml`) abgelegt werden.

Tabelle 22.2.: Verzeichnisstruktur eines EJB-Moduls

Die auf einem Application Server zu installierenden Komponenten (wie EJBs und Web-Schicht) müssen nicht separat ausgeliefert werden, sondern können (vom Application Assembler) zu einem *Enterprise ARchive* (EAR) zusammengefasst werden, dessen Struktur in Tabelle 22.3 zusammengefasst ist. Dieses kann dann vom Deployer in einem Schritt auf dem Server installiert werden.

Verzeichnis	*Inhalt*
/Name	Im Stammverzeichnis des EARs werden die einzelnen Modul-Archive wie EJB-Jars und WARs abgelegt. Auch weitere von diesen verwendete JAR-Files dürfen hier stehen.
/Name/lib/	Alle hier abgelegten JAR-Bibliotheken werden allen Anwendungskomponenten gemeinsam zur Verfügung gestellt.
/Name/META-INF/	Hier kann insbesondere ein gesonderter Deployment-Descriptor für die Enterprise Application (namens `application.xml`) abgelegt werden.

Tabelle 22.3.: Verzeichnisstruktur eines EARs

Falls in einem EAR-Archiv kein `application.xml`-Deployment Descriptor hinterlegt wird, sucht sich der Container beim Deployment die benötigten Informationen anderswo. So interpretiert er jedes enthaltene WAR-Archiv als Web-Modul, und ein JAR-Archiv wird als EJB-JAR

interpretiert, wenn es `class`-Dateien mit entsprechenden Annotationen enthält. Andere JAR-Archive, die keine EJBs enthalten, werden als Hilfsbibliotheken installiert, sofern sie im `lib`-Verzeichnis liegen oder sofern der Container feststellt, dass sie von Anwendungskomponenten wie der WebSchicht oder EJBs verwendet werden.

Diese Seite bleibt aus technischen Gründen frei!

Kurseinheit 7

Java EE: Entity-Klassen und Entwurfsmuster

Nachdem in Kurseinheit sechs die Umsetzung der Anwendungslogik mit Hilfe von Enterprise JavaBeans behandelt wurde, widmet sich die vorliegende, letzte Kurseinheit dem Teil des Anwendungskerns, der aus (persistenten) Anwendungsobjekten besteht.

Die Kurseinheit beginnt mit einer Einführung in die Realisierung und Verwendung persistenter Anwendungsobjekte, so genannter Entities. Diese werden mit Hilfe der Java Persistence API transparent und komfortabel auf Datenbanktabellen abgebildet.

Den Abschluss bildet eine Vorstellung ausgewählter Entwurfsmuster, die insbesondere eine sinnvolle Anbindung des Anwendungskerns an die Web-Schicht beschreiben.

Diese Seite bleibt aus technischen Gründen frei!

23. Entity-Klassen

23.1. Einführung

Nachdem in den Kurseinheiten drei und fünf in die Implementierung der Web-Schicht mit Java EE und JavaServer Faces sowie im letzten Kapitel die Implementierung der Anwendungslogik mit Hilfe von Enterprise JavaBeans behandelt wurden, geht es nun mit den persistenten Anwendungsobjekten um die letzte selbst zu implementierende Schicht der 6-Schichten-Architektur. Anwendungsobjekte sind vergleichbar mit Instanzen von Entitätsklassen, wie sie in [SE 1] verwendet wurden.

Im Folgenden verwenden wir einige (teilweise bereits in [SE 1] benutzte) Begriffe, die hier zunächst eingeführt werden sollen. Als *transientes Objekt* bezeichnen wir ein flüchtiges, d.h. ausschließlich im Arbeitsspeicher und nicht in der Datenbank gespeichertes Objekt, das nach Beenden der Software nicht mehr existiert. Unter einem *persistenten Objekt* wird dagegen ein Objekt verstanden, das durch Speicherung seines Zustandes in einer Datenbank nicht flüchtig ist. Zu einem persistenten Objekt existiert genau eine *externe Repräsentation* in Form eines Datenbankeintrags und höchstens eine *interne Repräsentation* in Form eines Objekts im Arbeitsspeicher. Wenn beide Repräsentationen existieren, sind sie *synchron*, d.h. Änderungen an der internen Repräsentation des persistenten Objekts wirken sich auch auf die externe Repräsentation aus. Objekte im Arbeitsspeicher, die zwar denselben Zustand wie ein persistentes Objekt haben, jedoch nicht mit dessen externer Repräsentation synchronisiert sind, sondern ohne Auswirkungen auf das persistente Objekt modifiziert werden können, bezeichnen wir als *transiente Kopie* des persistenten Objektes.

Als *persistente Klasse* wird eine Java-Klasse bezeichnet, deren Objekte persistent sein können. Der Vorgang, ein noch transientes Objekt einer persistenten Klasse zu einem persistenten Objekt zu machen, indem eine externe Repräsentation neu angelegt wird, heißt *persistieren des transienten Objektes*.

Zur Umsetzung der Persistenz von *Anwendungsobjektklassen* (AO-Klassen oder auch Entitätsklassen) verwenden wir im Folgenden die in Java EE angebotene Java Persistence API. Diese ist aus der EJB 2.x-Spezifikation hervorgegangen – bis dahin gab es neben Session Beans und Message-Driven Beans (zur Implementierung der Anwendungslogik) noch die so genannten *Entity Beans* zur Implementierung von Entitätsklassen für persistente Anwendungsobjekte. Mittlerweile ist diese Persistenzschnittstelle unabhängig von den EJBs, und die ehemals als Entity Beans bezeichneten Komponenten heißen jetzt nur noch schlicht *Entities*. Die Spezifikation der Java Persistence API ist zwar noch ein Teil der EJB-Spezifikation Version 3.2 [EJB/Spec],

jedoch bildet sie dort bereits ein gesondertes Teildokument [JPA/Spec] und wird voraussichtlich zukünftig komplett unabhängig von der EJB-Spezifikation veröffentlicht. Java EE stellt die Persistence API im EJB- und Web-Container zur Verfügung. Inzwischen ist die Persistence API auch mit Java SE, unabhängig von Java EE, nutzbar.

Die Java Persistence API ermöglicht es, Java-Klassen zu implementieren, deren Objekte in einem *relationalen Datenbanksystem* persistent gehalten werden können. Zum transparenten Speichern und Laden von Objekten solcher Klassen zur Laufzeit bindet der die API implementierende Container ein Persistenz-Framework wie z.B. Hibernate [Hiber] (vom JBoss Application Server [JBoss] verwendet) oder Oracle TopLink Essentials [OTLE] (vom Oracle Application Server [GFish] verwendet) als so genannten *Persistence Provider* ein. Die vom Persistence Provider für die Abbildung der persistenten Klassen auf Datenbank-Relationen benötigten Informationen werden mit Hilfe von Annotationen[1] in den Klassen hinterlegt.

Mit Hilfe der Java Persistence API implementierte persistente AO-Klassen seien im Folgenden als *Entity-Klassen*, ihre Instanzen als *Entity-Instanzen* oder kurz *Entities* bezeichnet. Wir übernehmen bewusst das englische Wort „Entity“ aus [JPA/Spec] zur Abgrenzung der plattformspezifischen Implementierung von den als Anwendungsobjekt oder Entität bezeichneten Objekten plattformunabhängiger Entwurfsklassen. Dementsprechend werden als Entities realisierte persistente oder transiente Objekte im Folgenden auch als persistente bzw. transiente Entities bezeichnet.

23.2. Erstellen einfacher Entity-Klassen

Die Implementierung einer Entity-Klasse verläuft ähnlich zu der einer EJB: Es wird eine einfache Java-Klasse erstellt und mit einer Annotation als Entity-Klasse ausgezeichnet. Weitere benötigte Informationen werden durch zusätzliche Annotationen an Attributen oder Methoden hinterlegt.

Um als Entity-Klasse fungieren zu können, muss die Implementierung einer Entity-Klasse zunächst einige Mindestvoraussetzungen erfüllen. Die Klasse darf nicht als `final` deklariert sein, damit der Container des Anwendungsservers eigene Hilfsklassen davon ableiten kann. Ebenso dürfen die Methoden und der zu persistierende Zustand (das heißt die Menge der Attributwerte) nicht `final` sein. Die Klasse muss einen Standardkonstruktur (parameterlosen Konstruktor) besitzen, welcher als `public` oder `protected` ausgezeichnet sein muss, sodass der Container die internen Repräsentationen zu persistenten Entities selbst erzeugen kann. Zusätzliche Konstruktoren für die manuelle Erzeugung neuer transienter Entities sind natürlich erlaubt. (Es sei daran erinnert, dass, wenn kein einziger Konstruktor explizit implementiert wird, Java implizit einen Standardkonstruktur bereitstellt.) Zudem kann es sinnvoll sein, eine Entity-Klasse serialisierbar zu implementieren, d.h. die Klasse sollte das Interface `java.io.Serializable` implementieren. Nur dann ist es z.B. möglich, ihre Instanzen als Parameter über ein Remote

[1] Wie schon bei EJBs gilt auch hier, dass alternativ zu Annotationen auch Deployment-Deskriptoren verwendet werden können. Wie werden im Folgenden aber nur die Annotationsvariante erläutern.

Interface an Methoden von Session Beans oder als Nachrichteninhalt an Message-Driven Beans zu übergeben. Die Serialisierbarkeit der Klasse ist jedoch keine Voraussetzung für deren Persistierung.

Eine Klasse, die alle Voraussetzungen erfüllt, wird durch eine Annotation des Typs `javax.persistence.Entity` zu einer Entity-Klasse. Zusätzlich muss noch ein Schlüssel definiert werden (siehe 23.2.2)

23.2.1. Property-based oder Field-based Access

Java EE bietet zwei Varianten der Abbildung einer Entity-Klasse auf ein Datenbankschema, wie es zur Speicherung der externen Repräsentationen persistenter Entities benötigt wird. Zum ersten können direkt die Attribute (Fields) einer Entity-Klasse auf Datenbankfelder abgebildet werden, was als *Field-based Access* bezeichnet wird. Die zweite Variante sieht vor, die Properties der Entity-Klasse auf Datenbankfelder abzubilden. Eine *Property* einer Entity-Klasse ist dabei im Wesentlichen analog zu einer Property einer JavaBean definiert, bestimmt sich also aus Gettern und Settern. Die beim Persistieren in die Datenbank zu schreibenden Werte werden bei dieser Variante über die Getter ermittelt, und beim Erstellen der internen Repräsentation zu einer persistenten Entity werden die aus den Datenbankfeldern gelesenen Werte mit Hilfe der Setter in das Objekt eingetragen. Diese Variante nennt sich *Property-based Access*.

Property-based Access hat im Wesentlichen den Vorteil, dass die Kapselung der privaten Attribute bis auf Datenbankebene beibehalten wird, während Field-based Access private Interna in die Datenbank exportiert, also die Kapselung durchbricht[2] – zumindest wenn man die Datenbank als öffentlich zugänglich ansieht. Eine Konsequenz ist, dass bei nachträglichen Änderungen der privaten Implementierung einer Entity, bei denen aber die öffentliche Schnittstelle identisch bleibt, das Datenbankschema bei Property-based Access nicht beeinflusst wird. Da bei Auslesen einer externen Repräsentation aus der Datenbank die Werte wieder über die Setter in die interne Repräsentation eingetragen werden, wird sogar sichergestellt (entsprechende Implementierung der Setter vorausgesetzt), dass die interne Repräsentation selbst dann wieder einen konsistenten Zustand annimmt, wenn die externe Repräsentation (z.B. aufgrund manuellen Eingriffs in die Datenbank) einen inkonsistenten Zustand haben sollte.

Welche Abbildungsvariante zum Einsatz kommen soll, wird nicht explizit deklariert, sondern implizit festgelegt, mit Hilfe der in den nachfolgenden Abschnitten vorgestellten Annotationen, wie z.B. der `Id`-Annotation. Werden diese Annotationen an den Attributen der Entity-Klasse angebracht, so wird Field-based Access verwendet, während Property-based Access zum Einsatz kommt, wenn die Annotationen an den Gettern der Properties angebracht werden.

[2] Hierbei gehen wir vom Regelfall aus, dass die Attribute als *private*, die Properties als *public* deklariert sind. Die Attribute sind also verkapselt, der Zustand der Entity kann nur über die Schnittstelle, ganz oder teilweise bestehend aus den öffentlichen Gettern und Settern, gelesen oder verändert werden.

23.2.2. Schlüssel

Während Objekte im Speicher durch Speicheradressen (Zeiger) referenziert werden können, müssen Datensätze in einer relationalen Datenbank durch einen Schlüsselwert eindeutig identifiziert werden, der – im Gegensatz zu einem Zeiger *auf* ein Objekt – Bestandteil des Datensatzes ist. Damit eine persistente Entity auf einen identifizierbaren Datensatz in einer solchen Tabelle abgebildet werden kann, wird der Schlüssel sowohl in der externen Repräsentation (Datensatz) als auch in der internen Repräsentation (Java-Objekt) gespeichert. Die Entity-Klasse muss hierfür zunächst ein Attribut oder eine Property zur Aufnahme des Schlüsselwertes implementieren. Zur expliziten Auszeichnung als Schlüsselfeld wird eine Annotation des Typs `javax.persistence.Id` am Attribut (bei Field-based Access) bzw. an der Getter-Methode der Property (bei Property-based Access) angebracht.

Wir verwenden im Folgenden ausschließlich Property-based Access, aus Gründen der Kapselung und der besseren Wartbarkeit.

```
1  @Entity
2  public class Rechnung implements Serializable {
3     private long reNr;
4     ...
5     @Id
6     public long getReNr() {
7        return reNr;
8     }
9     public void setReNr(long reNr) {
10       this.reNr = reNr;
11    }
12    ...
13 }
```

Listing 23.1: Einfache Entity-Klasse mit manuell zu setzendem Schlüssel

Codeausschnitt 23.1 zeigt als Beispiel einen Auszug aus einer Entity-Klasse zur Speicherung einer Rechnung. Als Schlüssel wurde die Rechnungsnummer bestimmt, die also für jede Rechnung eindeutig sein muss, und durch Platzierung der `Id`-Annotation am Getter der Rechnungsnummer-Property wurde für diese Entity-Klasse die Zugriffsmethode Property-based Access eingestellt. Das Beispiel geht davon aus, dass die Rechnungsnummer nicht leer sein darf und eindeutig ist, also eine Rechnung identifiziert und somit als Schlüssel geeignet ist.

In den meisten Fällen wird eine Entity nicht „von Hause aus“ einen solchen Schlüssel mitbringen. Gegebenenfalls ist eine zusätzliche Property zur Aufnahme eines Schlüssels einzuführen, die lediglich für die Persistierung technisch notwendig, jedoch für die Anwendung uninteressant ist. Die Werte einer solchen Schlüssel-Property sind dann vergleichbar mit Speicheradressen von Objekten, die zwar technisch zur Referenzierung verwendet werden, aber nicht zum Inhalt der Objekte beitragen. So wie Speicheradressen vom Betriebssystem transparent vergeben werden, ist es wünschenswert, dass derartige reine Schlüssel-Property-Werte ebenfalls transparent automatisch vergeben und verwendet werden und die Web-Anwendung im Normalfall nicht mit ihnen in Berührung kommt. Zu diesem Zweck kann zusätzlich zur `Id`-Annotation

eine weitere Annotation des Typs `javax.persistence.GeneratedValue` angefügt werden. Dieser Annotationstyp unterstützt unter anderem einen optionalen Parameter `strategy` vom Typ `javax.persistence.GenerationType`, mit dessen Hilfe die vom Persistence Provider bzw. dem zugrundeliegenden Datenbanksystem zu verwendende *Strategie zur Generierung neuer Schlüsselwerte* eingestellt werden kann. Es werden drei verschiedene Strategien zur Auswahl angeboten, die sich in erster Linie in den technischen Vorgehensweisen unterscheiden, aber auch Auswirkungen darauf haben, ob die generierten Schlüsselwerte nur pro Datenbankrelation oder global eindeutig sind. Solange das Anwendungsprogramm diese generierten Werte (analog zu Speicheradressen, s.o.) nicht beachtet, also keine bestimmte Nummerierung der Datensätze erwartet, spielt die Strategie eine eher untergeordnete Rolle. Da jedoch die Strategie `TABLE` als einzige kompatibel zu beliebigen Datenbanksystemen ist und somit maximale Portabilität bietet, werden wir diese im Folgenden stets benutzen. Sie verwendet eine gesonderte Datenbanktabelle zum Verwalten der schon vergebenen Schlüssel. Die damit generierten Schlüssel sind eindeutig für die gesamte Datenbank, nicht nur pro Relation, sodass keine zusammenhängende Nummerierung aller persistenten Entities einer Entity-Klasse erzielt wird. Eine Alternative ist die Strategie `AUTO`, die auch den Standardwert darstellt, also auch durch Weglassen des `strategy`-Parameters gewählt wird.

23.2.3. Persistente und transiente Properties

Im Normalfall werden alle Properties (bei Property-based Access) bzw. alle Attribute (bei Field-based Access) persistent gehalten. Es ist aber möglich, einzelne Properties bzw. Attribute explizit als transient zu kennzeichnen, sodass diese nicht zum *persistenten Zustand* beisteuern. Dazu ist lediglich eine Annotation des Typs `javax.persistence.Transient` am Getter der Property bzw. am Attribut anzubringen.

Transiente Properties könnten beispielsweise Sitzungsdaten aufnehmen, die lediglich während der Verwendung der internen Repräsentation einer persistenten Entity existieren sollen. So könnte z.B. durch den Standardkonstruktor die Systemzeit der Erzeugung der internen Repräsentation (und somit des Auslesens aus der Datenbank) in einer transienten Eigenschaft hinterlegt werden.

Sinnvoll einsetzbar ist die `Transient`-Annotation aber vor allem für *abgeleitete* Properties. So könnte eine Entity z.B. zwei persistente Properties zum Erfassen eines Vor- und eines Nachnamens aufweisen, sowie eine abgeleitete Property implementieren, die den aus Vor- und Nachname zusammengesetzten Namen ausgibt. Diese könnte als *Nur-Lese-Property* ausgelegt sein, also lediglich durch einen Getter implementiert sein, der den Namen zusammensetzt und ausgibt. Sie könnte auch als *Schreib-Lese-Property* realisiert sein, also zusätzlich einen Setter besitzen, der einen ihm übergebenen Namen wieder in Vor- und Nachnamen zerlegt und die beiden persistenten Properties entsprechend setzt. Der zusammengesetzte Name muss nicht zusätzlich in der Datenbank gespeichert werden, da er stets aus den schon persistenten Vor- und Nachnamen abgeleitet werden kann. Diese abgeleitete Eigenschaft sollte also transient sein.

Für jede als nicht transient gekennzeichnete, also *persistente Property* muss sichergestellt sein,

dass sie auf ein Feld einer Datenbanktabelle abgebildet werden kann. Eine solche Abbildung wird für folgende Typen unterstützt:

- Numerische und Boolesche Typen (genauer: Basisdatentypen und deren Wrapper-Klassen[3] sowie die Typen `BigInteger` und `BigDecimal` aus dem Paket `java.math`)
- Strings, Character-Arrays und Byte-Arrays (`char[]`, `Character[]`, `byte[]`, `Byte[]`)
- Zeit- und Datumsangaben (`java.util.Date`, `java.util.Calender`, `java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`)

  Dabei können nur die drei letztgenannten Typen (`Date`, `Time`, `Timestamp`) aus dem Paket `java.sql` direkt auf Datenbankfelder abgebildet werden. Für eine Property vom Typ `java.util.Date` oder `java.util.Calender` muss zusätzlich über eine Annotation des Typs `javax.persistence.Temporal` eine Meta-Information hinterlegt werden, die angibt, in welchen der drei `java.sql`-Typen sie bei der Persistierung intern konvertiert werden soll: Soll darin nur ein Datum (Tag, Monat, Jahr), nur eine Uhrzeit (Stunde, Minute, Sekunde) oder ein aus Datum und Uhrzeit kombinierter Zeitstempel (Timestamp) gespeichert werden?
- Aufzählungstypen (enumeration types, Schlüsselwort `enum`)
- serialisierbare Typen (die das Interface `Serializable` implementieren)

Zudem können Properties vom Typ einer Entity-Klasse oder vom Typ einer Menge anderer Entities sein. Solche Properties realisieren Assoziationen zwischen Entity-Klassen, wie sie in Abschnitt 23.3 behandelt werden.

23.2.4. Datenbankschemata

Zur Persistierung von Entity-Klassen werden vom Persistence Provider Datenbanktabellen angelegt. In der Regel wird eine Tabelle pro Entity-Klasse benötigt. Eine Ausnahme sind Klassenhierarchien, bei denen mit Hilfe von Annotationen zwischen verschiedenen Abbildungsstrategien gewählt werden kann. Die persistenten Properties werden auf Spalten dieser Tabelle abgebildet. Sonderfälle sind die in Abschnitt 23.3 behandelten Properties zur Darstellung von Verbindungen zwischen Entities, bei denen so genannte *Fremdschlüssel* (foreign keys) als Referenzen auf andere Datensätze abgelegt werden.

Genaue Kenntnisse dieser Abbildung auf Datenbanktabellen sind für die Anwendung der Persistence API nicht notwendig, durch ein Persistenz-Framework soll ja gerade von der Datenbank abstrahiert werden. Das setzt jedoch voraus, dass die Datenbankschemata vollautomatisch beim Deployment des Anwendungsprogramms generiert (aus den Entity-Klassen abgeleitet) werden, wovon wir im Folgenden stets ausgehen. Alternativ dazu bietet die Persistence API auch die

[3] Eine Wrapper-Klasse ist eine Klasse, die einen primitiven Datentypen umhüllt und somit Methoden zur Verfügung stellen kann, z.B. zur Konvertierung.

Möglichkeit, die Datenbankschemata selbst anzulegen und in den Entity-Klassen mit Hilfe von Annotationen Zuordnungen der persistenten Properties zu den Datenbankfeldern vorzunehmen.

Details zu den generierten Schemata oder manuellen Zuordnungen können der Spezifikation [JPA/Spec] entnommen werden.

23.2.5. String-Properties

Eine Besonderheit bei der Abbildung der Properties auf ein relationales Datenbankschema ist bei String-Properties zu beachten: Während Java-String-Variablen prinzipiell beliebig lange Zeichenketten aufnehmen können, hat ein gewöhnliches String-Feld einer relationalen Datenbank eine feste Maximallänge. Wird ein Datenbankschema zu einer Entity-Klasse automatisch generiert, so wird vom System standardmäßig eine Maximallänge von 255 Zeichen gewählt. Mit Hilfe einer Annotation des Typs `javax.persistence.Column` kann allgemein die Zuordnung von Properties einer Entity auf Spalten der korrespondierenden Datenbanktabelle beeinflusst werden. Speziell für String-Properties lässt sich mit dieser Annotation auch eine andere Maximallänge festlegen.

Wenn versucht wird ein Objekt zu persistieren, das einen längeren String in einer String-Property enthält, also nicht in das dazugehörige Datenbankfeld passt, schlägt der Persistierungsversuch fehl und die laufende Transaktion wird zurückgesetzt (dies wird im Folgenden noch genauer erklärt). Es obliegt also dem Anwendungsprogramm vor einem Persistierungsversuch sicherzustellen, dass kein zu langer String eingegeben wurde. Dabei hilft eine JSF Validatorkomponente oder die Verwendung einer Maximallänge für HTML-Texteingabefelder, sodass bereits der Browser keine Eingabe längerer Strings zulässt.

Falls keine Längenbeschränkung gewünscht ist, sondern eine String-Property einer Entity-Klasse tatsächlich beliebig lange Strings aufnehmen können soll, so kann über eine Annotation des Typs `javax.persistence.Lob` an der String-Property dem Persistenz-Framework mitgeteilt werden, dass diese Property nicht auf ein längenbeschränktes Standard-String-Feld, sondern auf ein so genanntes CLOB-Feld (Character Large OBject) für beliebig lange Strings abzubilden ist. Die meisten Datenbanksysteme unterstützen solche Stringfelder, jedoch verschlechtert deren Einsatz die Performanz bei Datenbankzugriffen, weshalb sie nicht standardmäßig eingesetzt werden.

23.3. Beziehungen zwischen Entity-Klassen

Ein Entitätsklassenmodell enthält meist nicht nur unabhängig nebeneinander stehende Entitätsklassen, sondern definiert auch Beziehungen zwischen diesen Klassen. Bei der Implementierung als Entity-Klassen schlagen sich diese in Attributen bzw. Properties vom Typ einer anderen Entity-Klasse oder in der Implementierung von Subklassen nieder. Damit auch solche Beziehungen korrekt persistiert werden, benötigt die Persistence API oft zusätzliche Meta-Informationen, die in Form weiterer Annotationen angegeben werden.

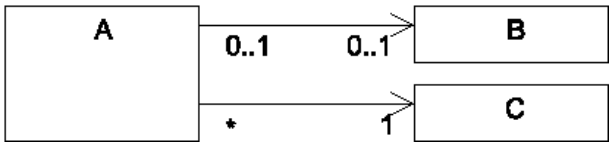

Abbildung 23.1.: Beispiele für unidirektionale 1:1- und n:1-Assoziationen

23.3.1. Unidirektionale 1:1- und n:1-Assoziationen

Die einfachsten Assoziationen ordnen einem Anwendungsobjekt einer Klasse höchstens ein Objekt einer anderen Klasse zu und sind nur in dieser Richtung navigierbar. Abbildung 23.1 zeigt ein Klassendiagramm mit zwei solchen Assoziationen.

Zur Implementierung dieses Beispiels wird in Klasse `A` jeweils ein Attribut bzw. eine Property der assoziierten Klassen `B` und `C` hinzugefügt. Sollen nun die Klassen `A`, `B` und `C` als Entity-Klassen persistiert werden, sind in Klasse `A` zusätzlich (bei Field-based Access an den Attributen selbst, bei Property-based Access an den Gettern der Properties) Annotationen anzufügen: Eine Annotation des Typs `javax.persistence.OneToOne` kennzeichnet eine 1:1-Assoziation, während der Annotationstyp `javax.persistence.ManyToOne` eine n:1-Assoziation markiert. Erst durch diese Annotationen wird gekennzeichnet, dass die Werte der Attribute bzw. Properties der Typen `B` und `C` nicht – wie bei anderen Properties – als Inhalt eines Objekts in einem Feld der zu Klasse `A` gehörenden Datenbanktabelle gespeichert werden sollen, sondern dass es sich bei den `B`- und `C`–Instanzen um verbundene Entities handelt, die bereits in eigenen Datenbanktabellen persistiert sind und im Datensatz eines `A`-Objekts lediglich durch Speicherung ihrer Schlüsselwerte referenziert werden sollen.

```
1 @Entity
2 public class A {
3    @OneToOne
4    public B getAssoziierteBInstanz() {...}
5    @ManyToOne(optional=false)
6    public C getAssoziierteCInstanz() {...}
7    ...
8 }
```

Listing 23.2: Umsetzung von unidirektionalen 1:1- und n:1:assoziationen (Property-based Access)

Codeausschnitt 23.2 zeigt die Anwendung dieser Annotationen auf die Getter der Klasse `A`. Hier und im Folgenden gehen wir von der Verwendung von Property-based Access aus und beziehen uns entsprechend nur noch auf Properties und nicht mehr auf Attribute.

Für die Assoziation zwischen `A` und `C` in Abbildung 23.1 wurde außerdem modelliert, dass jeder `A`-Instanz genau eine `C`-Instanz zugeordnet sein soll, die Property also nicht `null` werden darf. Dies ist in erster Linie durch die Anwendungslogik sicherzustellen. Zusätzlich gibt es zwar auch die Möglichkeit, dies wie in Codeausschnitt 23.2 in der Annotation über eine `optional`-Angabe zu deklarieren, dies ergibt aber nur eine Gültigkeitsprüfung auf Datenbankebene.

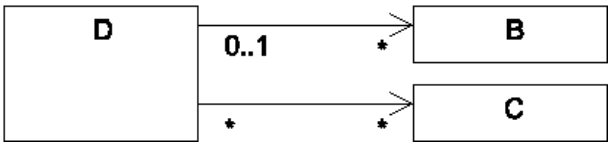

Abbildung 23.2.: Beispiele für unidirektionale 1:n- und m:n-Assoziationen

23.3.2. Unidirektionale 1:n- und m:n-Assoziationen

Etwas aufwendiger als die zuvor betrachteten maximal einwertigen Assoziationen sind mengenwertige Assoziationen, die hier zunächst ebenfalls in unidirektionaler Ausprägung betrachtet werden sollen. Abbildung 23.2 zeigt wieder ein Beispiel als Klassendiagramm.

Während im vorigen Beispiel für Klasse `A` noch eine einfache Property vom Typ `B` oder `C` genügte, muss in diesem Beispiel die Klasse `D` jeweils eine *Menge* von `B`- und `C`-Objekten in einer Property speichern. Damit diese Mengen später über die Persistence API persistiert werden können, sind folgende Punkte zu beachten:

- Die Property muss einen der folgenden Mengentypen aufweisen: `Collection`, `List`, `Map` oder `Set`. Da eine Assoziation im Entitätsklassenmodell im Normalfall eine Menge im mathematischen Sinne (insb. duplikatfrei und ohne Ordnung) von verbundenen Entitäten definiert, bietet sich der Typ `Set` an.
- Neben dem Mengentyp ist zusätzlich der Typ der in dieser Menge zu erfassenden Entities zu definieren. Die beste Möglichkeit hierzu besteht in der Verwendung generischer Ableitungen von Mengentypen (z.B. `Set<B>`).
- Wird für eine Property ein Listentyp verwendet, ist nicht sichergestellt, dass nach dem Auslesen einer Entity aus der Datenbank die Elemente der Liste wieder in derselben Reihenfolge sind wie vorher. Es besteht jedoch die Möglichkeit, mit Hilfe einer Annotation des Typs `javax.persistence.OrderBy` eine Sortierung zu definieren, die beim Auslesen aus der Datenbank angewendet wird.

Der Getter einer solchen Property wird mit einer Annotation des Typs `javax.persistence.OneToMany` bzw. `javax.persistence.ManyToMany` ausgezeichnet, wie in Codeausschnitt 23.3 demonstriert.

```
1 @Entity
2 public class D {
3    @OneToMany
4    public Set<B> getAssoziierteBInstanzen() {...}
5    @ManyToMany
6    public Set<C> getAssoziierteCInstanzen() {...}
7    ...
8 }
```

Listing 23.3: Umsetzung von unidirektionalen 1:n- und m:n-Assoziationen

23.3.3. Bidirektionale Assoziationen

Jede der bisher nur in unidirektionaler Ausprägung betrachteten Assoziationen kann auch auf bidirektionale Navigierbarkeit erweitert werden. Soll z.B. die Assoziation zwischen `B` und `D` in Abbildung 23.2 bidirektional navigierbar werden, d.h. von einer Entity der Klasse `B` auch eine Zugriffsmöglichkeit auf die verbundene Entity der Klasse `D` geschaffen werden, so ist zunächst in Klasse `B` eine Property vom Typ `D` hinzuzufügen und (in diesem Fall) mit `ManyToOne` zu annotieren.

Die Java Persistence API sieht außerdem einen Mechanismus vor, eine der beiden gegenläufigen Referenzen als Umkehrung der anderen zu deklarieren und so dem Persistenzframework mitzuteilen, dass beide zusammen eine bidirektionale Assoziation realisieren sollen. Zu diesem Zweck wird zunächst eine der beiden assoziierten Entity-Klassen als Besitzer (*Owning Side*) der Assoziation ausgewählt. Bei 1:1- und m:n-Assoziationen ist es prinzipiell egal, welche Seite zur Owning Side auserkoren wird, bei 1:n- bzw. n:1-Assoziationen dagegen *muss* die „n-Seite“ die Owning Side sein[4], also diejenige Seite, die maximal eine Entity der assoziierten Klasse referenziert (in obigem Beispiel Klasse `B`). Die Property auf der Gegenseite (*Inverse Side*, in obigem Beispiel Klasse `D`) wird nun als Umkehrung der auf der Owning Side realisierten Referenz ausgezeichnet, indem in der Assoziations-Annotation der Inverse Side der String-Parameter `mappedBy` eingetragen wird, dessen Wert dem Namen der umzukehrenden Owning Side-Property entspricht. Codeausschnitt 23.4 demonstriert die Umsetzung für obiges Beispiel.

```
1  @Entity
2  public class B {
3     //owning side
4     @ManyToOne
5     public D getAssoziierteDInstanz() {...}
6     ...
7  }
8
9  @Entity
10 public class D {
11    //inverse side
12    @OneToMany(mappedBy="assoziierteDInstanz")
13    public Set<B> getAssoziierteBInstanzen() {...}
14    ...
15 }
```

Listing 23.4: Umsetzung einer bidirektionalen 1:n-Assoziation

Durch die `mappedBy`-Angabe wird zwar auf der Datenbankebene logisch eine bidirektionale Assoziation wie in Abbildung 23.3 links erzeugt, auf Objektebene (z.B. bei internen Repräsentationen persistenter Entities) bestehen jedoch zwei gegenläufige unidirektionale Referenzen wie in Abbildung 23.3 rechts. Dies lässt sich nicht vermeiden, da Referenzen zwischen

[4]Der Grund liegt darin, dass die Abbildung auf die Datenbank so einfacher und effizienter ist: Zu jeder Entity der Klasse `B` wird in der Datenbank der Schlüssel der assoziierten Entity der Klasse `D` gespeichert, während zur Navigation in Gegenrichtung die *Menge* der mit einer `D`-Instanz verbundenen `B`-Instanzen intern über eine Anfrage ermittelt wird.

Abbildung 23.3.: Bidirektionale vs. gegenläufige unidirektionale Assoziation

Java-Objekten inhärent unidirektional sind. Es liegt in der Verantwortung der Anwendungslogik, bei Veränderungen an den Objektbeziehungen die Konsistenz der gegenläufigen Referenzen sicherzustellen, sodass tatsächlich logisch immer eine bidirektionale Beziehung besteht.

23.3.4. Lazy Loading

Im Normalfall ist für alle Assoziationen, auch für mengenwertige, so genanntes *Eager Loading*[5] voreingestellt. Das bedeutet, dass zu jeder Entity, die aus der Datenbank wiederhergestellt wird, alle verbundenen Entities ebenfalls geladen werden, sodass das gesamte Objektgeflecht im Speicher liegt. In vielen Fällen kann dies sehr aufwendig sein. Nehmen wir z.B. an, wir haben eine Datenbank mit Kursen und Studenten und einer bidirektionalen Belegungsbeziehung zwischen diesen. Wollen wir nun eine Studentin laden, werden alle von ihr belegten Kurse ebenfalls geladen. Zu jedem dieser geladenen Kurse werden dann auch alle Studenten, die diesen ebenfalls belegen, mit geladen. Natürlich werden dann auch alle weiteren von all den geladenen Studenten belegten Kurse und deren Beleger geladen etc. Im Extremfall kann das bedeuten, dass die gesamte Datenbank in den Speicher geladen wird, obwohl wir womöglich lediglich Informationen über eine einzige Studentin, vielleicht noch über die von ihr belegten Kurse, nicht jedoch über sämtliche weiteren Beleger dieser Kurse haben wollten.

Um einen solchen Effekt zu vermeiden, kann für Assoziationen so genanntes *Lazy Loading*[6] erlaubt werden. Dabei werden verbundene Entities nicht grundsätzlich mitgeladen, sondern erst bei Bedarf (bei Zugriff auf die Verbindung) dynamisch nachgeladen. Dies ist natürlich in erster Linie für mengenwertige Assoziationen von Interesse.

Alle Annotationstypen für Assoziationen kennen hierzu den Parameter fetch, über welchen eingestellt wird, ob der Persistence Provider alle verbundenen Entities laden muss (Eager Loading) oder ob er sie nur laden darf (Lazy Loading). Standardmäßig ist Eager Loading aktiviert.

```
1 @OneToMany(mappedBy="assoziierteDInstanz", fetch=FetchType.LAZY)
2 public Set<B> getAssoziierteBInstanzen()
3 {
4     ...
5 }
```

Listing 23.5: Annotation für Lazy Loading

Der Typ des `fetch`-Parameters ist `javax.persistence.FetchType`, ein Aufzählungstyp, der die Werte `EAGER` und `LAZY` annehmen kann. Codeausschnitt 23.5 demonstriert, wie für die

[5] engl. eager = eifrig, ungedulgig
[6] engl. lazy = faul, träge

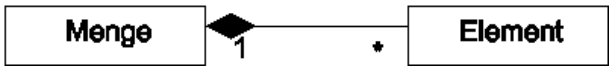

Abbildung 23.4.: Kompositionsbeziehung

Property `assoziierteBInstanzen` der Klasse `D` aus Codeausschnitt 23.4 die Erlaubnis zum Lazy Loading erteilt wird.

Den oben beschriebenen Vorteilen von Lazy Loading stehen erhebliche Nachteile gegenüber. Deshalb ist Lazy Loading nicht als Standardoption voreingestellt, sondern muss vom Anwendungsprogramm explizit erlaubt werden.

Das Problem von Lazy Loading ist, dass das Nachladen von Daten nicht jederzeit möglich ist, sondern (im Standardfall) ausschließlich innerhalb derselben Transaktion geschehen muss, in der die Entity selbst aus der Datenbank geladen wurde. Bei Web-Anwendungen wird eine Transaktion typischerweise die Abarbeitung einer Anwendungslogik-Routine umfassen, insbesondere also innerhalb einer einzigen Request-Bearbeitung gestartet und wieder beendet werden. Wird z.B. eine Entity im Request Scope abgelegt, so sind nachträgliche Zugriffe über die Expression Language oder für einen nachfolgenden Request auf noch nicht geladene Properties nicht mehr erlaubt. Die Anwendungslogik muss daher noch innerhalb der Transaktion das Nachladen sämtlicher Properties durchführen, die im Anschluss an die Transaktion benötigt werden könnten. Dies geschieht jeweils durch Lesezugriffe. Im Abschnitt zum Lebenszyklus einer Entity wird noch genauer auf technische Aspekte dieses Problems eingegangen.

Wird trotz des Verbotes nach Beendigung einer Transaktion auf eine Property zugegriffen, deren Inhalt noch nicht geladen wurde, so ist das Ergebnis nicht definiert.

23.3.5. Kaskaden

Ein Klassenmodell kann zu Assoziationen bestimmte Existenzabhängigkeiten darstellen. So enthält z.B. die in Abbildung 23.4 dargestellte Komposition die Information, dass ein einmal erzeugtes Element genau einer Menge zugeordnet ist und nicht ohne diese Menge existieren kann. Sobald die Menge gelöscht wird, sind auch alle ihre Elemente zu löschen.

Damit ein Anwendungsprogramm dieses Löschen nicht selbst erledigen muss, kann eine Kaskade für die Lösch-Operation definiert werden. Damit wird eine Lösch-Operation auf einer Mengen-Entity automatisch an alle ihre Element-Entities weitergereicht. Derartige Kaskaden, die das automatische Durchreichen von Operationen von einer Entity an verbundene Entities bewirken, gibt es nicht nur für die Lösch-Operation, sondern auch für andere später noch zu betrachtende Operationen (insbesondere für das Persistieren). Aktiviert werden sie durch Hinzufügen des `cascade`-Parameters zu einer Assoziations-Annotation (wie z.B. zu `OneToMany`). Der Wert des `cascade`-Parameters gibt an, welche Operationen weitergereicht werden sollen.

Codeausschnitt 23.6 zeigt einen Ausschnitt der Implementierung der Klasse Menge aus Abbildung 23.4, die ein Weiterleiten der Lösch-Operation vorsieht.

```
1 @Entity
2 public class Menge {
3    ...
4    @OneToMany(cascade=CascadeType.REMOVE)
5    Set<Element> getElemente() {...}
6    ...
7 }
```

Listing 23.6: Annotation für Remove-Kaskade

23.3.6. Klassenhierarchien

Eine weitere Beziehung zwischen Klassen – neben den bisher behandelten Assoziationen – stellt die *Generalisierung* dar. Die Java Persistence API unterstützt die Persistierung von Klassenhierarchien aus Entity-Klassen. Dabei ist im Wesentlichen zu beachten, dass lediglich die Wurzelklasse dieser Hierarchie einen Schlüssel mit Hilfe der `Id`-Annotation definiert, alle Subklassen erben diesen Schlüssel und dürfen keinen eigenen definieren.

Es ist auch möglich, eine *abstrakte Entity-Klasse* zu erstellen und von dieser konkrete Entity-Klassen abzuleiten. Weiterhin ist es erlaubt, dass eine Entity-Klasse *Subklasse* einer *transienten Klasse* ist. Sie erbt dann das Verhalten der Superklasse, der geerbte Zustand bleibt jedoch rein transient, wird also nicht mit persistiert. Als Mischform bietet die Persistence API noch die Möglichkeit, transiente Superklassen von Entities als *Mapped Superclass* auszuzeichnen. Abgeleitete Entity-Klassen persistieren den von diesen geerbten Zustand dann mit.

Für die Abbildung von Entity-Hierarchien auf ein relationales Datenbankschema gibt es verschiedene Möglichkeiten, die jeweils unterschiedliche Vor- und Nachteile aufweisen. Das betrifft den effizienten Zugriff einerseits und den Speicherverbrauch der Datenbank andererseits. Zwischen diesen Möglichkeiten kann mit Hilfe einer Annotation des Typs `javax.persistence.Inheritance`, platziert an der Wurzelklasse der gesamten zu persistierenden Hierarchie, gewählt werden. Da es sich dabei um sehr technische Überlegungen handelt, gehen wir nicht näher darauf ein.

23.4. Lebenszyklus einer Entity

Nachdem in den letzten Abschnitten im Wesentlichen die Erstellung von Entity-Klassen behandelt wurde, wenden wir uns nun deren Verwendung zu. In diesem und im folgenden Abschnitt werden zunächst einige technische Grundlagen angesprochen, bevor in Abschnitt 23.6 in die Implementierung von Anwendungslogik zur Verwendung von Entities eingeführt wird.

23.4.1. Zustände einer Entity

Persistente Entities werden durch Persistieren aus transienten Entities gewonnen. Um also eine neue persistente Entity zu erstellen, ist zunächst eine transiente Instanz der Entity-Klasse (einfach mittels eines Konstruktors) zu erzeugen. Der Zustand einer solchen transienten Entity vor ihrer Persistierung wird in [JPA/Spec] als *new* bezeichnet. Eine solche Entity kann anschließend (mit einer Operation namens persist) persistiert werden. Wir gehen im Folgenden stets davon aus, dass ein Transaktionsmanagement verwendet wird und die Persistierung *innerhalb einer Transaktion* stattfindet. Solange die begonnene Transaktion noch nicht abgeschlossen ist, befindet sich eine gerade persistierte Entity in einem als *managed* bezeichneten Zustand. In diesem Zustand existiert eine vorläufige externe Repräsentation, die nicht notwendig schon in die Datenbanktabelle eingetragen sein muss, sondern auch in einem Cache[7] zum Schreiben vorgemerkt sein kann, und das Java-Objekt im Speicher ist die dazugehörige interne Repräsentation. Die Transaktion kann noch abgebrochen werden (durch einen so genannten *Rollback*[8]), dann ist die Entity nicht persistiert, sondern nach wie vor transient. Wird die Transaktion dagegen erfolgreich abschlossen (durch einen so genannten *Commit*[9]), so wird die externe Repräsentation in der Datenbank festgeschrieben. Das Objekt im Hauptspeicher wird jedoch mit Abschluss der Transaktion von der persistenten Entity entkoppelt, stellt also – nach unserer Definition – keine interne Repräsentation, sondern nur noch eine transiente Kopie der persistenten Entity dar. Dieser Zustand wird in [JPA/Spec] als *detached* bezeichnet.

Ähnlich verhält es sich, wenn innerhalb einer Transaktion eine interne Repräsentation zu einer persistenten Entity bezogen wird. Auch diese befindet sich im *managed*-Zustand, solange die Transaktion andauert. Innerhalb dieser Transaktion kann die persistente Entity modifiziert werden (durch Manipulation ihrer internen Repräsentation) und ist das dynamische Nachladen von verbundenen Entities, für die Lazy Loading eingestellt ist, möglich. Alle Änderungen an der Entity können durch einen Commit der Transaktion bestätigt oder durch einen Rollback verworfen werden. In jedem Fall ist nach Abschluss der Transaktion das Objekt im Speicher wieder von der persistenten Entity entkoppelt, also im Zustand detached. Nach einem Commit ist das entkoppelte Objekt eine transiente Kopie der persistenten Entity, nach einem Rollback dagegen enthält es immer noch die verworfenen Änderungen. Der Rollback bewirkt also lediglich, dass die an der internen Repräsentation vorgenommenen Änderungen nicht persistiert werden, macht am Java-Objekt selbst die Änderungen aber nicht rückgängig.

Auf einem entkoppelten Objekt kann insbesondere auch kein Lazy Loading mehr durchgeführt werden. Es besteht lediglich Zugriff auf den so genannte *available state*, das ist der Teil des Objektzustands, der bereits aus der Datenbank geladen wurde, während das Objekt noch im *managed*-Zustand war. Für Assoziationen mit eingestelltem Lazy Loading bedeutet dies, dass noch während der Transaktion alle verbundenen Entities (durch Lesezugriff auf die entsprechende Property) nachgeladen werden müssen, die später in der transienten Kopie verfügbar sein sollen.

[7] Ein Cache ist ein Speicher zur Pufferung von Daten. Ein Cache kann in einer Hardware- oder Softwarestruktur abgebildet sein und vermeidet Zugriffe auf langsamere Speichermedien, indem z.B. oft benötigte Berechnungsergebnisse im Cache gespeichert werden.

[8] engl. rollback = das Zurückrollen

[9] engl. to commit = übergeben, überlassen

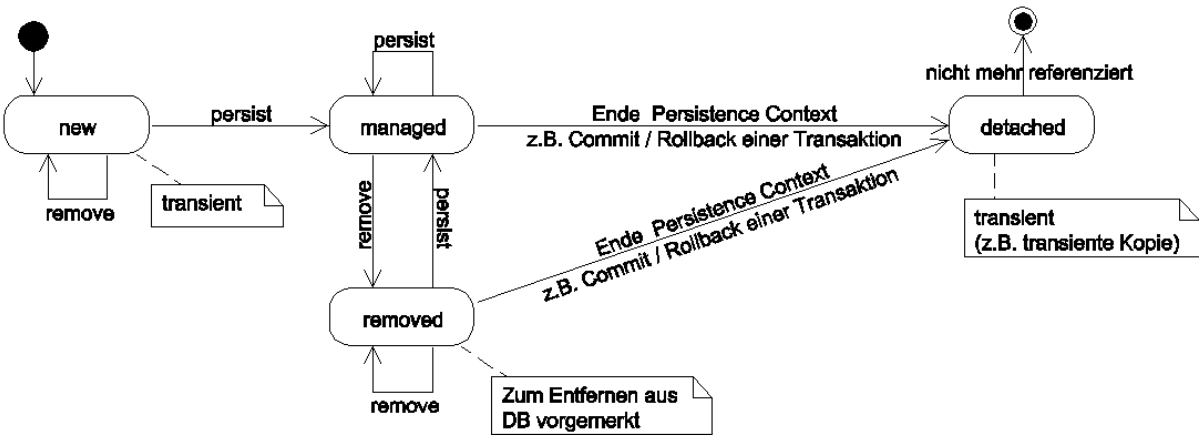

Abbildung 23.5.: Lebenszyklus einer Entity (vereinfacht)

Das Entfernen einer persistenten Entity wird auf einem Objekt im *managed*-Zustand (durch eine Operation namens `remove`) ausgelöst. Das Hauptspeicher-Objekt selbst wird dadurch nicht entfernt, die `remove`-Operation merkt lediglich die externe Repräsentation zum Löschen vor. Die Entity befindet sich nun im sog. *removed*-Zustand. Spätestens beim Commit der Transaktion wird die externe Repräsentation tatsächlich gelöscht. Aus der internen Repräsentation wird nun wieder ein transientes Objekt im *detached*-Zustand, das z.B. durch *Garbage Collection*[10] gelöscht wird, sobald keine Referenz mehr darauf existiert.

Abbildung 23.5 fasst die möglichen Zustände einer Entity zusammen und stellt über das bisher Gesagte hinaus noch ein paar weitere Details dar, z.B. dass auch auf einer Entity im *managed*-Zustand jederzeit noch die `persist`-Operation ausgeführt werden darf. Sie ändert nichts am Zustand, hat jedoch unter Umständen dennoch Auswirkungen, z.B. wenn diese managed Entity mit einer Entity im *new*-Zustand verbunden wurde und für diese Assoziation eine Kaskade der `persist`-Operation eingestellt ist.

23.4.2. Entity Manager, Persistence Context und Persistence Unit

Die oben bereits vorgestellten Operationen `persist` und `remove` sind nur zwei aus einer Reihe von Operationen zur Verwaltung von Entities, wie sie in Abschnitt 23.6 noch näher betrachtet werden. Diese Operationen werden über einen so genannten *Entity Manager* zur Verfügung gestellt.

Ein Entity Manager verwaltet eine als *Persistence Context* bezeichnete Menge von persistenten Entities. Jedes Element eines Persistence Context ist über seinen Schlüssel eindeutig identifizierbar. Auch ist die Existenz höchstens einer internen Repräsentation zu einer persistenten Entity sichergestellt, d.h. in einem Persistence Context kann zu jeder Entity höchstens ein Objekt der

[10] Garbage Collection (engl. Müllabfuhr) bezeichnet eine automatische Speicherbereinigung, die alle nicht mehr benötigten Speicherbereiche identifiziert und freigibt. So werden z.B. Instanzen von Klassen, auf die es keine Referenzen mehr gibt, aus dem Speicher gelöscht.

Entity-Klasse im Hauptspeicher existieren, das sich im Zustand *managed* befindet. Daneben können natürlich beliebig viele transiente Kopien (*detached*) existieren.

Ein Persistence Context hat eine bestimmte Lebensdauer. Im Standardfall existiert er genau für die Dauer einer Transaktion (*transaction-scoped Persistence Context*). Das erklärt das oben beschriebene Verhalten, dass außerhalb einer Transaktion keine Entities im *managed*-Zustand mehr existieren können. Mit einem so genannten *extended Persistence Context* bietet die Java Persistence API auch Möglichkeiten an, von dieser Voreinstellung abzuweichen und einen Persistence Context über Transaktionsgrenzen hinweg zu verwalten, worauf wir jedoch nicht näher eingehen.

Eine *Persistence Unit* ist eine Menge von Entity-Klassen, die eine zusammenhängende, nicht aufteilbare Einheit bilden (da sie z.B. Assoziationen aufweisen oder zu einer Gruppe zusammengefasst wurden). Alle Entities einer Persistence Unit müssen gemeinsam in derselben Datenbank abgelegt werden. Eine Persistence Unit definiert auch die Menge von Entity-Klassen, deren Instanzen von einer `EntityManager`-Instanz verwaltet werden können, d.h. einen Persistence Context bilden können.

23.4.3. Lifecycle Callbacks

Ähnlich wie bei EJBs gibt es auch für Änderungen am Zustand einer Entity *Callback-Ereignisse*, zu deren Behandlung (direkt in der Entity-Klasse oder extern) *Callback-Methoden* implementiert werden können. Wir haben diese Callbacks in Abbildung 23.5 aus Gründen der Übersichtlichkeit nicht eingezeichnet, sondern stellen sie in Tabelle 23.1 zusammen. Der in der linken Spalte stehende Name des Callback-Ereignisses ist gleichzeitig der Bezeichner eines entsprechenden Annotationstyps (aus dem Paket `javax.persistence`). Die Implementierung von Entity-Callback-Methoden mit Hilfe dieser Annotationstypen erfolgt analog zur Implementierung der EJB-Callback-Methoden.

Zu allen diesen Callbacks ist einschränkend anzumerken, dass im Fall eines automatischen Rollbacks der Transaktion aufgrund eines Fehlers (Exception) keine Callbacks mehr durch die vom Rollback ggf. verursachten Datenbankereignisse ausgelöst werden.

23.5. Transaktionsmanagement

Viele Operationen zum Verwalten von Entities, wie z.B. das Persistieren transienter oder das Zerstören persistenter Entities, werden über einen Entity Manager ausgeführt (wie im vorhergehenden Abschnitt beschrieben). Dazu muss die Anwendungslogik zunächst Zugriff auf einen solchen Entity Manager haben. In EJBs lassen sich so genannte *container-managed Entity Manager* über Dependency Injection beziehen. Diese werden automatisch durch den Container bereitgestellt und verwenden implizit das *Transaktionsmanagement* des EJB-Containers über die *Java Transaction API* (JTA). Alternativ könnten *application-managed Entity Manager* ver-

Callback-Ereignis	Beschreibung
`PrePersist`	Callback wird vor der Ausführung der `persist`-Operation des Entity Managers ausgelöst. Bei Kaskaden wird entsprechend auf jeder assoziierten Entity, an die ein `persist`-Aufruf weitergereicht wird, ebenfalls dieser Callback ausgelöst.
`PostPersist`	Callback wird ausgelöst, nachdem die externe Repräsentation des persistenten Objekts als Datensatz neu in die Datenbank eingefügt wurde. Bei `persist`-Operationen auf bereits persistenten Objekten kommt es daher nicht zu einem solchen Callback. Ein ggf. automatisch generierter Schlüssel ist zum Zeitpunkt dieses Callbacks auch in der internen Repräsentation des persistenten Objekts abrufbar. In der Regel wird eine Einfügeoperation nach dem Commit der Transaktion erfolgen, das ist jedoch abhängig vom verwendeten Container bzw. Persistence Provider. Es wäre z.B. auch denkbar, dass schon vor Commit der Transaktion ein Datensatz eingefügt und im Falle eines Rollbacks wieder entfernt wird. Auch kann während einer laufenden Transaktion über die `flush`-Operation des Entity-Managers das Erfassen bzw. Aktualisieren aller Datenbankeinträge zu Entities im *managed*-Zustand erzwungen werden. Aus Gründen der Portabilität sollte man sich bei Verwendung einer solchen Callback-Methode daher nicht auf ein bestimmtes Verhalten verlassen – und insbesondere nicht voraussetzen, dass nach diesem Callback-Ereignis die Entity definitiv persistent ist und nicht doch noch wieder aus der Datenbank gelöscht wird.
`PreRemove`	Analog zu `PrePersist`: Callback wird vor der Ausführung der `remove`-Operation des Entity Managers ausgelöst
`PostRemove`	Analog zu `PostPersist`: Callback erfolgt nach dem Entfernen des Datensatzes (der externen Repräsentation der Entity) aus der Datenbank.
`PostUpdate`	Analog zu `PostPersist`: Callback erfolgt nach der Aktualisierung des Datensatzes (der externen Repräsentation der Entity) in der Datenbank. Es hängt von der Implementierung ab, ob dieser Callback erst nach Transaktionsende erfolgt und ob die Entity anschließend endgültig persistent ist. Es wäre z.B. auch denkbar, dass innerhalb einer Transaktion eine Entity erst geändert, dann gelöscht wird – hier steht nicht fest, ob vor dem `PostRemove`-Callback noch ein `PostUpdate`-Callback stattfindet oder nicht.
`PreUpdate`	Ähnlich wie `PostUpdate`, nur wird dieser Callback nicht nach, sondern vor der Aktualisierung eines Datensatzes ausgelöst.
`PostLoad`	Dieser Callback tritt auf, nachdem zu einem persistenten Objekt eine interne Repräsentation im Speicher erzeugt und dazu die externe Repräsentation aus der Datenbank ausgelesen wurde.

Tabelle 23.1.: Lifecycle Callback Events zu Entities

wendet werden, um deren Erzeugung und Verwaltung sich die Anwendungsentwicklerin selbst kümmern müsste. Dafür hätte sie aber mehr Freiheiten, wäre z.B. nicht auf die JTA festgelegt. Das Beziehen eines container-managed Entity Managers sowie seine Verwendung zur Verwaltung persistenter Entities zeigen wir in Abschnitt 23.6 noch genauer.

Das *implizite* Transaktionsmanagement bei Verwendung von container-managed Entity Managers (im Folgenden kurz als *Container-Transaktionsmanagement* bezeichnet) erlaubt es, sich bei der Programmierung der Anwendungslogik nicht explizit um die Transaktionssteuerung kümmern zu müssen. Im Normalfall beginnt der Container automatisch eine Transaktion, sobald die Anwendungslogik vom Controller gestartet wird. Unteraufrufe von Methoden finden innerhalb der bereits laufenden Transaktion statt. Nach Abschluss der Anwendungslogik beendet der Container die laufende Transaktion wieder, im Normalfall durch einen Commit, in Ausnahmefällen durch einen Rollback.

In diesem Zusammenhang sei nochmals an die 6-Schichten-Architektur erinnert: Die Entities bilden dort die Anwendungsobjekt-Schicht. Die wesentlichen Operationen wie das Persistieren von Entities oder deren Beschaffung aus der Datenbank werden innerhalb der übergeordneten Schicht durch das Persistenz-Framework (zu der insbesondere der Entity Manager gehört) ausgeführt und innerhalb der wiederum darüber liegenden Anwendungslogik-Schicht ausgelöst – in unserem Fall also in EJB-Methoden, welche ihrerseits standardmäßig innerhalb von Transaktionen ausgeführt werden. Aus der noch höher gelegenen Web-Schicht wird höchstens auf transiente Entities zugegriffen: entweder auf transiente Kopien persistenter Entities (z.B. zu deren Präsentation) oder auf neu erzeugte und z.B. mit Benutzereingaben gefüllte transiente Entities, die anschließend an eine Anwendungslogik-Methode zur Persistierung übergeben werden (und von dieser an einen Entity Manager weitergereicht werden). Solche zwischen Web-Schicht und Anwendungslogik transferierten transienten Entities dienen praktisch nur als Datencontainer zur Kommunikation zwischen den Schichten und können auch – zur Entkopplung der Web-Schicht von der Anwendungsobjektschicht – durch so genannte *Data Transfer Objects* ersetzt werden, wie wir sie in Abschnitt 24.4 noch vorstellen werden. Die Web-Schicht kann nur auf transiente Entities zugreifen, da außerhalb von Transaktionen und somit außerhalb von Anwendungslogik keine Entities im *managed*-Zustand existieren können (vgl. Abschnitt 23.4). Arbeitet z.B. die Anwendungslogik auf einer persistenten Entity (im Zustand *managed*) und gibt diese am Ende an ihren Aufrufer, den Controller zurück, so wird durch das implizite Transaktionsende nach Ausführung der Anwendungslogik die dem Controller übergebene (und von diesem z.B. in einem Scope abgelegte und dort von einer JSF-Seite ausgelesene und präsentierte) Entity entkoppelt, also zur transienten Kopie.

23.5.1. Rollback einer Transaktion

In der Standardeinstellung des Container-Transaktionsmanagements wird der Container am Ende der Anwendungslogik einen Commit der Transaktion auslösen, sofern dieser technisch möglich ist (also keine systemseitigen Fehler aufgetreten sind, die einen Commit verhindern), und sofern die Anwendungslogik selbst keine Information hinterlegt hat, die einen Commit verbietet. Ein solches Verbot kann die Anwendungslogik z.B. aussprechen, falls während Ausführung

der Anwendungslogik eine nicht behebbare Komplikation eintritt und sich die persistenten Entities in einem ungewünschten, vielleicht inkonsistenten Zustand befinden. Diese sollen dann nicht endgültig persistiert, sondern wieder auf den Ursprungszustand zurückgesetzt werden.

Um den Commit zu verbieten, also einen Rollback anzufordern, hat die Anwendungslogik die Methode `setRollbackOnly` der `SessionContext`-Schnittstelle aufzurufen. Die dazu benötigte `SessionContext`-Ressource kann z.B. über Dependency Injection bezogen werden. Codeausschnitt 23.7 demonstriert dies an einem kurzen Beispiel.

```
1  import javax.ejb.SessionContext;
2  import ...
3  @Stateless
4  public class MeineALBean implements MeineAL {
5     @Resource SessionContext ctx;
6     ...
7     public void meineAl() {
8        ...
9        if (...) //Fehlerfall, Commit verbieten
10          ctx.setRollbackOnly();
11       ...
12    }
13    ...
14 }
```

Listing 23.7: Verbieten eines Commits / Anfordern eines Rollbacks

23.5.2. Eingriffe ins Transaktionsmanagement

Meistens genügt das Container-Transaktionsmanagement in seiner bisher betrachteten Grundeinstellung, sodass man sich bei der Anwendungslogik-Implementierung nicht um die Steuerung von Transaktionen kümmern muss. Jedoch bietet selbst dieses implizite Transaktionsmanagement einige Eingriffsmöglichkeiten. Standardmäßig arbeitet das Container-Transaktionsmanagement mit so genannter *container-managed Transaction Demarcation*[11], d.h. der Container bestimmt selbst, wann er Transaktionen starten und stoppen wird. Diese container-managed Transaction Demarcation kann in gewissem Rahmen beeinflusst werden: Für jede Methode in einer EJB existiert ein so genanntes *Transaction Attribute*, das die Transaktionssteuerung beeinflusst, vgl. Tabelle 23.2. Die Voreinstellung für jede Methode ist *Required*, was zum bereits skizzierten Standardverhalten führt: Bei Eintritt in die Anwendungslogik wird eine Transaktion gestartet und bei Austritt wieder beendet, wobei Unteraufrufe von EJB-Methoden innerhalb der bereits gestarteten Transaktion stattfinden.

Um die voreingestellten Transaction Attributes zu modifizieren, ist eine Annotation des Typs `javax.ejb.TransactionAttribute` zu verwenden. Durch Anbringen einer solchen Annotation an einer EJB-Methode kann das Attribut für die Methode separat gesetzt werden, durch Anbringen einer solchen Annotation an einer EJB-Klasse (wie in Codeausschnitt 23.8 demons-

[11] engl. demarcation = Abgrenzung, Begrenzung

Transaction Attribute	Bedeutung für eine EJB-Methode M
`Required`	M muss in einer Transaktion ausgeführt werden. Falls noch keine Transaktion läuft, wird beim Eintritt in M eine neue Transaktion gestartet.
`RequiresNew`	M muss in einer *eigenen* Transaktion ausgeführt werden, d.h. beim Eintritt in M wird zwingend eine neue Transaktion gestartet. Eine ggf. schon laufende Transaktion wird vorübergehend suspendiert.
`Supports`	Es ist egal, ob M in einer Transaktion ausgeführt wird, d.h. ein Aufruf von M hat keinerlei Einfluss auf das Transaktionsmanagement.
`NotSupported`	M darf nicht in einer Transaktion ausgeführt werden. Eine ggf. laufende Transaktion wird vorübergehend suspendiert.
`Mandatory`	Ähnlich zu `Required`, nur wird eine laufende Transaktion vorausgesetzt, niemals eine neue gestartet (andernfalls Exception)
`Never`	Ähnlich zu `NotSupported`, nur wird vorausgesetzt, dass noch keine Transaktion läuft (andernfalls Exception)

Tabelle 23.2.: Transaction Attributes

triert) wird eine Standard-Einstellung für alle nicht individuell annotierten EJB-Methoden der Klasse gesetzt.

```
1 @TransactionAttribute(TransactionAttributeType.SUPPORTS)
2 @Stateless
3 public class MeineALBean implements MeineAL {...}
```

Listing 23.8: Setzen des Default Transaction Attribute für eine Session Bean

Bei container-managed Transaction Demarcation ist eine EJB-Methode die kleinste Einheit, die *innerhalb* einer Transaktion ausführbar ist, d.h. es ist nicht möglich, innerhalb der Methode Transaktionen zu starten oder zu beenden. Alternativ kann für eine EJB die sog. *bean-managed Transaction Demarcation* eingestellt werden, bei der nicht der Container die Transaktionsstart- und Endpunkte bestimmt, sondern die Anwendungslogik über eine gegebene Schnittstelle die Start- und Endmarkierungen für Transaktionen setzt. Dabei führt jedoch weiterhin der Container die Transaktionssteuerung auf Basis dieser Markierungen durch.

Welche Transaction Demarcation für eine Bean-Klasse verwendet wird, ist über eine Annotation des Typs `javax.ejb.TransactionManagement` eingestellt. Fehlt diese Annotation, wird container-managed Demarcation eingesetzt. Codeausschnitt 23.9 demonstriert die bean-managed Demarcation.

```
1  import javax.ejb.*;
2  import javax.transaction.UserTransaction;
3  import ...
4  @Stateless
5  @TransactionManagement(TransactionManagementType.BEAN)
6  public class MeineALBean implements MeineAL {
7     @Resource UserTransaction ut;
8
9     public void meineAL(...){
10       ...
11       ut.begin(); //Transaktion starten
12       ...
13       ut.commit(); //Commit der Transaktion anfordern
14       ...
15    }
16    ...
17 }
```

Listing 23.9: Bean-managed Transaction Demarcation

Das Beispiel demonstriert den Bezug einer Instanz des Schnittstellen-Typs `UserTransaction` und deren Anwendung zum Starten einer Transaktion und Anforderung eines Commits. Die Commit-Anforderung muss übrigens nicht zwangsläufig zu einem Commit der Transaktion führen: Falls der Commit fehlschlägt und stattdessen ein Rollback stattfindet, wird eine `RollbackException` erzeugt. Die Schnittstelle bietet noch weitere Operationen an, insbesondere kann statt eines Commits auch explizit ein Rollback angefordert werden.

23.6. Verwendung von Entities

Bisher haben wir uns im Wesentlichen mit der Erstellung der Entity-Klassen sowie mit Grundlagen für Ihren Einsatz beschäftigt. Dieser Abschnitt behandelt nun zur Abrundung des Themenkomplexes die konkrete Verwendung von Entities im Rahmen eines EJB-basierten Anwendungskerns, beschreibt also auch genauer, wie man in EJB-Methoden Entities persistiert, ändert, löscht oder sucht.

Um mit persistenten Entities arbeiten zu können, muss eine EJB-Klasse zunächst einen Entity Manager zur Verwaltung eines Persistence Context beziehen, was mit Hilfe von Dependency Injection über eine spezielle Annotation des Typs `javax.persistence.PersistenceContext` geschehen kann. Die so bezogene `EntityManager`-Instanz kann dann in allen Methoden der EJB verwendet werden, um die im Folgenden näher betrachteten Operationen anzustoßen.

Die nachfolgenden Codebeispiele zu Entity Manager-Operationen seien sämtlich Methoden der in Codeausschnitt 23.10 skizzierten EJB-Klasse und greifen insbesondere auf den injizierten Entity Manager `em` zurück. Als Entity-Klasse wird jeweils die Klasse `Rechnung` aus Codeausschnitt 23.1 verwendet.

```
@Stateless
public class RechnungenVerwaltenBean implements RechnungenVerwalten {
   @PersistenceContext EntityManager em;
   ... Methoden aus den nachfolgenden Listings ...
}
```

Listing 23.10: Rahmen für nachfolgende Beispiele

23.6.1. Persistieren einer neuen transienten Entity

Zum Persistieren einer neuen transienten Entity dient die Operation persist des Entity Managers (vgl. auch Abschnitt 23.4), deren Anwendung in Codeausschnitt 23.11 demonstriert wird. Sie sorgt dafür, dass eine neue externe Repräsentation der Entity angelegt wird und die bisher transiente Entity in den Zustand managed wechselt, also zur internen Repräsentation der nun persistenten Entity wird (vgl. auch Abbildung 23.5). Dabei wird ggf. zunächst ein geeigneter Schlüsselwert generiert und in die Schlüssel-Property eingetragen.

```
public boolean rechnungErfassen(Rechnung rechnung) {
   try {
      em.persist(rechnung);
      return true;
   } catch (EntityExistsException e) {
      return false;
   }
}
```

Listing 23.11: Persistieren einer Entity

Die `persist`-Operation schlägt jedoch mit einer `EntityExistsException` fehl, falls versucht wird, eine Entity zu persistieren, unter deren Schlüssel bereits eine persistente Entity existiert. Das kann beim Persistieren einer neuen transienten Entity z.B. auftreten, wenn keine automatische Schlüsselgenerierung eingestellt ist und versehentlich ein bereits vergebener Schlüssel (im obigen Beispiel eine bereits vergebene Rechnungsnummer) gewählt wird. Ein weiteres Beispiel ist der Versuch, eine transiente Kopie einer Entity erneut zu persistieren.

Eine `EntityExistsException` führt zwar dazu, dass die Entity nicht persistiert wird, jedoch wird dadurch *nicht* die laufende Transaktion zurückgesetzt! Weitere Änderungen am Persistence Context, die in derselben Transaktion vorgenommen wurden, bleiben also bestehen, wobei unter Umständen inkonsistente Zustände entstehen können. Um im Falle einer `EntityExistsException` einen Rollback der Transaktion zu erzwingen, kann im `catch`-Block, der einen auftretenden Fehler abfängt, ein `setRollbackOnly`-Aufruf erfolgen (vgl. Codeausschnitt 23.7). In der einfachen Methode aus Codeausschnitt 23.11 ist das jedoch nicht nötig.

23.6.2. Persistieren von Objektgeflechten

Wenn eine zu persistierende Entity Verbindungen zu anderen Entities eingeht, sind einige Aspekte zu beachten. Eine neue transiente Entity kann nur in folgenden Fällen erfolgreich persistiert werden:

- Die Entity geht noch keine Verbindungen ein. Verbindungen können anschließend noch zwischen persistenten Entities (im *managed*-Zustand) hergestellt werden.
- Sie geht nur Verbindungen zu bereits persistenten Entities ein. Diese können problemlos persistiert werden, da die verbundenen Entities bereits eine persistente Identität besitzen.
- Sie geht zwar Verbindungen zu weiteren neuen transienten Entities (Zustand new) ein, die aber alle innerhalb derselben Transaktion mit persistiert werden – entweder durch explizite separate `persist`-Aufrufe für jede Entity oder implizit unter Verwendung einer Kaskade für die `persist`-Operation.

Es ist also beispielsweise nicht möglich, neue transiente Entities `a` der Klasse `A` sowie `b` der Klasse `B` aus Abschnitt 23.3 zu erzeugen, diese zu verbinden und anschließend nur `a` zu persistieren, `b` jedoch transient zu lassen. Denn solange `b` nicht persistent ist, kann auf Datenbankebene keine Verbindung von `a` zu `b` abgebildet werden. In diesem Fall würde (spätestens beim Versuch, die Transaktion durch einen Commit abzuschließen) das Persistenzframework einen Rollback der laufenden Transaktion erzwingen und eine `RollbackException` auslösen, welche über den aufgetretenen Fehler informiert.

Es ist aber sehr wohl möglich, die Verbindung zwar zwischen den noch transienten Entities `a` und `b` herzustellen, dann jedoch beide Entities innerhalb derselben Transaktion (also im Normalfall innerhalb derselben EJB-Methode) zu persistieren. Um nicht in der Anwendungslogik die Persistierung der verbundenen Entity sicherstellen zu müssen, kann für die `OneToOne`-Assoziation in Klasse `A` eine `persist`-Kaskade eingestellt werden, die immer, wenn auf einem `A`-Objekt die `persist`-Operation stattfindet, auch einen `persist`-Aufruf für das verbundene `B`-Objekt auslöst. Die `persist`-Kaskade kann aber nicht sämtliche Fehlersituationen vermeiden: Sei z.B. `a` eine Entity der Klasse `A` im Zustand *new* und sei `a` verbunden mit einer Entity `b` der Klasse `B`. Ist `b` nicht im Zustand *new*, sondern im Zustand *detached*, so wird eine `persist`-Operation auf `b` in der Regel mit einer `EntityExistsException` fehlschlagen, wie oben bereits beschrieben. Dementsprechend lässt sich dieses Objektgeflecht aus `a` und `b` nicht persistieren, egal ob mit oder ohne Kaskade. Bei eingestellter Kaskade wird die `EntityExistsException` jedoch beim Aufruf `persist(a)` ausgelöst, obwohl dieser Fehler ursächlich auf den Zustand von Objekt `b` zurückzuführen ist, was bei der Fehlersuche auf falsche Fährten führen kann. Ist `b` übrigens im Zustand *managed*, so stellt die (dann überflüssige) Kaskade kein zusätzliches Fehlerpotential dar, da auf Entities im *managed*-Zustand die `persist`-Operation zulässig (und praktisch wirkungslos) ist, vgl. Abbildung 23.5.

Prinzipiell empfehlen wir, `persist`-Kaskaden nur in Fällen einzusetzen, bei denen die verbundenen Komponenten immer zusammen mit ihrem Besitzer erzeugt und persistiert werden sollen, z.B. bei Kompositionen.

23.6.3. Laden einer persistenten Entity

Zum Zugriff auf die persistenten Entites in einer Datenbank gibt es zwei verschiedene Möglichkeiten: Man kann eine einzelne Entity über ihren Schlüssel (Id) referenzieren und einzeln in den Speicher laden. Alternativ kann man mit Hilfe einer Anfragesprache eine (ggf. auch leere oder einelementige) Menge von Entities zu gewissen Suchkriterien ermitteln lassen. Auf Anfragen wird am Ende dieses Abschnitts eingegangen, zunächst betrachten wir hier die erste Möglichkeit.

Zum Suchen einer persistenten Entity in einem Persistence Context und Bezug ihrer internen Repräsentation bietet der zugehörige Entity Manager die `find`-Operation an. Dieser ist der Schlüssel der persistenten Entity zu übergeben. Zudem benötigt die Operation noch die Information, von welcher Entity-Klasse das gesuchte Objekt sein soll, woraus sich insbesondere die Datenbanktabelle bestimmt, in welcher nach der externen Repräsentation zu suchen ist. Existiert eine Entity dieser Klasse mit dem angegebenen Schlüssel, so wird diese als Funktionsergebnis zurückgegeben. Das Ergebnis ist dank des Einsatzes von Generizität direkt vom Typ der Entity-Klasse, es ist also keine *Typumwandlung* (*Cast*) notwendig. Existiert hingegen keine Entity zum übergebenen Schlüssel, so gibt die `find`-Operation `null` als Ergebnis aus.

Codeausschnitt 23.12 demonstriert den Ladevorgang, indem eine Rechnung (vgl. Entity-Klasse aus Codeausschnitt 23.1) anhand ihres Schlüssels, der Rechnungsnummer, lokalisiert wird.

```
1 public Rechnung rechnungLaden(long reNr) {
2    return em.find(Rechnung.class, reNr);
3 }
```

Listing 23.12: Laden einer persistenten Entity

23.6.4. Ändern einer persistenten Entity

Eine persistente Entity kann direkt modifiziert werden, indem ihre interne Repräsentation (ein Objekt der Entity-Klasse im *managed*-Zustand) modifiziert wird. Auch das Herstellen von Verbindungen zwischen persistenten Entities ist so möglich. Mit dem Commit der laufenden Transaktion, in der Regel also beim Beenden der laufenden Anwendungslogik-Methode, werden diese Änderungen automatisch persistent. Zum Ändern wird also keine Operation des Entity Managers benötigt. Oft liegt einer Anwendungslogik-Methode jedoch noch gar keine persistente Entity zur Bearbeitung vor. Wenn der Schlüssel der zu bearbeitenden persistenten Entity bekannt ist, kann diese einfach zunächst mit der `find`-Operation geladen und anschließend modifiziert werden.

Eine typische Situation im Rahmen einer Web-Anwendung könnte nun wie folgt aussehen: Eine Entity wurde während einer Request-Verarbeitung geladen und (als transiente Kopie) an die Web-Schicht zur Präsentation weitergereicht. Die Benutzerin in der Web-Anwendung gibt mit einem späteren Request Änderungen an dieser Entity in Auftrag. Diese Änderungen werden noch in der Web-Schicht (vom Controller) aus den Request-Parametern in die transiente Kopie übertragen. Diese modifizierte transiente Entity wird dann vom Controller wieder an eine EJB-

Methode übergeben, die den Auftrag hat, die Änderungen zu persistieren. Diese Methode könnte nun anhand des in der modifizierten transienten Kopie hinterlegten Schlüssels die persistente Entity über die `find`-Operation des Entity Managers beziehen und dann die Änderungen von der transienten in die persistente Entity übertragen, indem jede persistente Property einzeln kopiert wird.

Ein Entity Manager bietet für diesen Fall mit der `merge`-Operation jedoch eine deutlich komfortablere Möglichkeit, den Zustand einer transienten Entity in die Datenbank einzupflegen. Wird der `merge`-Operation eine transiente Entity (typischerweise im *detached*-Zustand) übergeben, die den Schlüssel einer persistenten Entity trägt, sich in allen anderen Properties aber von der persistenten Entity unterscheiden darf, so lokalisiert die Operation die persistente Entity mit demselben Schlüssel und aktualisiert sie. Die `merge`-Operation automatisiert also praktisch den oben beschriebenen Vorgang.

Der Aufruf der `merge`-Operation entspricht dem der `persist`-Operation. Eine Methode zum Übernehmen von Änderungen, die an einer transienten Kopie einer Rechnung vorgenommen wurden, wird in Codeausschnitt 23.13 skizziert.

```
1 public boolean rechnungAktualisieren(Rechnung rechnung) {
2    try {
3       em.merge(rechnung);
4       return true;
5    } catch (IllegalArgumentException e) {
6       return false;
7    }
8 }
```

Listing 23.13: Aktualisieren einer persistenten Entity durch die merge-Operation

Angenommen, die zu persistierende Klasse geht Assoziationen ein, für welche Lazy Loading erlaubt ist. Dann kann die Situation eintreten, dass für eine transiente Entity-Kopie, die modifiziert und dann der `merge`-Operation übergeben wird, nicht alle verbundenen Entities geladen wurden, und aufgrund des *detached*-Zustands auch nicht mehr nachgeladen werden können. In diesem Fall hat der Persistence Provider laut [JPA/Spec] sicherzustellen, dass beim Merge die betroffenen Verbindungen unverändert bleiben, also nicht etwa gelöscht werden.

Wird der `merge`-Operation übrigens eine transiente Entity übergeben, zu der keine zu aktualisierende persistente Entity existiert (z.B. eine neue Entity oder eine transiente Kopie einer zwischenzeitlich gelöschten persistenten Entity), so legt die Operation eine neue *persistente Kopie* der Entity an. Die der `merge`-Operation übergebene transiente Entity behält in jedem Fall ihren Zustand, nimmt also – anders als bei Anwendung der `persist`-Operation – nicht den *managed*-Zustand an.

Die in Codeausschnitt 23.13 abgefangene `IllegalArgumentException` kann z.B. auftreten, wenn die übergebene Entity im *removed*-Zustand ist.

23.6.5. Löschen einer persistenten Entity

Das Entfernen einer persistenten Entity aus einem Persistence Context und somit die Zerstörung ihrer externen Repräsentation geschieht mit Hilfe der `remove`-Operation des dazugehörigen Entity Managers (vgl. auch Abschnitt 23.4). Dieser muss die zu löschende persistente Entity übergeben werden, genauer deren interne Repräsentation im *managed*-Zustand. Wie schon beim Ändern einer persistenten Entity wird diese interne Repräsentation in der Regel der EJB-Methode, die die Löschung vornehmen soll, noch nicht vorliegen. Sie muss dann zunächst bezogen werden. Codeausschnitt 23.14 demonstriert dies am Beispiel der Entity-Klasse Rechnung, wobei in diesem Beispiel der Methode einfach der Schlüssel (die Rechnungsnummer) der zu löschenden Entity übergeben wird.

```
1  public boolean rechnungLoeschen(long reNr) {
2     Rechnung rechung = em.find(Rechnung.class, reNr);
3     if (rechnung!=null)
4        try {
5           em.remove(rechnung);
6           return true;
7        } catch (IllegalArgumentException e) {
8           return false;
9        }
10    else
11       return false;
12 }
```

Listing 23.14: Löschen einer persistenten Entity anhand des Schlüssels

Die `remove`-Operation kann ebenfalls eine `IllegalArgumentException` erzeugen, insbesondere wenn die übergebene Entity im *detached*-Zustand ist. Auf transienten Entities im *new*-Zustand ist ein `remove`-Aufruf dagegen erlaubt (vgl. Abbildung 23.5), aber in der Regel wirkungslos (er kann jedoch über Kaskaden an verbundene persistente Entities weitergereicht werden und so doch noch Wirkung entfalten).

Nun erscheint es auf den ersten Blick vielleicht wenig zweckmäßig, eine persistente Entity erst aus der Datenbank laden zu müssen (also ein Objekt als interne Repräsentation anzulegen und den persistenten Zustand aus der Datenbank auszulesen und in dieses Objekt zu kopieren), nur um sie anschließend aus der Datenbank löschen zu können. Es gibt auch eine Möglichkeit, das zu vermeiden: Falls vor dem Löschen tatsächlich kein Zugriff auf den persistenten Zustand der Entity notwendig ist, so kann anstelle der `find`-Operation die `getReference`-Operation des Entity Managers zum Einsatz kommen, die nicht zwangsweise den Zustand sofort ausliest, sondern ein Stub-Objekt als Referenz auf die persistente Entity zurückgeben darf[12], das höchstens bei Bedarf den Zustand nachlädt. Wir gehen jedoch auf diese Möglichkeit nicht näher ein, da sie durch den verzögerten Datenbankzugriff neue Probleme mit sich bringt. Zudem wird in vielen Fällen der Zustand der zu löschenden Entity vor dem Löschen benötigt, um alle Verbindungen zu prüfen und ggf. zu lösen, wie im Folgenden gezeigt wird.

[12] Es ist dem Persistence Provider freigestellt, diese Funktionalität zu implementieren

Eine persistente Entity kann nur dann direkt gelöscht werden, wenn sie entweder gar keine Verbindungen eingeht oder von ihr lediglich unidirektionale Verbindungen zu anderen persistenten Entities ausgehen. Wird sie jedoch von anderen persistenten Entities referenziert, so kann sie nicht gelöscht werden, da diese Referenzen sonst ungültig würden. Das Datenbanksystem würde eine derartige Inkonsistenz des Datenbankzustands beim Commit-Versuch erkennen, einen Rollback der laufenden Transaktion auslösen und die Web-Anwendung durch eine `RollbackException` davon unterrichten.

Somit ist vor dem Löschen von persistenten Objekten einer Entity-Klasse, die durch bidirektionale oder ankommende unidirektionale Assoziationen mit anderen Entity-Klassen verbunden ist, sicherzustellen, dass eine zu löschende Entity nicht durch andere persistente Entities referenziert wird. Ansonsten darf die Entity entweder nicht gelöscht werden, oder die verbundenen Entities müssen ebenfalls gelöscht werden, oder aber die Verbindungen müssen explizit gelöst werden.

Eine Entity der Klasse `A` aus Abbildung 23.1 könnte z.B. problemlos gelöscht werden, da von ihr ausschließlich unidirektionale Verbindungen ausgehen können, sie jedoch nicht selbst von anderen Entites referenziert werden kann. Dagegen müsste für eine zu löschende Entity der Klasse `D` aus Codeausschnitt 23.3 zunächst sichergestellt werden, dass sie nicht mehr mit Entities der Klasse `B` verbunden ist – ob dazu lediglich alle noch bestehenden Verbindungen gelöst werden sollen oder die verbundenen Entities auch gelöscht werden sollen, ist im Anwendungskontext zu beurteilen bzw. sollte durch einen Anwendungsfall spezifiziert werden. In Abschnitt 23.7 werden wir ein Beispiel zeigen, in dem eine EJB-Methode zum Löschen einer Entity alle Verbindungen löst, bevor sie die `remove`-Operation anstößt.

Wie schon beim Persistieren liegt ein besonderer Fall vor, wenn gewisse verbundene Entities grundsätzlich mit gelöscht werden sollen, wie es z.B. bei Kompositionen der Fall ist (vgl. Abbildung 23.4). Hierzu kann eine Remove-Kaskade (vgl. Codeausschnitt 23.6) festgelegt werden. Ein Lösen der Verbindung ist in diesen Fällen nicht notwendig.

23.6.6. Prüfen, ob eine Entity persistent/managed ist

Für verschiedene Operationen, insbesondere für das Ändern und Löschen von persistenten Entities, ist es wesentlich, dass das dem Entity Manager übergebene Objekt eine interne Repräsentation einer persistenten Entity ist (Zustand *managed*) und nicht z.B. eine transiente Kopie einer persistenten Entity (Zustand *detached*).

In gewissen Fällen ist es wünschenswert, zur Laufzeit feststellen zu können, ob eine Entity *managed* ist oder nicht. Hierzu bietet der Entity Manager eine Operation namens `contains` an, die prüft, ob sein Persistence Context eine bestimmte Entity enthält. Zum Persistence Context gehören bekanntlich genau die *persistenten* Entities. D.h., genau dann, wenn eine Instanz einer Entity-Klasse im Persistence Context enthalten ist, handelt es sich um die interne Repräsentation einer persistenten Entity, ist ihr Zustand also *managed*. Die Anwendung dieser Operation wird in Codeausschnitt 23.15 skizziert.

```
...
if (em.contains(rechnung)) {
  //rechnung ist managed und kann z.B. gelöscht/aktualisiert werden.
}
...
```

Listing 23.15: Prüfen, ob eine Entity im managed-Zustand ist

23.6.7. Anfragen (Queries) über Entities

Im Abschnitt 23.6.3 wurde bereits eine Möglichkeit vorgestellt, eine Entity aus einer Datenbank zu beziehen. Auf diese `find`-Operation haben wir vor dem Ändern oder Löschen einer Entity zurückgegriffen. Sie hat jedoch zwei große Nachteile: Einerseits muss der Schlüssel der zu lokalisierenden Entity bekannt sein – aber woher soll man z.B. nach dem Neustart einer Web-Anwendung, wenn alle transienten Objekte verloren sind, die möglicherweise generierten Schlüssel der persistenten Entities kennen? Andererseits kann mit der `find`-Operation immer nur genau eine Entity geladen werden, aber keine Menge von Entities (allenfalls indirekt die Menge aller mit einer so lokalisierten Entity verbundenen Entities).

Da die persistenten Entities in einer relationalen Datenbank abgelegt sind, besteht die Möglichkeit, Anfragen (*Queries*) in der Sprache *SQL (Structured Query Language)* zu stellen, um so Mengen von Datensätzen zu ermitteln oder statistische Auswertungen vorzubereiten. Der Entity Manager bietet auch Operationen zu diesem Zweck an. Problematisch dabei ist allerdings, dass beim Arbeiten direkt auf der Datenbank die eigentlich durch die Persistence API eingeführte Abstraktion durchbrochen wird. Zudem muss das aus den Entity-Klassen generierte Datenbankschema zum Formulieren einer SQL-Anfrage bekannt sein. Die ausgelesenen Ergebnisse in Form von Datensätzen müssen außerdem wieder geeignet auf Entities abgebildet werden.

Aus diesen Gründen wurde mit der Java Persistence API eine eigene (von SQL abgeleitete) Anfragesprache, die *Java Persistence Query Language* spezifiziert, in der man die Anfragen auf Entity-Klassen statt auf Datenbanktabellen formuliert.

Nehmen wir z.B. an, unsere Entity-Klasse `Rechnung` enthalte eine String-Property `empfaenger`, (normalerweise wird man eher eine separate Entity-Klasse für Kunden modellieren und mit der Rechnung-Klasse assoziieren, aber um dieses Beispiel möglichst einfach zu halten, sehen wir von Assoziationen ab). Codeausschnitt 23.16 zeigt eine SQL-Anfrage zum Suchen nach allen Rechnungen, deren Empfänger die Zeichenkette „Meier“ enthält. Dabei sind auch der vom Framework generierte Name der Rechnungs-Relation sowie der Feldname für die `empfaenger`-Property einzusetzen. Dieselbe Anfrage, formuliert in Java Persistence Query Language, zeigt Codeausschnitt 23.17.

```
SELECT *
FROM TabellennameRechnungsRelation r
WHERE r.FeldnameEmpfänger LIKE "%Meier%"
```

Listing 23.16: Einfache SQL-Anfrage

```
1 SELECT r
2 FROM Rechnung r
3 WHERE r.empfaenger LIKE "%Meier%"
```

Listing 23.17: Einfache Anfrage in Java Persistence Query Language

In Codeausschnitt 23.17 steht `Rechnung` für die Entity-Klasse `Rechnung`, `r` für die entsprechenden Entities, und `r.empfaenger` für die Property `empfaenger` der Entity `r`.

Um nun eine solche Anfrage auszuführen, wird zunächst mit Hilfe der Operation `createQuery` des Entity Managers ein Objekt vom Schnittstellen-Typ `javax.persistence.Query` generiert, in dem der Anfrage-String hinterlegt ist. Über die Methode `getResultList` des Query-Objekts wird die Anfrage ausgeführt, das Ergebnis ist eine (untypisierte) Liste von Entities. Codeausschnitt 23.18 zeigt beispielhaft, wie eine EJB-Methode zum Suchen aller Rechnungen zu einem übergebenen Namen aussieht. In dieser Anfrage wurde der Platzhalter bzw. Parameter `suchbegriff` verwendet, der durch einen Doppelpunkt eingeleitet wird. Durch die Methode `setParameter(...)` kann diesem Platzhalter ein Wert zugewiesen werden.

```
1 public List sucheRechnungen(String name) {
2    Query q = em.createQuery(  "SELECT r FROM Rechnung r WHERE r.empfaenger
        LIKE :suchbegriff").setParameter("suchbegriff", name);
3    return q.getResultList();
4 }
```

Listing 23.18: Ausführen einer Anfrage

Die Spezifikation der Java Persistence Query Language findet sich in Kapitel 4 von [JPA/Spec]. Wir können im Rahmen dieses Kurses nicht ausführlicher auf diese Sprache eingehen, mit SQL-Vorkenntnissen ist eine Einarbeitung aber einfach.

23.7. Beispiel: Firmen-/Kunden-Verwaltung, Anwendungsobjekte

In diesem Abschnitt demonstrieren wir an einem größeren Beispiel die Implementierung von Anwendungsobjekten mit Hilfe von Entities. Anschließend ergänzen wir dieses Beispiel um eine EJB-basierte Anwendungslogik mit Zugriff auf die hier erstellten Entities. Abschnitt 23.8 schließt ab mit Hinweisen zur Anbindung einer Web-Schicht.

Das Klassendiagramm in Abbildung 23.6 zeigt vier als Entity zu persistierende Klassen und die Assoziationen zwischen ihnen. Die Rollenbezeichner an den Assoziationen und die Attributbezeichner in den Klassen stehen jeweils für die Property-Bezeichner der zu implementierenden Entity-Klassen.

Es werden im Wesentlichen Personen und Firmen modelliert, deren Gemeinsamkeiten in einer

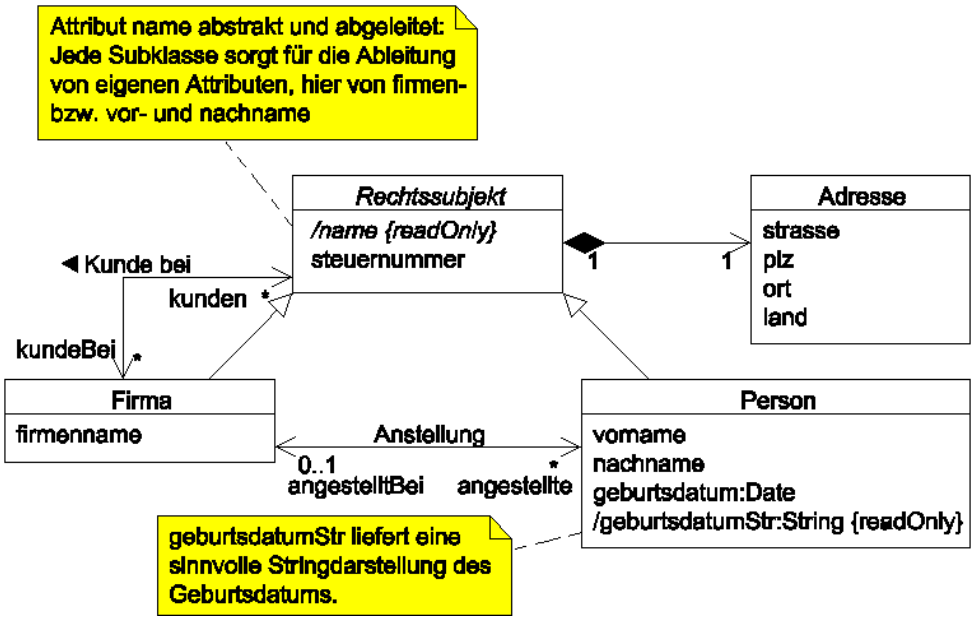

Abbildung 23.6.: Entitätsklassendiagramm zur Firmen-/Kundenverwaltung

abstrakten Superklasse `Rechtssubjekt`[13] zusammengefasst sind. Zu diesen Gemeinsamkeiten gehört, dass sowohl Firmen als auch Personen einen Namen haben. Der Aufbau dieser Namen unterscheidet sich jedoch: Firmen haben nur einen Namen, Personen haben einen Vor- und einen Nachnamen, die hier getrennt erfasst werden sollen. Aufgrund dieser Unterschiede werden Firmenname bzw. Vor- und Nachname in den Klassen `Firma` bzw. `Person` realisiert, und die Klasse `Rechtssubjekt` bekommt eine abgeleitete Property `name`, die den Namen des Rechtssubjekts aus dem Firmennamen bzw. dem Vor- und Nachnamen der Person ableitet. Diese Ableitung muss in der jeweiligen Subklasse erfolgen, weshalb die Property `Rechtssubjekt.name` abstrakt ist. Diese Property vereinfacht es z.B., eine Liste von Namen von Rechtssubjekten zu erstellen.

```
1  package kurs01796.ke7.entityBeispiel.entities;
2  import ...
3  @Entity
4  public class Adresse implements Serializable {
5     private long id;
6     private String land;
7     ... weitere Attribute ...
8
9     public Adresse() {}
10
11    public Adresse(String str, String plz, String ort, String land)  {
12       setStrasse(str);
13       setPlz(plz);
14       setOrt(ort);
```

[13] *Rechtssubjekt* ist der Oberbegriff für *natürliche Personen* und *juristische Personen*. Statt von natürlichen bzw. juristischen Personen sprechen wir aber der Einfachheit halber von *Personen* bzw. *Firmen*, auch wenn diese Begriffe nicht ganz äquivalent sein mögen.

```
15        setLand(land);
16     }
17
18     @Id @GeneratedValue(strategy=GenerationType.TABLE)
19     public long getId() {
20        return id;
21     }
22
23     public void setId(long id) {
24        this.id = id;
25     }
26
27     public String getLand() {
28        return land;
29     }
30
31     public void setLand(String land) {
32        this.land = land;
33     }
34
35     ... weitere Getter und Setter ...
36  }
```

Listing 23.19: Entity-Klasse Adresse (Auszug)

Codeausschnitt 23.19 zeigt einen Auszug aus der Implementierung der Klasse `Adresse`. Wie man sieht, haben wir eine zusätzliche Schlüssel-Property `id` mit automatischer Schlüsselgenerierung verwendet. Neben dem erforderlichen Standard-Konstruktor, der zur Erstellung interner Repräsentationen persistenter Adress-Entities zum Einsatz kommt, wurde ein komfortablerer zweiter Konstruktor zur manuellen Erzeugung neuer transienter Adress-Entities hinzugefügt.

Zur Implementierung der Klassenhierarchie beginnen wir mit ihrer Wurzel, also der Klasse `Rechtssubjekt`, s. Codeausschnitt 23.20. Als Wurzelklasse der Hierarchie muss sie – im Gegensatz zu beiden Subklassen – einen Schlüssel definieren, wofür wir wieder eine separate Schlüssel-Property einführen. Für die abgeleitete abstrakte Nur-Lese-Property `name` wird ein abstrakter Getter deklariert und als transient annotiert, da sich diese Property aus persistenten Properties ableitet und nicht selbst persistent sein soll.

Um lästige `null`-Zeiger-Prüfungen in der Anwendungslogik zu vermeiden, legen wir für *Mengen-Properties* (wie in diesem Fall `kundeBei`) fest, dass diese immer eine (ggf. leere) Menge enthalten sollen, also niemals den Wert `null` annehmen dürfen. Für vom Persistence Provider erzeugte interne Repräsentationen persistenter Entities gilt dies ohnehin, für vom Anwendungsprogramm neu konstruierte transiente Entities muss das dagegen selbst sichergestellt werden. Dazu konstruieren wir zu jedem Mengen-Attribut eine leere Menge vom Typ `HashSet`.

```
1  package kurs01796.ke7.entityBeispiel.entities;
2  import ...
3  @Entity
4  public abstract class Rechtssubjekt implements Serializable {
5     private long id;
```

```
6     private String steuernummer;
7     private Adresse adresse;
8     private Set<Firma> kundeBei = new HashSet<Firma>();
9
10    @Id @GeneratedValue(strategy=GenerationType.TABLE)
11    public long getId() {
12       return id;
13    }
14
15    public void setId(long id) {
16       this.id = id;
17    }
18
19    public String getSteuernummer() {
20       return steuernummer;
21    }
22
23    public void setSteuernummer(String steuernummer) {
24       this.steuernummer = steuernummer;
25    }
26
27    @Transient
28    public abstract String getName();
29
30    @OneToOne(cascade=CascadeType.ALL, optional=false)
31    public Adresse getAdresse() {
32       return adresse;
33    }
34
35    public void setAdresse(Adresse adresse) {
36       this.adresse = adresse;
37    }
38
39    @ManyToMany
40    public Set<Firma> getKundeBei() {
41       return kundeBei;
42    }
43
44    public void setKundeBei(Set<Firma> kundeBei) {
45       this.kundeBei = kundeBei;
46    }
47 }
```

Listing 23.20: Entity-Klasse Rechtssubjekt

Als nächstes erstellen wir die Klasse `Person`, s. Codeausschnitt 23.21. Als Subklasse von `Rechtssubjekt` erbt sie die Schlüssel-Property, darf also nicht selbst einen Schlüssel definieren. Sie implementiert insbesondere die abstrakte Getter-Methode `getName`. Mit `getGeburtsdatumStr` wird eine zweite transiente, von einer persistenten Property abgeleitete Nur-Lese-Property implementiert, die das gespeicherte Geburtsdatum als präsentierbare Zeichenkette ausgibt. Die Property `geburtsdatum` muss, da sie vom Typ `Date` ist, mit einer `Temporal`-Annotation versehen werden, um dem Persistence Provider mitzuteilen, dass sie ein

Datum ohne Uhrzeit repräsentiert.

```
1  package kurs01796.ke7.entityBeispiel.entities;
2  import javax.persistence.*;
3  import java.text.DateFormat;
4  import java.util.Date;
5  import java.util.GregorianCalendar;
6  @Entity
7  public class Person extends Rechtssubjekt {
8     ... private Attribute ...
9
10    public Person() {}
11
12    public Person(String vorname, String nachname, String steuernr,
13                  Adresse adresse, int tag, int monat, int jahr ) {
14       ... Setzen der Properties ...
15    }
16    ... Getter und Setter zu Vor- und Nachname ...
17
18    @Temporal(value=TemporalType.DATE)
19    public Date getGeburtsdatum() {
20       return geburtsdatum;
21    }
22
23    public void setGeburtsdatum(Date geburtsdatum) {
24       this.geburtsdatum = geburtsdatum; }
25
26    public void setGeburtsdatum(int tag, int monat, int jahr) {
27       geburtsdatum = new GregorianCalendar(jahr,monat-1,tag).getTime(); }
28
29    @Transient
30    public String getGeburtsdatumStr() {
31       return DateFormat.getDateInstance(DateFormat.LONG).format(
            geburtsdatum);
32    }
33
34    @ManyToOne
35    public Firma getAngestelltBei() {
36       return angestelltBei;
37    }
38
39    public void setAngestelltBei(Firma angestelltBei) {
40       this.angestelltBei = angestelltBei;
41    }
42
43    @Override @Transient
44    public String getName() {
45       return getNachname()+", "+getVorname();
46    }
47 }
```

Listing 23.21: Entity-Klasse Person (Auszug)

Die Implementierung der zweiten Subklasse `Firma` verläuft ähnlich (s. Codeausschnitt 23.22). Hier ist zu beachten, dass die Klasse `Firma` für zwei bidirektionale Assoziationen jeweils die Inverse Side bildet (für die 1:n-Anstellungsrelation ist dies notwendig, die Wahl der Owning Side für die n:m-Kunde-bei-Relation erfolgte dagegen willkürlich), weshalb das Attribut mappedBy in die Annotationen aufzunehmen ist. Außerdem gehen wir davon aus, dass die Mengen der Kunden und Angestellten einer Firma einerseits sehr groß werden können und andererseits nicht bei jedem Lesezugriff auf eine Firma auch von Interesse sind, weshalb wir für beide Assoziationen Lazy Loading vorsehen.

```
1  package kurs01796.ke7.entityBeispiel.entities;
2  import ...
3  @Entity
4  public class Firma extends Rechtssubjekt {
5     private String firmenname;
6     private Set<Person> angestellte = new HashSet<Person>();
7     private Set<Rechtssubjekt> kunden = new HashSet<Rechtssubjekt>();
8
9     public Firma() { }
10
11    public Firma(String name, String steuernr, Adresse adresse) {
12       ... Setzen der Properties ...
13    }
14
15    @ManyToMany(mappedBy="kundeBei", fetch=FetchType.LAZY)
16    public Set<Rechtssubjekt> getKunden() {
17       return kunden;
18    }
19
20    public void setKunden(Set<Rechtssubjekt> kunden) {
21       this.kunden = kunden;
22    }
23
24    @OneToMany(mappedBy="angestelltBei", fetch=FetchType.LAZY)
25    public Set<Person> getAngestellte() {
26       return angestellte;
27    }
28
29    ... weitere Getter und Setter ...
30
31    @Override
32    @Transient
33    public String getName() {
34       return getFirmenname();
35    }
36 }
```

Listing 23.22: Entity-Klasse Firma (Auszug)

23.8. Beispiel: Firmen-/Kunden-Verwaltung, Anwendungslogik und Web-Schicht

23.8.1. Anwendungslogik (Datenbankschnittstelle)

Nachdem wir im letzten Abschnitt die Entity-Klassen implementiert haben, erstellen wir als nächstes die Anwendungslogik, die in diesem Beispiel nur zur Verwaltung der Entities dient und keine komplexere, fachliche Logik enthält. Der Einfachheit halber sehen wir in diesem Beispiel eine einzige Stateless Session Bean-Klasse zur Datenhaltung vor, die sämtliche Operationen zum Persistieren, Ändern, Laden und Löschen von Anwendungsobjekten bereitstellt.

Operationen zum Persistieren bekommen jeweils eine neue transiente Entity (Zustand *new*) übergeben, Operationen zum Ändern eine ggf. modifizierte transiente Kopie (Zustand *detached*). Zum Löschen sehen wir jeweils zwei Methoden vor, von denen eine den Schlüssel der zu löschenden Entity (vom Typ `long`) übergeben bekommt, die andere dagegen die persistente Entity selbst (Zustand *managed*[14]) oder eine transiente Kopie (Zustand *detached*).

Für jede Assoziation (mit Ausnahme der Komposition) zwischen zwei Entity-Klassen erstellen wir Methoden zum Verbinden zweier Entities und zum Lösen einer Verbindung. Auch diese bieten wir jeweils in zwei Variationen an, von denen eine die Schlüssel der Entities, die andere dagegen Entity-Objekte (*detached* oder *managed*) entgegennimmt. Zum Beziehen von Entities sehen wir zweierlei vor: eine Operation zum Laden einer Entity anhand ihres Schlüssels sowie Routinen, die Listen aller Entities einer Entity-Klasse zurückliefern. Für Firmen implementieren wir beispielhaft auch eine Suchfunktion, die eine Liste genau derjenigen Firmen-Entities zurückliefert, die einem bestimmten Suchkriterium entsprechen (das hier lediglich auf dem Firmennamen basiert). Wird eine einzelne Firma lokalisiert, so sehen zwei Boolean-Parameter vor, dass ein Laden der verbundenen Kunden bzw. Angestellten separat erzwungen wird, da für beide Assoziationen Lazy Loading erlaubt ist.

Alle Routinen geben in diesem Beispiel der Einfachheit halber lediglich ein Boolean-Ergebnis zurück, das anzeigt, ob die Operation erfolgreich war oder fehlgeschlagen ist. Für eine benutzerfreundliche Lösung sollte man dieses Vorgehen natürlich nicht übernehmen, sondern detailliertere Fehlerinformationen zurückliefern. Das Business-Interface wird im Codeausschnitt 23.23 gezeigt.

```
1 package kurs01796.ke7.entityBeispiel.ejb;
2 import kurs01796.ke7.entityBeispiel.entities.*;
3 import ...
4 @Local
5 public interface Datenhaltung {
6
7    //Persistieren neuer transienter Entities
8    public boolean firmaErfassen(Firma firma);
```

[14] Die übergebene Entity kann natürlich nur dann im *managed*-Zustand sein, wenn die Löschen-Methode von einer anderen EJB-Methode und somit innerhalb einer laufenden Transaktion aufgerufen wird.

```
9     public boolean personErfassen(Person person);
10
11    //Aktualisieren einer persistenten Entity durch eine
12    //Entity im detached-Zustand (modifizierte transiente Kopie)
13    public boolean firmaAktualisieren(Firma firma);
14    public boolean personAktualisieren(Person person);
15
16    //Löschen persistenter Entities
17    public boolean personLoeschen(Person person);
18    public boolean personLoeschen(long id);
19    public boolean firmaLoeschen(Firma firma);
20    public boolean firmaLoeschen(long id);
21
22    //Verbinden persistenter Entities
23    public boolean anstellungVerbinden(Firma firma, Person person);
24    public boolean anstellungVerbinden (long firma, long person);
25    public boolean kundeVerbinden (Firma firma, Rechtssubjekt kunde);
26    public boolean kundeVerbinden (long firma, long kunde);
27
28    //Verbindungen lösen
29    public boolean anstellungLoesen(Person person);
30    public boolean anstellungLoesen(long person);
31    public boolean kundeLoesen(Firma firma, Rechtssubjekt kunde);
32    public boolean kundeLoesen(long firma, long kunde);
33
34    //Persistente Entity anhand Primärschlüssel laden
35    public Firma getFirma(long id,   boolean mitKundenListe, boolean
          mitAngestelltenListe);
36    public Person getPerson(long id);
37    public Rechtssubjekt getRechtssubjekt(long id);
38
39    //Listen aller persistenten Entities
40    public List<Firma> getAlleFirmen();
41    public List<Person> getAllePersonen();
42
43    //Suchen nach Firmen
44    //Der Suchbegriff name kann folgende Wildcards enthalten:
45    //% für beliebig viele beliebige Zeichen,
46    //_ für genau ein beliebiges Zeichen.
47    public List<Firma> sucheFirmen(String name);
48 }
```

Listing 23.23: Business Interface der Datenhaltungs-Schnittstelle

In Codeausschnitt 23.24 zeigen wir Auszüge aus der Implementierung der Session Bean, die das Erfassen, Ändern, Löschen, Laden und Suchen am Beispiel der Entity-Klasse `Firma`, und das Herstellen und Lösen bidirektionaler Verbindungen am Beispiel der Anstellungsbeziehung demonstrieren.

```
1  package kurs01796.ke7.entityBeispiel.ejb;
2  import ...
3
4  @Stateless
```

```
5  public class DatenhaltungBean implements Datenhaltung {
6     @PersistenceContext EntityManager em;
7     @Resource SessionContext ctx;
8
9     public boolean firmaErfassen(Firma firma) {
10       if (firma.getAdresse()==null) {
11          return false; //Verbundenes Adress-Objekt obligatorisch!
12       } else {
13          try {
14             em.persist(firma); //Adresse wird auch persistiert
15             return true;
16          } catch (EntityExistsException e) {
17             ctx.setRollbackOnly();
18             return false;
19          }
20       }
21    }
22
23    public boolean firmaAktualisieren(Firma firma) {
24       if (firma.getAdresse()==null) {
25          return false; //Verbundenes Adress-Objekt obligatorisch!
26       } else {
27          try {
28             em.merge(firma); //Auch Adresse wird aktualisiert.
29             return true;
30          } catch (IllegalArgumentException e) {
31             ctx.setRollbackOnly();
32             return false;
33          }
34       }
35    }
36
37    public boolean firmaLoeschen(Firma firma) {
38       try {
39          Firma f = getManagedFirma(firma);
40          if (f!=null) {
41             //Zunächst alle Verbindungen lösen!
42             Set<Rechtssubjekt> kunden_orig = f.getKunden();
43             if (!kunden_orig.isEmpty()) {
44                List<Rechtssubjekt> kunden_kopie =
45                      new LinkedList<Rechtssubjekt>(kunden_orig);
46                for (Rechtssubjekt k : kunden_kopie)
47                   kundeLoesen(f,k);
48             }
49             Set<Firma> kundebei_orig = f.getKundeBei();
50             if (!kundebei_orig.isEmpty()) {
51                List<Firma> kundebei_kopie =
52                       new LinkedList<Firma>(kundebei_orig);
53                for (Firma f2 : kundebei_kopie)
54                   kundeLoesen(f2,f);
55             }
56             Set<Person> angestellte_orig = f.getAngestellte();
57             if (!angestellte_orig.isEmpty()) {
```

```
58                List<Person> angestellte_kopie =
59                       new LinkedList<Person>(angestellte_orig);
60                for (Person a : angestellte_kopie)
61                   anstellungLoesen(a);
62             }
63             em.remove(f);
64             return true;
65          } else
66             return false;
67       } catch (IllegalArgumentException e) {
68          return false;
69       }
70    }
71
72    public boolean firmaLoeschen(long id) {
73       Firma f = getFirma(id,false,false);
74       if (f!=null)
75          return firmaLoeschen(f);
76       else
77          return false;
78    }
79
80    public boolean anstellungVerbinden (Firma firma, Person person) {
81       Firma f = getManagedFirma(firma);
82       Person p = getManagedPerson(person);
83       if (f!=null && p!=null) {
84          //Bidirektionale Verbindung herstellen (beide Richtungen)
85          f.getAngestellte().add(p); //*-Seite
86          p.setAngestelltBei(f);     //0..1-Seite
87          return true;
88       } else
89          return false;
90    }
91
92    public boolean anstellungVerbinden (long firma, long person) {
93       Firma f = getFirma(firma,false,false);
94       Person p = getPerson(person);
95       return anstellungVerbinden(f,p);
96    }
97
98    public boolean anstellungLoesen(Person person) {
99       Person p = getManagedPerson(person);
100      if (p!=null) {
101         Firma f = p.getAngestelltBei();
102         if (f!=null) {
103            //Bidirektionale Verbindung loesen (beide Richtungen)
104            f.getAngestellte().remove(p);
105            p.setAngestelltBei(null);
106         }
107         return true;
108      } else {
109         return false;
110      }
```

```
111     }
112
113     public boolean anstellungLoesen(long person) {
114        Person p = getPerson(person);
115        return anstellungLoesen(p);
116     }
117
118     public Firma getFirma(long id,   boolean mitKundenListe,
119                           boolean mitAngestelltenListe) {
120        Firma f = em.find(Firma.class, id);
121        if (mitKundenListe)
122           f.getKunden();
123        if (mitAngestelltenListe)
124           f.getAngestellte();
125        return f;
126     }
127
128     @SuppressWarnings("unchecked")
129     public List<Firma> sucheFirmen(String name) {
130        Query q = em.createQuery("SELECT f FROM Firma f "+
131                           "WHERE f.firmenname LIKE :firmenname")
132                    .setParameter("firmenname", name);
133        return q.getResultList();
134     }
135
136     private Firma getManagedFirma(Firma f) {
137        if (em.contains(f) || f==null)
138           return f;
139        else
140           return em.find(Firma.class, f.getId());
141     }
142
143     ... Weitere Business-Methoden sowie weitere getManaged...-Methoden ...
144  }
```

Listing 23.24: Auszug aus der Session Bean zur Datenhaltungs-Schnittstelle

Anders als in Codeausschnitt 23.11 wird für den Fall einer `EntityExistsException` beim Persistieren einer Firma explizit ein *Rollback* angefordert. Der Grund dafür ist, dass durch die eingestellte Kaskade *zwei* Entities persistiert werden: ein Rechtssubjekt und eine Adresse. Falls das Persistieren einer dieser beiden Entities fehlschlägt, stellt der Rollback sicher, dass auch die jeweils andere Entity nicht persistiert wird.

Für Methoden wie z.B. `firmaLoeschen(Firma)`, denen eine Entity im Zustand *managed* oder *detached* zu übergeben ist, die aber in jedem Fall eine Entity im Zustand *managed* benötigen, haben wir private Hilfsmethoden wie `getManagedFirma` implementiert, die zunächst prüfen, ob eine ihnen übergebene Entity schon *managed* ist. Falls ja, geben sie diese unverändert zurück, falls nein, gehen sie jeweils davon aus, dass die übergebene Entity *detached* (z.B. eine transiente Kopie) ist und besorgen eine neue Entity im *managed*-Zustand mit Hilfe des in der transienten Entity gespeicherten Primärschlüssels.

Die Methode `firmaLoeschen(Firma)` demonstriert etwas Weiteres: Wie in Abschnitt 23.6 erläutert, kann eine persistente Entity nur dann gelöscht werden, wenn sie keine Verbindungen zu anderen persistenten Entities mehr eingeht. Da es beim Löschen einer Firma nicht in Frage kommt, alle ihre Angestellten oder Kunden ebenfalls komplett zu löschen, sind vor dem Löschen einer persistenten Firma vielmehr ihre Verbindungen zu allen Kunden und Angestellten zu lösen. Nicht zu vergessen ist auch die von `Rechtssubjekt` geerbte Beziehung zu anderen Firmen, bei denen die zu löschende Firma selbst Kunde ist! Zum Lösen der Verbindungen werden die entsprechenden EJB-Methoden eingesetzt. Es ergibt sich dabei jedoch eine kleine Komplikation: Es ist nicht möglich, mit einem Iterator oder einer For-Each-Schleife über die Menge aller Kunden zu iterieren und dabei die Verbindungen zu diesen zu lösen, weil das Lösen der Verbindungen die Kundenmenge, über die iteriert wird, modifiziert – was nicht zulässig ist. Daher müssen wir zunächst von der Kundenmenge (genauer: der Menge von Referenzen auf persistente Kunden) eine Kopie anlegen. Über diese Kopie kann dann problemlos iteriert werden, da sie vom Lösen der Verbindungen nicht beeinflusst wird. Da wir die Kopie ausschließlich zum Iterieren benötigen, wählen wir als Datenstruktur dafür keine Menge (`Set`), sondern eine verkettete Liste, die sich besonders effizient aufbauen und auch durchlaufen lässt.

Zum Erstellen oder Lösen einer Verbindung zwischen zwei persistenten Entities genügt es, wie in Methode `anstellungErfassen` die Properties der Entities (Zustand *managed*) zu bearbeiten. Die Änderungen an den internen Repräsentationen werden beim Commit in die externen Repräsentationen übernommen. Hierzu wird also keine Operation des Entity Managers benötigt.

Abschließend betrachten wir noch eine Besonderheit bei den Query-Methoden, hier `sucheFirmen`. Leider wurde die Unterstützung für generische Mengentypen in der Persistence API bei Queries nicht vollständig durchgehalten: Das Ergebnis einer Query ist immer eine Liste, deren Elementtyp nicht bekannt ist (Typ `List`). Die Ausgabe unserer EJB-Methoden soll aber für Typsicherheit und Benutzungskomfort immer eine Liste von Firmen sein, also vom Typ `List<Firma>`. Unsere Anfragen stellen sicher, dass die Ergebnisliste ausschließlich aus Firmen bestehen kann. Dies ist jedoch für den Compiler und sein Typsystem nicht erkennbar, da die Anfrage nur als String-Literal vorliegt und vom Compiler nicht interpretiert wird. Unsere Lösung ist, die Liste vom Typ `List` als Funktionsergebnis vom Typ `List<Firma>` auszugeben, was in etwa einem Classcast von List zu `List<Firma>` gleichkommt. Die Annotation vom Typ `java.lang.SuppressWarnings` weist den Compiler nun an, in diesen Methoden diesbezügliche Warnungen zu unterdrücken. Wir sichern damit dem Compiler gewissermaßen zu, dass die Anfrageergebnisse tatsächlich immer nur aus Firmen bestehen.

23.8.2. Web-Schicht

Da Web-Anwendungen bereits recht ausführlich behandelt wurden und es in diesem Teil des Kurses in erster Linie um den Anwendungskern geht, werden wir hier nicht die gesamte Web-Schicht der Beispielanwendung vorstellen.

Der Controller kann jederzeit Methoden der oben vorgestellten Anwendungslogik verwenden,

um transiente Kopien persistenter Entities zu beziehen. Diese transienten Kopien kann er z.B. in einem Scope ablegen, so dass sie durch eine JSF-Seite präsentiert werden können. JSF-Seiten können insbesondere über die EL auf die Properties der Entities zugreifen. In Abschnitt 24.4 wird mit dem DTO-Muster eine Alternative zum Übertragen von Entity-Objekten in die Web-Schicht vorgestellt, zunächst soll jedoch das Beispiel möglichst einfach gehalten werden.

Stellvertretend werden wir einen Fall, nämlich das Modifizieren eines Datensatzes, etwas genauer betrachten. Codeausschnitt 23.25 zeigt einen Ausschnitt aus einer Methode einer Managed Bean, die von einem Formular zum Bearbeiten einer Firma ausgelöst wird. In diesem Formular können nur die im Klassendiagramm gezeigten Attribute, nicht jedoch die Verbindungen der Firma oder ihr Schlüssel bearbeitet werden. Die Eingaben (Firmenname, Adressdaten etc.) finden sich in einer Managed Bean und sollen nun durch die Methode in die Datenbank eingepflegt werden. Es wird außerdem vorausgesetzt, dass der Controller bereits vor der Präsentation der Firma im Web-Formular auch eine transiente Kopie der zu bearbeitenden Entity unter dem Namen `firma` im Session Scope abgelegt hat. Von dort holt sich die `speichern`-Methode diese Kopie zurück und überträgt die Änderungen aus dem Formular in die transiente Kopie. Die modifizierte Kopie wird anschließend mit Hilfe der Methode `firmaAktualisieren` in die Datenbank eingepflegt.

```
1  public void speichern()
2    FacesContext fc = FacesContext.getCurrentInstance();
3
4    @EJB
5     Datenhaltung db;
6
7     boolean erfolg = false;
8     Firma f = (Firma)fc.getExternalContext().getSessionMap().get("firma");
9     if (f != null) {
10       f.setFirmenname(getName());
11       f.setSteuernummer(getSteuerNr());
12       Adresse a = f.getAdresse();
13       a.setStrasse(getStrasse());
14       a.setPlz(getPLZ());
15       a.setOrt(getOrt());
16       a.setLand(getLand());
17       erfolg = db.firmaAktualisieren(f);
18    }
19    ... Setzen von Erfolgs- oder Fehlermeldungen, abschließend Forward etc.
          ...
20 }
```

Listing 23.25: Auszug aus der Web-Schicht

23.9. Nebenläufigkeit und (Optimistic) Locking

Ein typisches Merkmal einer Web-Anwendungen ist, dass mehrere Benutzer nebenläufig mit ihr arbeiten können. In der Beispielanwendung aus Abschnitt 23.7 könnte z.B. folgendes Szenario

eintreten: Zwei Benutzer A und B laden je eine transiente Kopie derselben persistenten „Person“ zur Bearbeitung. Benutzerin A korrigiert den Namen der Person, und speichert als erste ihre Änderungen. Benutzerin B trägt die bisher noch fehlende Steuernummer nach und speichert ihre Änderungen kurz nach A. Ohne besondere Vorkehrungen würde B damit die bereits von A geänderte Entity wieder überschreiben, d.h. die Namensänderung wieder zurücknehmen! Noch schlimmer: Weder weiß A, dass ihre Änderung verloren ist, noch weiß B, dass sie Änderungen einer anderen Benutzerin überschrieben hat. Der Fehler wird also möglicherweise niemandem auffallen.

Um solches Verhalten zu verhindern, gibt es verschiedene sogenannte Locking-Strategien: *Pessimistic Locking* hat zum Ziel, durch explizites Setzen von Sperren (*Locks*) auf in Bearbeitung befindliche Daten ein paralleles Bearbeiten derselben Daten auszuschließen. *Optimistic Locking* dagegen geht von der Annahme aus, dass solche Konflikte eher selten vorkommen, also Ausnahmefälle darstellen, deren Vermeidung einen unangemessenen Aufwand verursacht. Konflikte sollen demnach nicht vermieden, sondern beim Schreibversuch erkannt und als Ausnahmen behandelt werden.

Im Folgenden wollen wir beispielhaft die Implementierung von Optimistic Locking für Entities mit Hilfe der Java Persistence API genauer betrachten. Diese wirkt sich dahingehend aus, dass eine Nutzerin ihre Änderungen an einer transienten Kopie einer persistenten Entity P nur so lange (mittels der `merge`-Operation des Entity Managers) speichern kann, wie sich P nicht verändert hat. Andernfalls wird der Merge-Versuch fehlschlagen. Im vorigen Beispiel könnte also B ihre Änderungen nicht mehr speichern und somit auch nicht unwissentlich die von A vorgenommenen Änderungen überschreiben.

Eine Entity-Klasse unterstützt Optimistic Locking, indem sie jeder Instanz neben ihrer ID zusätzlich eine Versionsnummer zuordnet. Bei jedem Merge-Vorgang wird die Versionsnummer inkrementiert, wobei ein Merge nur dann erlaubt ist, wenn die Versionsnummern der persistenten Kopie und der modifizierten transienten Kopie identisch sind, also die persistente Instanz seit dem Anlegen der transienten Kopie nicht verändert wurde. Die Versionsnummer darf nicht von der Web-Anwendung selbst modifiziert werden, sondern wird ausschließlich vom Framework verwaltet. Eine entsprechende Property wird durch eine Annotation des Typs `javax.persistence.Version` markiert. Deklariert man Getter und Setter als `protected`, so ist sichergestellt, dass kein Anwendungscode die Versionsnummer manipulieren kann. In Klassenhierarchien wird die Versions-Property (genauso wie die ID) lediglich in der Wurzelklasse einer Hierarchie implementiert und an alle Subklassen vererbt.

Codeausschnitt 23.26 demonstriert die Ergänzung der Entity-Klasse Rechtssubjekt aus Abschnitt 23.7 um Optimistic Locking.

```
1 import javax.persistence.Version;
2 ...
3 @Entity
4 public abstract class Rechtssubjekt implements Serializable {
5    private long id;
6    private long optLockVersion;
7    ...
```

```
8     @Version
9     protected long getOptLockVersion() { return optLockVersion; }
10    protected void setOptLockVersion(long optLockVersion) { this.
          optLockVersion = optLockVersion; }
11    ...
12 }
```

Listing 23.26: Versions-Property für Optimistic Locking

Allein durch Hinzufügen einer solchen annotierten Versions-Property wird bereits Optimistic Locking für Objekte der Entity-Klasse aktiviert, jedoch sollte zusätzlich eine Ausnahmebehandlung für Konfliktfälle implementiert werden. Die Merge-Operation des Entity Managers wird, wenn das persistente Objekt sich schon verändert hat, einen Schreibvorgang ablehnen, indem ein Fehler des Typs `javax.persistence.OptimisticLockException` erzeugt wird. Es ist nicht sichergestellt, dass dieser Fehler bereits unmittelbar bei Ausführung der Merge-Operation ausgelöst wird, der Container muss den Fehler jedoch spätestens auslösen, wenn entweder ein Commit der Transaktion stattfindet oder wenn die Methode `flush` des Entity Managers aufgerufen wird.

Nehmen wir an, eine Methode einer EJB löst die Merge-Operation aus und soll unmittelbar überprüfen, ob das Speichern erfolgreich war oder nicht[15]. Dazu sollte innerhalb eines Try-Catch-Blocks nach der Merge- unmittelbar die Flush-Operation aufgerufen werden, wie Codeausschnitt 23.27 demonstriert:

```
1  try {
2     Person p = em.merge(bearbeitetePerson);
3     em.flush();
4  }
5     catch (OptimisticLockException e) {
6     ... Ausnahmebehandlung ...
7  }
```

Listing 23.27: Direktes Abfangen einer OptimisticLockException

Abschließend sei noch angemerkt, dass bei Optimistic Locking dasselbe modifizierte transiente Objekt nicht zweimal hintereinander mittels merge gespeichert werden kann, da beim Merge zwar das persistente Objekt aktualisiert wird, das übergebene transiente Objekt – und damit auch seine Versionsnummer – jedoch unverändert bleibt. Die Methode `merge` gibt aber eine neue interne Repräsentation der geänderten persistenten Entity als Funktionsergebnis zurück, mit der weitergearbeitet werden kann.

[15] Wird eine *OptimisticLockException* nicht innerhalb der EJB, welche die Merge-Operation ausführt, behandelt, so wird sie in einer EJBException verpackt, welche dann der Aufrufer abfangen kann.

23.10. Deployment einer Persistence Unit

Eine Persistence Unit (vgl. auch Abschnitt 23.4) besteht im Wesentlichen aus einer Menge von Entity-Klassen. Sie kann als Teil eines EJB-JARs oder auch als separates JAR gepackt und als solches einem EAR hinzugefügt werden (vgl. auch Abschnitt 23.9). Die Verzeichnisstruktur ist praktisch identisch zu der eines EJB-JARs und wird nochmals in Tabelle 23.3 zusammengefasst.

Verzeichnis	Inhalt
/Name	Der Name des Persistence Unit-Teilprojekts ist zugleich das Wurzelverzeichnis. Bei Distribution als JAR tritt der Name nicht in der Verzeichnisstruktur auf, sondern wird als Name für das Archiv herangezogen. Die Entity-Klassen sowie ggf. weitere von diesen benutze Klassen sind entsprechend ihrer Paketstruktur in Unterverzeichnisse einzugliedern.
/Name/META-INF/	Enthält Metadaten über das Archiv. Insbesondere ist hier der Deployment Descriptor für die Persistence Unit (persistence.xml) abzulegen.

Tabelle 23.3.: Verzeichnisstruktur zum Deployment einer Persistence Unit

Zum Deployment einer Web-Anwendung, die solche Persistence Units einbindet, genügt das Deployment des EAR-File auf einem Application Server leider nicht. Es muss vorher eine Verbindung zu einem Datenbanksystem eingerichtet werden. Auf dem Application Server sind hierzu zunächst Ressourcen anzulegen, die insbesondere das Datenbanksystem festlegen, die Verbindung zum Datenbankserver konfigurieren und auch eine von diesem anzusprechende bzw. anzulegende Datenbank zu spezifizieren. Diese Datenbankverbindung wird unter einem JNDI-Namen verzeichnet.

Im Deployment Descriptor `persistence.xml` der Persistence Unit wird dieser JNDI-Name eingetragen, um die Zuordnung der Persistence Unit zur Datenbank vorzunehmen, in der sie persistiert werden soll.

Codeausschnitt 23.28 zeigt den Deployment Descriptor des Fallbeispiels aus Abschnitt 23.7. Neben der Zuordnung zum JNDI-Namen `jdbc/01796/entityBeispiel` wird dort auch die Verwendung der Java Transaction API (JTA) eingestellt, die wir in den vorhergehenden Abschnitten stets vorausgesetzt haben.

Außerdem muss im Deployment Descriptor noch die Anweisung hinterlegt werden, dass die Datenbankschemata automatisch beim Deployment der Persistence Unit aus deren Entity-Klassen zu generieren und die Datenbanktabellen entsprechend anzulegen sind, denn schließlich wollen wir ja die Datenbanken nicht manuell anlegen müssen, sondern die Abbildung der Entities auf die Datenbank komplett dem Container bzw. dessen Persistence Provider überlassen. Leider gibt es keinen plattformunabhängigen Weg, diese Einstellung vorzunehmen, sondern es ist ein vom verwendeten Persistence Provider abhängiger Eintrag im `<properties>`-Tag des Deployment Descriptors zu hinterlegen. In Codeausschnitt 23.28 findet sich ein solcher Eintrag (Zeilen

11 und 12), spezifisch für den Persistence Provider des Oracle Application Servers, Oracle Toplink Essentials. Dieser Eintrag bewirkt nicht nur das Erstellen der Datenbanktabellen beim Deployment, sondern auch das Löschen von alten Tabellen, die bereits im Rahmen eines vorherigen Deployments erzeugt wurden. Bei Verwendung eines anderen Persistence Providers (z.B. Hibernate) sähe diese Konfiguration anders aus.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <persistence version="1.0"
3     xmlns="http://java.sun.com/xml/ns/persistence"
4     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5     xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
6     http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd">
7
8     <persistence-unit name="entityBeispielDB" transaction-type="JTA">
9        <jta-data-source>jdbc/01796/entityBeispiel</jta-data-source>
10       <properties>
11          <property name="toplink.ddl-generation"
12                    value="drop-and-create-tables" />
13       </properties>
14    </persistence-unit>
15 </persistence>
```

Listing 23.28: persistence.xml zum Fallbeispiel

Diese Seite bleibt aus technischen Gründen frei!

24. Ausgewählte Entwurfsmuster

Abschließend betrachten wir einige ausgewählte Entwurfsmuster (design patterns), die beim Entwurf größerer Web-Anwendungen häufig eingesetzt werden.

Analog zu Architekturmustern, wie sie bereits in [SE 1] behandelt wurden, handelt es sich bei Entwurfsmustern um Vorlagen, die auf relativ abstrakter Ebene gängige Konzepte zur Lösung häufiger Entwurfsprobleme beschreiben. Die folgenden Abschnitte beschreiben einige ausgewählte Entwurfsmuster zur Anbindung des Anwendungskerns an die Web-Schicht, wobei wir wieder unsere 6-Schichten-Architektur für Web-Anwendungen zu Grunde legen.

24.1. Session Fassade

Das Entwurfsmuster *Fassade* beschreibt allgemein eine Klasse, die eine ausgewählte Teilmenge der Funktionalität eines komplexen Subsystems über eine zentrale Schnittstelle anbietet. Details des Subsystems werden hinter der Fassade verborgen.

Das Java EE-Entwurfsmuster *Session Fassade* (aus [Al/Cr/Ma]) sieht eine Session Bean als Fassade für einen Teil des Java EE-Anwendungskerns vor. Zugriffe eines Client (in unserem Fall der Web-Schicht) auf Anwendungslogik und Anwendungsobjekte erfolgen dann ausschließlich über derartige Fassaden, welche logisch zusammengehörende Funktionalität über eine einheitliche zentrale Schnittstelle anbieten und so den Anwendungskern verkapseln. Session Fassaden eignen sich zur Reduktion der Kopplung zwischen den Schichten und erhöhen die Wartungsfreundlichkeit der Software. Zudem können die Transaktionskontrolle sowie das Sicherheitsmanagement vollständig von Session Fassades übernommen und somit zentralisiert und aus der eigentlichen Anwendungslogik herausgezogen werden.

Die von einer Session Fassade exportierte Funktionalität ist in der Regel relativ grobgranular und kann sich ihrerseits auf deutlich feinergranularen Teilen des Anwendungskerns abstützen. Bei verteiltem Einsatz von EJBs (über Remote Schnittstellen) hat solche Grobgranularität insbesondere den Vorteil, dass relativ wenig Netzwerkverkehr zwischen Client und EJB-Server[1] stattfindet und die Aufspaltung in feinergranulare Operationen (wie z.B. EJB-Methoden oder Zugriffe auf Entities) lokal auf der Server-Seite stattfindet. Derartige Fassaden, die die Anzahl von Remote-Methodenzugriffen reduzieren sollen, werden auch durch das von Java EE unabhängige

[1] Mit EJB-Server meinen wir kurz denjenigen Server im Netzwerk, auf dem die EJBs einer verteilten Anwendung (in einem EJB Container) laufen, zu dem also der Client (der die Web-Schicht ausführt) eine Netzwerkverbindung aufbauen muss.

Entwurfsmuster *Remote Fassade* aus [FO] beschrieben.

Eine Session Fassade wird laut [Al/Cr/Ma] üblicherweise auf einen fachlich zusammenhängenden Teil des Anwendungskerns zugeschnitten. In einer Banking-Anwendung könnte z.B. eine Session Fassade alle zur Verwaltung eines Kontos angebotenen Operationen bündeln, während eine zweite Fassade den Zahlungsverkehr abdeckt. Damit sind die Session Fassaden im Allgemeinen nicht spezifisch auf Anwendungsfälle zugeschnitten. So wären z.B. in einer Banking-Anwendung mehrere Anwendungsfälle wie „Konto eröffnen“, „Konto schließen“ etc. denkbar, deren Anwendungslogik über dieselbe Session Fassade angesprochen wird.

Da eine Session Fassade als EJB implementiert wird und Aufgaben des Anwendungskerns (wie z.B. das Transaktionsmanagement) zentralisieren kann, zählt sie mit zur Anwendungslogik-Schicht.

24.2. Business Delegate

Während eine Session Fassade eine Schnittstelle zur Anwendungslogik darstellt, die auf der Seite des EJB-Servers angesiedelt ist, sieht das Entwurfsmuster *Business Delegate* [Al/Cr/Ma] eine entsprechende Schnittstelle auf der Seite des Clients vor. Alle Zugriffe von Klassen (z.B. Managed Beans) auf die Anwendungslogik erfolgen bei Einsatz dieses Musters nicht als direkte Zugriffe auf EJB-Methoden, sondern in Form Client-lokaler Methodenaufrufe auf einem Business Delegate, welches seinerseits die Aufrufe an die Anwendungslogik-Schicht delegiert. Business Delegates zentralisieren dabei die zum Zugriff auf den Anwendungskern technisch notwendigen Maßnahmen (wie z.B. den Lookup einer Session Bean) und verkapseln so die zur Realisierung des Anwendungskerns gewählte Plattform. Bei unseren Web-Anwendungen kann so die gesamte Web-Schicht – mit Ausnahme der Business Delegate-Klassen – unabhängig davon gestaltet werden, ob der Anwendungskern mit EJBs realisiert wird, ob Local oder ob Remote Access verwendet wird.

Wie schon das Session Fassade-Muster führt also auch der Einsatz des Business Delegate-Musters zu einer Reduktion der Kopplung und Verbesserung der Wartbarkeit. Zusätzlich sind weitere Vorteile realisierbar, so kann ein Business Delegate-Objekt durch Implementierung geeigneter Caching-Maßnahmen wiederholte Netzwerkzugriffe auf entfernte Anwendungslogik vermeiden und damit die Performance der Web-Anwendung steigern.

Die Muster Business Delegate sowie Session Fassade können sehr gut kombiniert werden. Dabei übernimmt ein Business Delegate-Objekt auf Clientseite die Abstraktion von der Verteilung und Technologie des Anwendungskerns und delegiert die Anwendungslogik-Aufrufe an eine Session Fassade weiter. Die Session Fassade wiederum sorgt auf der Serverseite für die Weiterverarbeitung und delegiert z.B. an mehrere feinergranulare EJB-Methoden. Die Schnittstellen einer Business Delegate-Klasse und der von ihr benutzten Session Fassade-Klasse werden dabei in der Regel übereinstimmen.

Anders als eine Session Fassade übernimmt und zentralisiert eine Business Delegate-Klasse

Aufgaben, die ansonsten in den einzelnen Controller-Klassen angesiedelt wären, weshalb sie bei unserer 6-Schichten-Architektur für Web-Anwendungen mit zur Web-Schicht – und dort zum Controller – zu rechnen ist.

24.3. Service Locator

Das Business Delegate Pattern stützt sich in der Regel auf einem weiteren Entwurfsmuster, dem *Service Locator*, ab (vgl. [Al/Cr/Ma]). Dem liegt im Wesentlichen die Beobachtung zugrunde, dass in den einzelnen Business Delegate-Klassen in der Regel gleichartige, redundante Tätigkeiten zur Vorbereitung des Zugriffs auf den Anwendungskern anfallen, z.B. der Lookup einer Session Bean mit Hilfe eines JNDI-InitialContext-Objekts. Zur Redundanzvermeidung sollten solche Tätigkeiten zentral in eine separate Klasse ausgelagert werden. Diese wird als Service Locator bezeichnet, da sie die Aufgabe der Lokalisierung einer Komponente übernimmt, welche die benötigten Dienste des Anwendungskern anbietet.

Neben der Redundanzvermeidung kann ein Service Locator durch Implementierung geeigneter Caching-Maßnahmen die Anzahl von JNDI-Lookups verringern und so positive Auswirkungen auf die Performance haben.

Da ein Service Locator Gemeinsamkeiten aus den Business Delegate-Klassen herauszieht, zählt er ebenfalls zur Web-Schicht.

24.4. Data Transfer Object (DTO)

Das Muster *Data Transfer Object* aus [Fo] ist vergleichbar mit dem Muster *Transfer Object* aus [Al/Cr/Ma], welches von Alur et al. ursprünglich einmal als *Value Object* bezeichnet wurde[2].

Kurz gesagt handelt es sich bei einem Data Transfer Object (DTO) um ein (in der Regel serialisierbares) Objekt zur Aufnahme einer Sammlung von Daten, das gebündelt zwischen zwei Kommunikationspartnern zu übertragen ist. Durch das Übertragen in einem Schritt kann die Anzahl benötigter Kommunikationsvorgänge reduziert werden.

Das DTO-Muster wird meist in Verbindung mit Datenübertragungen zwischen einzelnen Schichten einer Schichtenarchitektur bzw. zwischen Servern in einem Netzwerk verwendet. Beispielsweise setzt man es gern für die Kommunikation zwischen Web-Schicht und Anwendungslogik ein, indem der Controller die benötigten Daten mit nur einem Methodenaufruf von der Anwendungslogik anfordert, die in Form eines DTO als Funktionsergebnis zurückgegeben werden. Der Controller kann ein solches DTO dann seinerseits der View zur Präsentation zugänglich machen.

Empfohlen wird ein derartiger Einsatz eines DTO insbesondere für Remote-Zugriffe über ein

[2] Alur et al. gaben die Bezeichnung „Value Object“ zu Gunsten von „Transfer Object“ auf, vermutlich weil die Bezeichnung „Value Object“ z.B. von Fowler (u.a. in [Fo]) bereits für ein andersartiges Muster verwendet wurde.

Netzwerk, weil durch die Anforderung und Übertragung der Daten in nur einem Remote-Aufruf – im Gegensatz zur Anforderung der einzelnen Daten in mehreren separaten Remote-Aufrufen – das Netzwerk entlastet werden kann. Aber auch bei rein lokaler Kommunikation kann wegen der geringeren (Programm-)Komplexität und der besseren Übersichtlichkeit die Bündelung von Daten in einem DTO von Vorteil sein. Andererseits muss für das DTO eine zusätzliche Klasse definiert werden und das DTO vor der Rückgabe erzeugt und entsprechend gesetzt werden. Ob Vor- oder Nachteile überwiegen muss im Einzelfall entschieden werden.

Ein DTO ist in der Regel auf einen bestimmten Verwendungszweck zugeschnitten. Beispielsweise kann ein DTO spezifisch zu einer JSF-Seite erstellt werden, wobei genau die in der JSF-Seite zu präsentierenden Daten gebündelt werden, damit diese nicht in vielen einzelnen Schritten ermittelt und im Scope abgelegt werden müssen.

Beim Datenaustausch zwischen Anwendungskern und Web-Schicht stellt die Verwendung spezifischer DTO-Klassen eine Alternative zur Übermittlung transienter Kopien von Entities dar, wie wir sie bereits in Abschnitt 23.5 angesprochen und auch im Beispiel aus Abschnitt 23.8 der Einfachheit halber verwendet haben. Mit DTOs ist z.B. eine Konzentration auf die tatsächlich benötigten Daten möglich (insbesondere können Verbindungen zu anderen Entities ausgeblendet werden), und es können Daten aus verschiedenen Entities aggregiert werden. Erhält der Controller dagegen eine transiente Kopie einer Entity, so muss er für jede Property der Entity, für die Lazy Loading erlaubt ist, wissen, ob diese bereits von der Anwendungslogik geladen wurde oder nicht. Davon hängt nämlich ab, ob er überhaupt auf diese Property der transienten Kopie zugreifen darf. Konzeptionell spricht deshalb auch für den Einsatz von DTOs, dass eine striktere Schichten-Architektur realisierbar ist, während beim Durchreichen von Entities an die Web-Schicht eine Abhängigkeit von der Web-Schicht zur Anwendungsobjektschicht eingeführt wird (dies wird durch drei Pfeile von der Web-Schicht in die Anwendungsobjektschicht in Abbildung 24.1 dargestellt).

Ein DTO wird der niedrigsten Schicht einer Schichtenarchitektur zugeordnet, in der es benutzt wird, damit ausschließlich Abhängigkeiten zwischen den Schichten von oben nach unten auftreten können. Ein DTO, das Daten von der Anwendungslogik an die Web-Schicht zurückliefert, gehört z.B. zur Anwendungslogik (andernfalls wäre die Anwendungslogik von der höher gelegenen Web-Schicht abhängig, was der Schichtenarchitektur widerspräche).

24.5. Value List Handler

Das letzte betrachtete Entwurfsmuster nimmt sich eines Problems von Web-Anwendungen an, die in größeren Datenmengen Suchoperationen durchführen. Die vollständige Darstellung umfangreicher Ergebnislisten sorgt im Kontext von Web-Anwendungen nicht nur für unübersichtliche Web-Seiten, sondern auch für lange Transferzeiten der Response zum Browser des Benutzers. Bei interner Verteilung von Web-Schicht und Anwendungskern würden lange Ergebnislisten auch bereits bei der Übertragung zwischen diesen beiden Schichten die Netzwerklast erhöhen. Eine von vielen Web-Anwendungen wie Online-Shops oder Suchmaschinen bekannte Lösungsstrategie besteht darin, nicht die gesamte Ergebnisliste auf einmal darzustellen, sondern

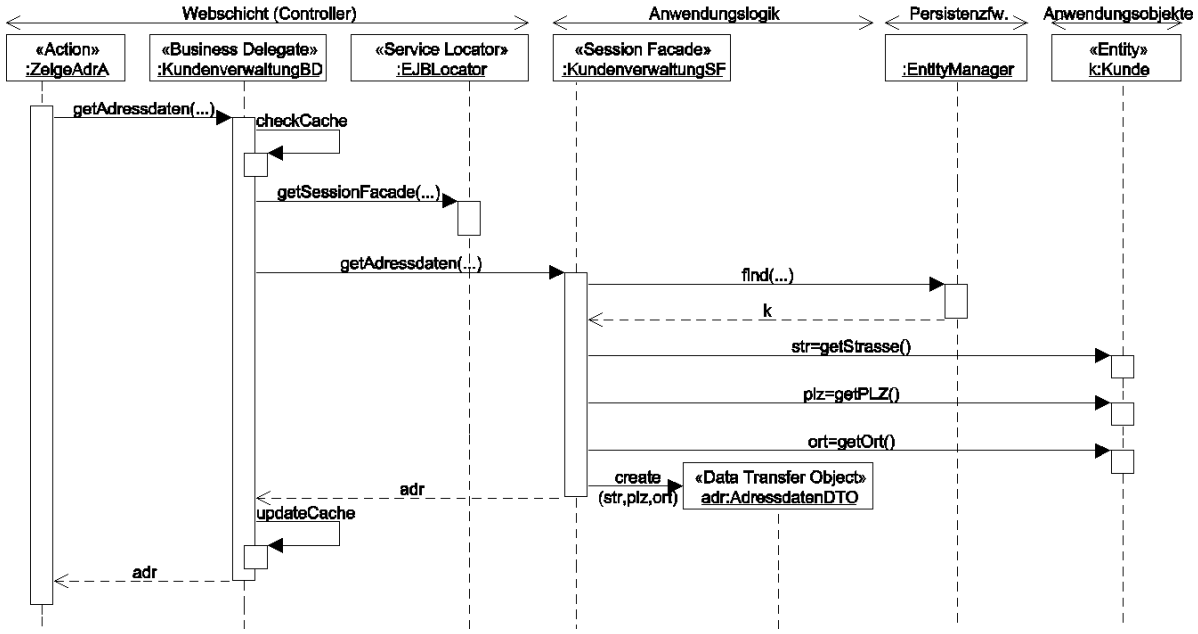

Abbildung 24.1.: Szenario zur Kombination mehrerer Entwurfsmuster

nur einen Ausschnitt daraus zu präsentieren und dem Benutzer die Möglichkeit zu geben, zu weiteren Ausschnitten zu wechseln.

Value List Handler ist ein Entwurfsmuster aus [Al/Cr/Ma], das eine mögliche Umsetzung dieser Strategie beschreibt. Nach diesem Muster wird in der Anwendungslogik-Schicht eine Value List Handler-Klasse realisiert. Eine Instanz dieser Klasse startet auf Anforderung eines Clients eine Suche (durch Delegation), speichert das gesamte Suchergebnis in einer lokalen Ergebnisliste und ermöglicht es dem Client, über kompakte Ausschnitte des Suchergebnisses zu iterieren. Auf diese Weise wird die gesamte Ergebnisliste lediglich Server-intern ermittelt und verwaltet, jedoch nie en bloc über eine Netzwerkverbindung übertragen (weder über die mögliche Netzwerkverbindung zwischen Web-Schicht und Anwendungskern noch über die Netzwerkverbindung zwischen Browser und Web-Server).

Eine solche Value List Handler-Klasse kann als Session Bean implementiert werden. Diese muss dann *stateful* sein, da die gepufferte Ergebnisliste einen Client-spezifischen Conversational State darstellt.

24.6. Kombination der Entwurfsmuster

Die oben vorgestellten Entwurfsmuster lassen sich sinnvoll kombinieren. Abbildung 24.1 zeigt ein grobes Beispiel-Szenario für den kombinierten Einsatz der ersten vier Muster: Der Controller einer Web-Anwendung, hier eine Action, benötige Daten vom Anwendungskern, im Beispiel die Adressdaten zu einem Kunden.

Wir gehen davon aus, dass die Anwendungslogik eine entsprechende Routine `getAdressdaten` anbietet, die diese Daten zu einem bestimmten Kunden ermittelt. Auf der Seite der Web-Schicht stehe zum Zugriff auf den Anwendungskern ein Business Delegate-Objekt `KundenverwaltungBD` bereit, auf welchem die Action die Adressdaten anfordert. Auf der Seite des Anwendungskerns bestehe eine korrespondierende Session Fassade, an deren gleichnamige Methode der Aufruf durch `KundenverwaltungBD` delegiert wird (sofern das Ergebnis nicht bereits im Cache des Business Delegate-Objekts vorliegt). Zum Beziehen einer Referenz auf diese Fassade (implementiert als Session Bean) dient auf der Seite der Web-Schicht ein Service Locator. Die Session Fassade sammelt nun die gewünschten Adressdaten und bündelt diese in einem Data Transfer Object, welches sie an die Web-Schicht zurück transferiert.

Der Einfachheit halber führt die Session Fassade in Abbildung 24.1 alle Operationen selbst aus, sie könnte natürlich auch Aufgaben an andere EJBs delegieren.

Der Controller kann nun dieses DTO unter anderem über einen Scope der View (z.B. einer JSF-Seite) zur Präsentation zugänglich machen. Auf diese Weise erfolgt nur ein einziger Aufruf zwischen Web- und Anwendungslogikschicht.

Auch das Value List Handler Pattern lässt sich natürlich mit den oben genannten Mustern kombinieren. Der Value List Handler könnte z.B. selbst eine Session Fassade darstellen oder hinter einer Session Fassade verborgen werden. Durch den Einsatz einer Business Delegate-Klasse mit Cache auf Seiten der Web-Schicht kann insbesondere auch für den Fall, dass ein Benutzer im Ergebnis wieder zurückblättert, das wiederholte Übertragen von Ergebnis-Teillisten von der Anwendungslogik- zur Web-Schicht vermieden werden. Zur Übertragung einer Ergebnis-Teilliste kann wieder ein Data Transfer Object herangezogen werden.

Abbildungsverzeichnis

Listings

Tabellenverzeichnis

Abkürzungsverzeichnis

AI ..... Adobe Illustrator
AJAX ..... Asynchronous JavaScript and XML
API ..... Application Programming Interface
ASP ..... Active Server Pages
CDN ..... Content Delivery Network
CGI ..... Common Gateway Interface
CLOB ..... Character Large OBject
COD ..... Code on Demand
CSS ..... Cascading Style Sheets
DNS ..... Domain Name System
DOM ..... Document Object Model
DTD ..... Document Type Definition
DTO ..... Data Transfer Object
EJB ..... Enterprise JavaBean
EL ..... Expression Language
EPS ..... Encapsulated PostScript
FLV ..... Flash Video
GIF ..... Graphics Interchange Format
GUI ..... Graphical User Interface
HTML ..... Hypertext Markup Language
HTTP ..... Hypertext Transfer Protocol
HTTPS ..... Hypertext Transfer Protocol Secure
ID ..... Identifier
IDE ..... Integrated Development Environment
IIS ..... Internet Information Services
IP ..... Internet Protocol
JAR ..... Java Archive
Java EE ..... Java Enterprise Edition
Java SE ..... Java Standard Edition

JMS ..... Java Message Service
JNDI ..... Java Naming and Directory Interface
JPA ..... Java Persistence API
JPEG ..... Joint Photographics Expert Group
JS ..... JavaScript
JSF ..... JavaServer Faces
JSP ..... JavaServer Pages
JSTL ..... JavaServer Pages Standard Tag Library
JTA ..... Java Transaction API
MVC ..... Model View Controller
PHP ..... PHP Hypertext Preprocessor
PNG ..... Portable Network Graphics
POJO ..... Plain Old Java Object
SDK ..... Software Development Kit
SoC ..... Separation of Concerns
SQL ..... Structured Query Language
SSL ..... Secure Socket Layer
TCP ..... Transmission Control Protocol
TIF ..... Tagged Image File Format
UDP ..... User Datagram Protocol
UI ..... User Interface
URI ..... Uniform Resource Identifier
URL ..... Uniform Resource Locator
URN ..... Uniform Resource name
VDL ..... View Declaration Language
WAR ..... Web Archive
WMF ..... Windows Metafile
WWW ..... World Wide Web
XHTML ..... Extensible Hypertext Transfer Protocol
XML ..... Extensible Markup Language

Literaturverzeichnis

Das Literaturverzeichnis ist nach Kurseinheiten sortiert. Die Literatur jeder Kurseinheit ist dabei alphabetisch sortiert.

Kurseinheit 1

[Be/Mi] H. Behme, S. Mintert: XML in der Praxis, addison-Wesley, 2000, http://mintert.com/xml/buch/

[HTML] SELFHTML, http://www.selfhtml.org

[HTTPD] Apache HTTPD Web Server, http://httpd.apache.org

[IIS] Microsoft Internet Information Services, http://www.iis.net/

[JEE/Spec] Oracle: java Platform, Enterprise Edition (Java EE) Specification, v7, https://jcp.org/en/jsr/detail?id=342

Kurseinheit 2

[ASP] ASP.NET, www.asp.net (Januar 2014)

[Bo/Vo] F. Bongers, M. Vollendorf: jQuery - Das Praxishandbuch, Galileo Press GmbH, 2013

[CSS] Cascading Style Sheets, www.w3.org/Style/CSS (Januar 2014)

[Gar/Iri] T. Garsiel, P. Irish: Funktionsweise von Browsern - Hinter den Kulissen moderner Web-Browser, http://www.html5rocks.com/de/tutorials/internals/howbrowserswork/#Introduction (20.01.2014)

[Gross] A. Grosskurth, M. W. Godfrey: A reference architecture for web browsers, ICSM '05: Proceesings of the 21st IEEE International Conference on Software Maintenance, IEEE, 2005

[HTML] SELFHTML, de.selfhtml.org (Januar 2014)

[Jun] J. Juneau: Java EE 7 - Recipes „A Problem-Soluation Approach“, Apress, 2013

[Lor] P.A. Lorenz: ASP.NET - Webserverprogrammierung und XML Web Services im .NET-Framework, 2. Auflage, Hanser, 2004

[Lub] M. Lubkowitz: Webseiten programmieren und gestalten, 4. Auflage, Galileo Press GmbH, 2004

[Schi/Schm] M. Schießer, M.Schmollinger: Workshop Java EE 7 - Ein praktischer Einstieg in die Java Enterprise Edition mit dem Web Profile, dpunkt.verlag, 2013

[Schmidt] U. Schmidt: Professionelle Videotechnik: Grundlagen, Filmtechnik, Fernsehtechnik, Geräte- und Studiotechnik in SD, HD, DI, 3D, 5. Auflage, Springer, 2009

[Theis] T. Theis: Einstieg in PHP 5 und MySQL 5, 4. Auflage, Galileo Press GmbH, 2006

Kurseinheit 3

[EJB/Spec] Oracle: Enterprise JavaBeans, Version 3.2, https://jcp.org/en/jsr/detail?id=345

[GFish] GlassFish Community, http://glassfish.java.net

[JSE/Spec] Oracle: Java SE 7 Release Contents, https://jcp.org/en/jsr/detail?id=336

[JB/Spec] Oracle: JavaBeans Specification 1.01, http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec/

[JEE/Tut] Oracle: The Java EE 7 Tutorial, http://docs.oracle.com/javaee/7/tutorial/doc

[JEE/Spec] Oracle: Java Platform, Enterprise Edition 7 (Java EE 7) Specification, https://jcp.org/en.jsr/detail?id=342

[JPA/Spec] Oracle: JavaServer Pages Specification Version 2.1, https://jcp.org/en/jsr/detail?id=338

[JSP/Spec] Oracle: Java Persistence API Specification 2.3, https://jcp.org/en/jsr/detail?id=245

[JSTL/Spec] Oracle: JavaServer Pages Standard Tag Library Version 1.2, https://jcp.org/en/jsr/detail?id=52

[Serv/Spec] Oracle: Java Servlet Specification Version 3.1, https://jcp.org/en/jsr/detail?id=340

[SE I] H.-W. Six, M. Winter: Kurs 01793 „Software Engineering I - Methodische Entwicklung objektorientierter Desktop-Applikationen“, FernUniversität in Hagen, 2006

[Tu/Sa/Le] V. Turau, K. Saleck, C. Lenz: Web-basierte Anwendungen entwickeln mit JSP 2, 2. Auflage, dpunkt.verlag 2004

Kurseinheit 4

[SE I] H.-W. Six, M. Winter: Kurs 01793 „Software Engineering I - Methodische Entwicklung objektorientierter Desktop-Applikationen“, FernUniversität in Hagen, 2006

Kurseinheit 5

[JEE/Tut] Oracle: The Java EE 7 Tutorial, http://docs.oracle.com/javaee/7/tutorial/doc

[SE I] H.-W. Six, M. Winter: Kurs 01793 „Software Engineering I - Methodische Entwicklung objektorientierter Desktop-Applikationen“, FernUniversität in Hagen, 2006

[Ku/Ma] M. Kurz, M. Marinschek: JavaServer Faces 2.2 - Grundlagen und erweiterte Konzepte, 3. Auflage, dpunkt.verlag, 2014

[Schi/Schm] M. Schießer, M. Schmollinger: Workshop Java EE 7, 1. Auflage, dpunkt.verlag, 2013

[Ge/Ho] D. Geary, C. Horstmann: Core JavaServer Faces, 2. Auflage, Prentice Hall, 2007

[Bu/Sch] E. Burns, C. Schalk: JavaServer Faces 2.0 - The Complete Reference, 1. Auflage, Mc Graw Hill, 2010

Kurseinheit 6

[BuMo] B. Burke, R. Monson-Haefel: Enterprise JavaBeans 3.0, O'Reilly, 5. Auflage, 2006

[EJB/Spec] Oracle: Enterprise JavaBeans, Version 3.1, http://jcp.org/aboutJava/communityprocess/final/jsr318/index.html

[Derby] Apache Derby, http://db.apache.org/derby/

[Hilber] Hibernate, http://hibernate.org/

[JDB] Oracle: Java DB, http://www.oracle.com/technetwork/java/kavadb/overview/index.html

[JEE/Tut] Oracle: The Java EE 7 Tutorial, http://docs.oracle.com/javaee/7/tutorial/doc

[JEE/Spec] Oracle: Java Platform, Enterprise Edition (Java EE) Specification, v7, http://jcp.org/en/jsr/detail?id=342

[JNDI] Oracle: Java Naming and Directory Interface, http://www.oracle.com/technetwork/java/jndi/index.html

[JPA/Spec] Oracle: Java Persistence API, Version 2.1, http://jcp.org/en/jsr/detail?id=338

[MySQL] Oracle: MySQL, http://mysql.de/

[SE I] H.-W. Six, M. Winter: Kurs 01793 „Software Engineering I - Methodische Entwicklung objektorientierter Desktop-Applikationen“, FernUniversität in Hagen, 2006

Kurseinheit 7

[Al/Cr/Ma] Deepak A., Crupi J., Malks D.: core J2EE Patterns 2nd edition, Sun Microsystems Press, Prentice Hall, 2003

[EJB/Spec] Oracle: Enterprise JavaBeans, Version 3.1, http://jcp.org/aboutJava/communityprocess/final/jsr318/index.html

[Fo] Martin Fowler: Patterns of Enterprise Application Architecture, Addison Wesley, 2003

[GFish] GlassFish Community, http://glassfish.java.net

[Hilber] Hibernate, http://hibernate.org/

[JBoss] JBoss Application Server http://www.jboss.org/jbossas

[JPA/Spec] Oracle: Java Persistence API, Version 2.1, http://jcp.org/en/jsr/detail?id=338

[OTLE] Oracle Toplink Essentials, http://www.oracle.com/technology/products/ias/toplink/index.html

[SE I] H.-W. Six, M. Winter: Kurs 01793 „Software Engineering I - Methodische Entwicklung objektorientierter Desktop-Applikationen“, FernUniversität in Hagen, 2006

Diese Seite bleibt aus technischen Gründen frei!

Stichwortverzeichnis

002628996
(10/20)

01796-4-01-S 1