LE 30
1

Software-Technik

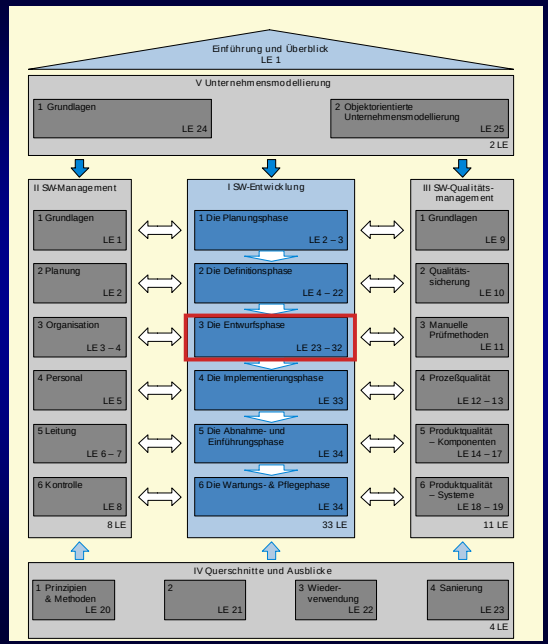
3 Die Entwurfsphase Web-Architekturen

[leicht gekürzt]

Prof. Dr. Helmut Balzert
Lehrstuhl für Software-Technik
Ruhr-Universität Bochum

© Helmut Balzert 2001

LE 30
2



LE 30
3

Inhalt

LE 30
4

3.10.1 Einleitung

- Grundkonzepte
  - HTML *(Hyper Text Markup Language)*
  - *Web-Server*
  - *Web-Client*
  - HTTP
  - *Web-Browser* und *Web-Server* kommunizieren über das HTTP *(Hyper Text Transfer Protocol)*
    - Der *Browser* sendet eine Anfrage als HTTP-Anfrage *(request)* an den *Web-Server*, der als Reaktion meist eine HTML-Seite über HTTP an den *Browser* schickt
  - Intranet
  - Extranet
  - VPN.

LE 30 5

3.10.2 Web-Architekturen & Unternehmenslsg.

- Vorteile von Web-Anwendungen
  - + *Web-Browser* bekannt, reduziertes Training
  - + *Client*-CS: Nur BS & *Web-Browser* zu installieren
    - Alle anderen Komponenten liegen auf einem oder wenigen *Servern*
  - + Vereinfachter Zugang für Kunden oder Lieferanten
    - Es müssen keine zusätzlichen Infrastrukturen geschaffen werden
  - + Reduzierte Hardware-Anforderungen an die *Client*-CS, da sie nur noch den *Web-Browser* ausführen müssen
- *Web-Server*
  - Auf dem *Server* eines Unternehmens muss ein *Web-Server* laufen.

LE 30 6

3.10.2 Web-Architekturen & Unternehmenslsg.

- *Server*seitige Web-Konzepte
  - Basieren vielfach auf Skripten
  - Statt einer einfachen HTML-Seite ruft der *Web-Browser* (ausgelöst z.B. durch das Anklicken eines *Hyperlinks)* ein Skript auf dem *Server* auf und übergibt gleichzeitig Parameter an das Skript
  - Ein Skript kann man sich als Prozedur vorstellen, die auf dem *Web-Server* als Reaktion auf die Anfrage gestartet wird und die vom *Browser* übermittelten Parameter erhält
  - Das Skript führt eine Reihe von Aktionen durch und produziert eine Ausgabe
  - Diese Ausgabe – meist eine HTML-Seite – wird zum *Browser* geschickt und dem Benutzer angezeigt.

LE 30 7

3.10.2 Web-Architekturen & Unternehmenslsg.

- *Client*seitige Web-Konzepte
  - Kleine Programme, die in eine HTML-Seite eingebettet sind und vom *Browser* ausgeführt werden
  - Beispiel
    - Bei der Eingabe von Daten in HTML-Formulare ist es häufig sinnvoll, die Eingaben zu prüfen, bevor sie an den *Web-Server* geschickt werden
    - Mit einem *client*seitigen Skript lässt sich oft eine bessere Reaktionszeit der Anwendung erzielen
    - Außerdem wird die Netzbelastung gesenkt, als wenn fehlerhafte Daten erst zum *Server* geschickt und Fehlermeldungen wieder zum *Browser* übertragen werden müssen.

LE 30 8

3.10.2 Web-Architekturen & Unternehmenslsg.

- Skripte
  - Werden meist in Skript-Sprachen programmiert
  - Es handelt sich um Interpreter-Sprachen, bei deren Konzeption mehr auf einfache Benutzbarkeit als z.B. auf Typsicherheit Wert gelegt wurde
- *Client*-Seite
  - JavaScript & Visual Basic Script
- *Server*-Seite
  - *Servlets*, JSP, ASP
  - Viele *Web-Server* können durch *Plugins* um beliebige Skript-Konzepte erweitert werden
  - *Servlets* sind keine Skripte im engeren Sinne, da es sich um Java-Programme handelt.

LE 30
9

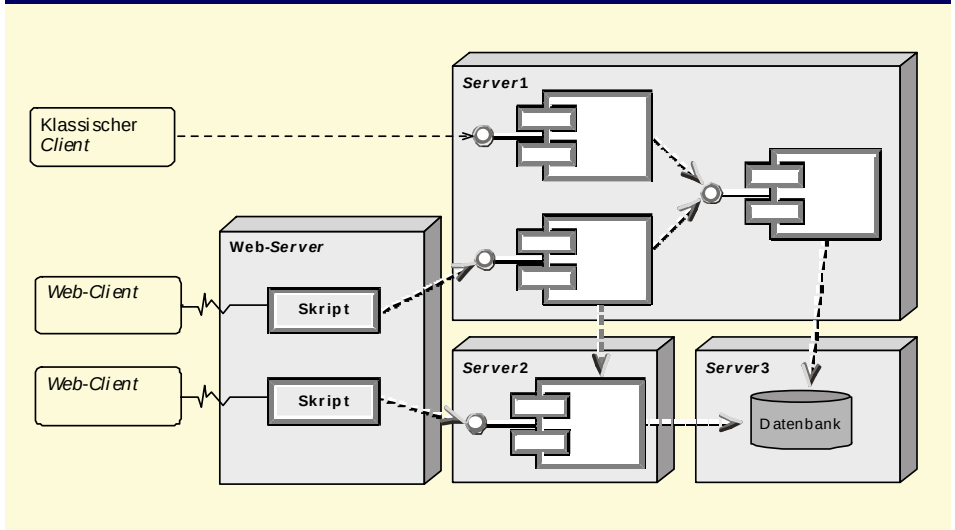
3.10.2 Web-Architekturen & Unternehmenslsg.

- **Probleme mit Skripten**
  - Für umfangreiche Programme **nicht** geeignet
  - Skripte sind über viele Quellen (z.B. HTML-Seiten) verteilt, was die **Pflege erschwert**
- **Komponentenbasierte Konzepte**
  - Funktionalität & Datenhaltung mit *server*seitigen Komponenten-Plattformen entwickeln
  - Statt einer *Client*-Anwendung wird ein ***Web-Server*** an die Unternehmenslösung angeschlossen
  - Anwendungsschicht muss **nicht** verändert werden
  - *Clients* mit klassischen Benutzungsoberflächen und *Web-Clients* können **parallel** eingesetzt werden.

LE 30
10

3.10.2 Web-Architekturen & Unternehmenslsg.

- **Aufbau einer Web-Architektur**



LE 30
11

3.10.2 Web-Architekturen & Unternehmenslsg.

- **Anbindung eines *Web-Servers*...**
  - an eine komponentenbasierte Unternehmenslösung erfolgt über die Skript-Programme
  - Ein Skript-Programm greift nicht mehr direkt auf die Datenbank zu
  - Stattdessen nutzt es Dienstleistungen von Geschäftsobjekten der Anwendungsschicht, die es über Operationsaufrufe anspricht
  - Umgebungen für *server*seitige Skriptsprachen stellen die erforderlichen Mechanismen bereit, um aus Skripten heraus Geschäftsobjekte erzeugen oder ansprechen zu können.

LE 30
12

3.10.3 *Servlets*

- ***Web-Server* greift auf Anwendungs-*Server* zu**
  - Es muss möglich sein, den *Web-Server* um Dienste zu erweitern, die den Zugriff auf einen bestimmtem Anwendungs-*Server* ermöglichen
- **Java-*Servlets***
  - Ermöglichen einen solchen Zugriff
  - Es gibt ein API, das die relevanten Schnittstellen und einige Standardimplementierungen enthält
  - *Servlets* machen *server*seitigen Code für web-basierte *Clients* nutzbar, indem sie Web-Entwicklern einen einfachen Mechanismus zur Erweiterung der Funktionalität eines Web-*Servers* zur Verfügung stellen
  - Besitzen hohe Effizienz.

LE 30
13

3.10.3 *Servlets*

- ***Servlet Engine***
  - Spezieller Container
  - Übersetzt Anfragen, die nach einem bestimmten Protokoll gebildet wurden, in ein Objekt, das dem *Servlet* übergeben wird
  - Zusätzlich wird dem *Servlet* ein weiteres Objekt für seine Antwort übergeben
  - Die *Servlet Engine* erzeugt nun aus diesem Objekt eine dem entsprechenden Protokoll genügende Antwort
  - Zusätzlich regelt die *Servlet Engine* den Lebenszyklus der *Servlets*
  - Verschiedene Hersteller bieten Implementierungen der *Servlet Engine* an.

LE 30
14

3.10.3 *Servlets*

- ***Servlet*-Definition der Firma Sun /Sun 99a/:**
  - *»A Servlet is a web component, managed by a container, that generates dynamic content*
  - *Servlets are small, plattform independent Java classes compiled to an architecture neutral bytecode that can be loaded dynamically into and run by a web Server*
  - *Servlets interact with web Clients via a request response paradigm implemented by the Servlet container*
  - *This request-response model is based on the behavior of the Hypertext Transfer Protocol (HTTP)«.*

SemorgFormular.html - Editor

Datei Bearbeiten Suchen ?

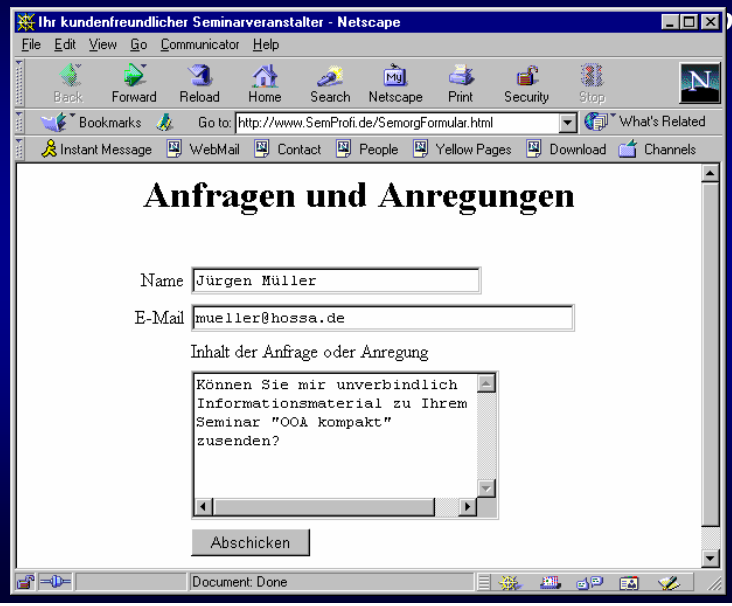
```
<HTML>
<HEAD>
<TITLE>Ihr kundenfreundlicher Seminarveranstalter</TITLE>
</HEAD>

<BODY>
<FORM ACTION="http://localhost:8080/servlet/Anfrage" METHOD="POST">
<CENTER>
<H1>Anfragen und Anregungen</H1>
<BR>
<TABLE WIDTH="50%">
<TR>
        <TD ALIGN="right">Name</TD>
        <TD><INPUT TYPE="Text" NAME="Name" ALIGN="LEFT" SIZE="30"></TD>
</TR>
<TR>
        <TD ALIGN="right">E-Mail</TD>
        <TD><INPUT TYPE="Text" NAME="EMail" ALIGN="LEFT" SIZE="40"></TD>
</TR>
<TR>
        <TD></TD>
        <TD ALIGN="left">Inhalt der Anfrage oder Anregung</TD>
</TR>
<TR>
        <TD></TD>
        <TD ALIGN="left"><TEXTAREA NAME="Inhalt" COLS="30" ROWS="6"></TEXTAREA><TD>
</TR>
<TR>
        <TD></TD>
        <TD ALIGN="left"><INPUT TYPE="Submit" NAME="AbschickenKnopf" VALUE="Abschicken"></TD>
</TR>
</TABLE>
</CENTER>
</FORM>
</BODY>
</HTML>
```


LE 30
16

3.10.3 *Servlets*

owser



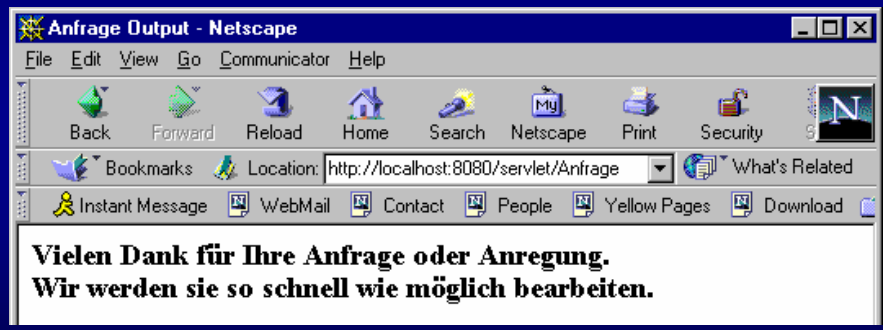
LE 30
17

3.10.3 *Servlets*

LE 30
18

3.10.3 *Servlets*

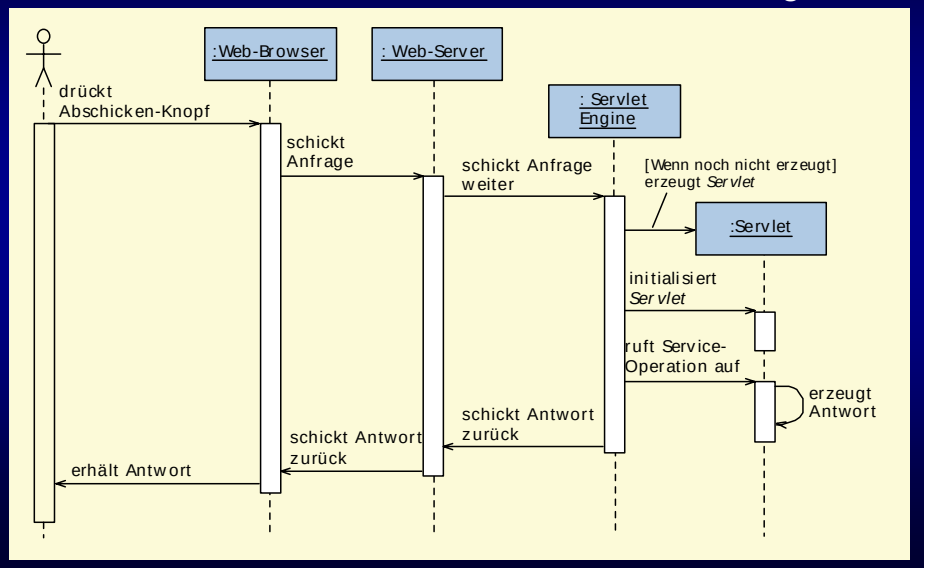
- Die Bestätigung für den Kunden wird als HTML-Seite vom *Servlet* erzeugt



LE 30
19

3.10.3 *Servlets*

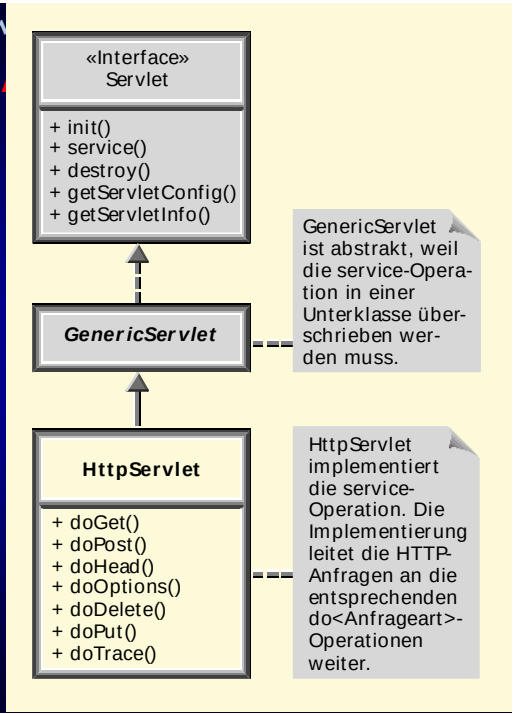
- Die internen Abläufe bei einer Kundenanfrage



LE 30
20

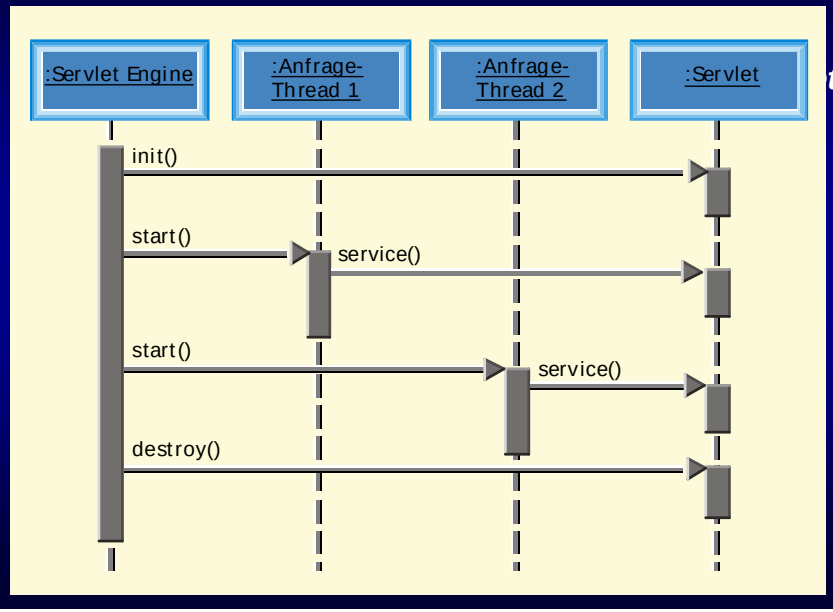
3.10.3.1 Das *Servlet-*

- Wichtige Klassen & Schnittstellen des *Servlet*-API



LE 30
21

3.10.3.1 Das *Servlet*-API



LE 30
22

3.10.3.1 Das *Servlet*-API

- **Erzeugen und Initialisieren**
  - **Beim Start der *Servlet Engine* oder bei einer ersten Anfrage an ein *Servlet*, erzeugt die *Servlet Engine* ein uninitialisiertes *Servlet*-Objekt**
  - **Bevor Anfragen bearbeitet werden können, muss als erstes das *Servlet*-Objekt initialisiert werden**
  - **Hierzu wird die Operation `init()` aufgerufen**
    - **Alle notwendigen Initialisierungen werden durchgeführt, wie das Lesen von Konfigurationsinformationen oder das Laden von kostenintensiven Ressourcen.**

LE 30
23

3.10.3.1 Das *Servlet*-API

- **`ServletConfig`-Schnittstelle**
  - **Die *Servlet Engine* übergibt der `init()`-Operation ein Objekt, das die `ServletConfig`-Schnittstelle implementiert**
  - **Über die Operationen der `ServletConfig`-Schnittstelle können Konfigurationsparameter für das *Servlet* ausgelesen und auf ein Kontext-Objekt zugegriffen werden, das die Schnittstelle `ServletContext` implementiert**
  - **Mit Hilfe der `ServletContext`-Operationen kann das *Servlet* mit der *Servlet Engine* kommunizieren.**

LE 30
24

3.10.3.1 Das *Servlet*-API

- **Beispiel**
  - **`init()`-Operation des Anfrage-*Servlets* der »Seminarorganisation« liest mit Hilfe der Schnittstelle `ServletConfig` 2 Parameter aus**
  - **1. Parameter `EMailSupport` beinhaltet als Wert die *e-mail*-Adresse, an die die Anfrage weitergeleitet werden soll**
  - **2. Parameter `SmtpServer` gibt die Netzwerkadresse des *Mail-Servers***
  - **Die Parameter werden mit Hilfe der Operation `ServletConfig::getInitParameter` ausgelesen.**

LE 30
25

3.10.3.1 Das *Servlet*-API

```
public void init(ServletConfig config) throws
   ServletException
{  super.init(config);
   //Lesen der Adresse des SMTP-Servers
   SmtpServer =
   config.getInitParameter("SmtpServer");
   //Lesen der Mail-Adresse desKundensachbearbeiters
   EMailSupport =
   config.getInitParameter("EMailSupport");
}.
```


LE 30
26

3.10.3.1 Das *Servlet*-API

- Bearbeiten von Anfragen
  - Trifft eine Anfrage ein, so weist ihr die *Servlet Engine* einen eignen *Thread* zu, der die Operation `service()` des *Servlets* aufruft
  - `service()` bekommt Referenzen auf ein Anfrage-Objekt vom Typ `ServletRequest` und ein Antwort-Objekt vom Typ `ServletResponse` übergeben
  - Das *Servlet* kann nun mittels des Anfrage-Objektes zunächst Informationen über den Typ der Anfrage, Parameter usw. erfragen, bevor es dann mit Hilfe des Antwort-Objektes eine Antwort formuliert.

LE 30
27

3.10.3.1 Das *Servlet*-API

- Beispiel
  - Das Anfrage-*Servlet* der »Seminarorganisation« erbt von der Klasse `HttpServlet`
  - `HttpServlet` implementiert bereits die Operation `service()`
  - HTTP-Anfragen werden ausgewertet und an entsprechende Operationen der Klasse `HttpServlet` weitergeleitet
  - Soll eine HTTP-Anfrageart unterstützt werden, so ist lediglich die korrespondierende Operation der Klasse `HttpServlet` zu überschreiben.

LE 30
28

Das Wichtigste zu HTTP

- HTTP-Anfrage
  - Besteht in der Regel aus mehreren Teilen:
    - *General-Header* / *Request-Line* / *Request-Header* / *Entity-Header* / *Entity-Body*
  - *Header*-Anteile sind optional und enthalten Metainformationen, wie z.B. den vom *Client* verwendeten Zeichensatz.
  - *Request-Line* enthält die Zugriffsmethode, die URL einer Ressource (Datei, Programm), sowie die Nummer der verwendeten HTTP-Version
  - Beispiel:
    - `GET http://www.software-technik.de HTTP/1.0.`

LE 30
29

Das Wichtigste zu HTTP

- HTTP-Zugriffsoperationen
  - HTTP kennt folgende Zugriffsoperationen:
    - `GET, HEAD, POST, PUT, DELETE`
  - Aufbau einer GET-Anfrage
    - *Server* bekommt eine URL zugeschickt
    - Antwort: Dokument, das die URL bezeichnet
    - Wird durch die URL ein Programm bezeichnet, das dynamisch Daten erzeugt, dann wird die Programmausgabe als Antwort zurückgeschickt und nicht das Programm selbst
  - Es können auch Parameter und ihre Werte zum *Server* geschickt werden
    - Die Daten werden als Parameter/Werte-Paare an die URL angehängt.

LE 30
30

Das Wichtigste zu HTTP

- Beispiel
  - `http://de.altavista.com/cgi-bin/query?pg=q&sc=on&q=Software-Technik&kl=de&what=de`
  - Mit Hilfe einer bekannten Suchmaschine soll nach dem Begriff »Software-Technik« gesucht werden
  - Der Suchbegriff wird als Wert des Parameters `q` vom *Web-Browser* zusammen mit den Parametern `pg, sc, kl` und `what` und ihren Werten automatisch an die URL `de.altavista.com/cgi/query` angehängt
  - Getrennt wird die URL von den Parametern durch das »?«
  - Die Parameter werden durch »&« getrennt.

LE 30
31

3.10.3.1 Das *Servlet*-API

- Klasse `HttpServlet`
  - Stellt eine HTTP-spezifische Implementierung der Schnittstelle Servlet dar
  - Die `service()`-Operation nimmt Anfragen entgegen und bestimmt zunächst ihre Art (`GET, POST, HEAD, OPTIONS, DELETE, PUT` und `TRACE`), um anschließend die entsprechende `do<Anfrageart>`-Operation aufzurufen
  - Will man mit einem *Servlet* eine bestimmte Anfrageart unterstützen, dann muss man die ihr entsprechende `do<Anfrageart>`-Operation in einer Unterklasse überschreiben
  - `doGet()`-Operation behandelt eine HTTP-`GET`- und die `doPost()`-Operation eine HTTP-`POST`-Anfrage.

LE 30
32

3.10.3.1 Das *Servlet*-API

- Beispiel
  - Das *Servlet* `Anfrage` der »Seminarorganisation« wird als Unterklasse von `HttpServlet` realisiert
  - Es wird die Operation `doPost()` überschrieben

```
import java.io.*;
import javax.servlet.*; //Wegen ServletException
//Wegen Klasse javax.servlet.HttpServlet
import javax.servlet.http.*;...
//Anfrage als Unterklasse von HttpServlet
public class Anfrage extends HttpServlet
{...  //doPost-Operation ueberschreiben
   public void doPost(HttpServletRequest req,
   HttpServletResponse resp)
   throws ServletException, IOException
   {...} }.
```

LE 30
33

3.10.3.1 Das *Servlet*-API

- HTTP-Anfragen enthalten Parameter und ihre Werte, die über Operationen der `HttpServletRequest`-Schnittstelle ausgelesen werden können
- Die Operation `getParameter()` liefert zu einem Parameternamen den assoziierten Wert
- Die Schnittstelle bietet zusätzlich Operationen, um weitere HTTP-spezifische Informationen der HTTP-Anfrage zu ermitteln
- Beispiel
  - Es sollen in der Operation `doPost()` die Parameterwerte mit Hilfe der Operation `HttpServletRequest::getParameter()` ausgelesen werden, die der Kunde in die Formularfelder im *Web-Browser* eingegeben hat.

LE 30
34

3.10.3.1 Das *Servlet*-API

```
public void doPost(HttpServletRequest req,
   HttpServletResponse resp)
throws ServletException, IOException
{...  //Zunächst werden die Werte der Eingabefelder
      //gelesen, die in der HTTP-Anfrage als
      //Parameter/Wert-Paar gespeichert werden, und
      //schreiben sie als E-Mail-Text auf.
  String EMail, Name, Inhalt;
  Name = req.getParameter("Name");
  EMail = req.getParameter("EMail");
  Inhalt = req.getParameter("Inhalt");
   ...
}.
```


LE 30
35

3.10.3.1 Das *Servlet*-API

- Schnittstelle `HttpServletResponse` enthält Operationen, mit deren Hilfe eine Antwort an den *Web-Browser* zurückgeschickt werden kann
- Als erstes muss hierzu die Operation `HttpServletResponse::setContentType()` aufgerufen werden, um den Typ des Inhalts der zurückgeschickten Antwort festzulegen, den MIME-*type (Multipurpose Internet Mail Extensions type)*
- Beispiel
  - Antwort auf die Kundenanfrage: Bestätigung als HTML-Text zurückgeschicken:

```
public void doPost(HttpServletRequest req,
   HttpServletResponse resp)
   throws ServletException, IOException
{  resp.setContentType("text/html");...}.
```


LE 30
36

3.10.3.1 Das *Servlet*-API

- Anschl. kann mit der Operation `HttpServletResponse::getWriter()` eine Referenz auf ein Objekt der Klasse `PrintWriter` angefordert werden
- Mit Hilfe der Operation `PrintWriter::println` kann 1 Zeile Text als Antwort zurückgeschickt werden
- Beispiel
  - Das *Servlet* `Anfrage` soll eine Bestätigung als HTML-Seite an den Kunden zurückschicken

```
public void doPost(HttpServletRequest req,
   HttpServletResponse resp)
   throws ServletException, IOException
{  resp.setContentType("text/html");
   PrintWriter out = resp.getWriter();... .
```

LE 30
37

3.10.3.1 Das *Servlet*-API

```
//Eine HTML-Bestätigung an den Kunden schicken.
out.println("<HTML>");
out.println("<HEAD><TITLE>Anfrage
   Output</TITLE></HEAD>");
out.println("<BODY>");
out.println("<H3>Vielen Dank für Ihre Anfrage oder
   Anregung.");
out.println("<BR>Wir werden sie so schnell wie
   m&ouml;glich bearbeiten.</H3>");
out.println("</BODY>");
out.println("</HTML>");
out.close();
}.
```


LE 30
38

3.10.3.2 Sitzungen und Sitzungsverfolgung

- Mechanismen für die Sitzungsverfolgung
  - URL *rewriting*
    - Daten zur Identifikation werden an die URL als Parameter angehängt
    - Bei jeder `GET`-Anfrage werden nun die Identifikationsdaten mitgeschickt
    - Nachteil: Häufig dienen zur Identifikation einer Sitzung sensible Nutzerdaten, wie Benutzerkennung und Passwort, die nun offen im *Web-Browser* als URL-Anhängsel angezeigt werden.

LE 30
39

3.10.3.2 Sitzungen und Sitzungsverfolgung

- Versteckte HTML-Formularfelder
  - Ein ähnlicher Ansatz basiert auf der HTTP-`POST`-Anfrage
  - Zur Sitzungsverfolgung werden versteckte HTML-Formularfelder verwendet
  - Auch hier werden wieder Identifikationsdaten als Parameter mit der Anfrage zugeschickt
  - Sensible Nutzerdaten werden nicht so offenkundig »zur Schau gestellt« wie bei dem URL *rewriting*
  - Jedoch kann nach wie vor jeder, der an den Daten interessiert ist, sich den HTML-Quellcode der Anfrage anschauen und somit ebenfalls die Daten.

LE 30
40

3.10.3.2 Sitzungen und Sitzungsverfolgung

- Versteckte *Cookies*
  - Kleine Textdateien, in denen Parameter/Werte-Paare gespeichert werden
  - Soll ein *Cookie* mit Daten erzeugt werden, dann sendet der *Web-Server* in einer HTTP-Antwort Anweisungen an den *Web-Browser*, ein *Cookie* mit einem bestimmten Namen und Wert zu erzeugen
  - Schickt der *Web-Browser* nun eine Anfrage an den *Web-Server*, so wird das assoziierte *Cookie* mitgeschickt
  - Ein *Servlet* kann nun z.B. die Anfrage in einen Sitzungskontext einordnen.

LE 30
41

3.10.3.2 Sitzungen und Sitzungsverfolgung

- Java-*Servlet*-API
  - `HttpSession`-Klasse bietet komfortable Möglichkeit, um Sitzungen zu verwalten
  - *Servlet*-Programmierer muss sich nicht darum kümmern, ob *Cookies* oder URL *rewriting* eingesetzt wird, wenn *Cookies* vom empfangenden *Web-Browser* nicht akzeptiert werden
  - Referenz auf das mit einer Anfrage assoziierte `HttpSession`-Objekt:
    - `httpServletRequest::getSession()`
  - Beliebige Objekte können mit der Sitzung bzw. dem HttpSession-Objekt verbunden werden durch
    - `HttpSession::putValue()`
  - Beenden Sitzung: `HttpSession::invalidate()`.

LE 30
42

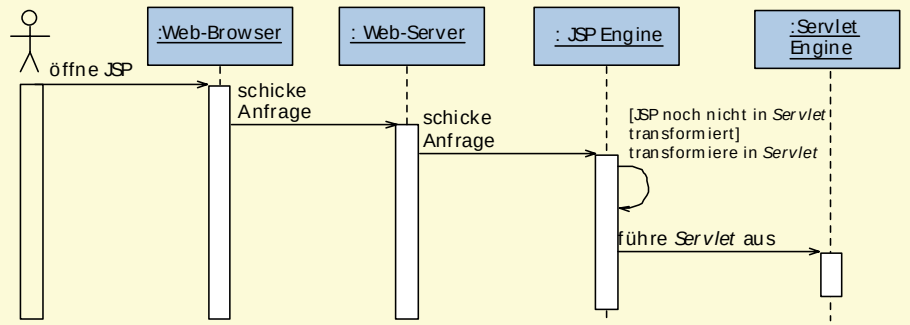
3.10.4 *Java Server Pages*

- *Java Server Page* (JSP)
  - Textdatei, die HTML- oder XML-Anteile mit speziellen JSP-Markierungen *(tags)* vermischt
  - Wenn ein *Client* auf eine JSP zugreifen möchte, so wird die Seite, falls noch nicht geschehen, von der so genannten JSP *Engine* zunächst in ein *Servlet* übersetzt und anschl. ausgeführt
  - Die Ausgabe dieses *Servlets* wird dann an den *Client* zurückgeschickt.

LE 30
43

3.10.4 *Java Server Pages*

- Vorgänge, die beim Anfordern einer JSP ablaufen



LE 30
44

3.10.4 *Java Server Pages*

- Beispiel
  - JSP liefert als Ausgabe eine Web-Seite, die den Text »Hello World« enthält:

```
<HTML>
<HEAD> <TITLE> Hello World Beispiel </TITLE> </HEAD>
<BODY>
<!-- globale Informationen der Seite setzen -->
<%@ page language="java" %>
<!-- Deklaration einer Zeichenkette HelloWorld -->
<%! String HelloWorld; %>
<!-- Nun ein Scriptlet mit Java programmieren -->
<% HelloWorld = "Hello World"; %>
<!-- Die Zeichenkette als Ausgabe in die HTML-Seite
  schreiben -->
<%= HelloWorld %> </BODY> </HTML>.
```

LE 30
45

3.10.4 *Java Server Pages*

- Elemente einer JSP
  - **Direktiven** dienen dazu, globale Informationen für die ganze Seite zu setzen
  - **Deklarationen** erlauben es, Variablen und Operationen zur deklarieren, die eine seitenweite Gültigkeit besitzen
  - ***Scriptlets*** enthalten den Java-Quellcode, der für die Berechnung von Ausgaben notwendig ist
  - **Ausdrücke** enthalten Variablen, deren Werte, in eine Zeichenkette konvertiert, von der *JavaServer Page* ausgegeben werden
  - →vordefinierte Objekte.

LE 30
46

JSP-Syntax

- Ausgabe Kommentar *(output comment)*
  - Der Kommentar wird mit in die Ausgabe geschrieben
  - `<!--comment [<%= expression %>] -->`
- Versteckter Kommentar *(hidden comment)*
  - Der Kommentar erscheint nur in der JSP
  - `<%-- comment --&>`
- Deklaration
  - Deklariert Operationen und Variablen in der korrekten Skriptsprache (z.B. Java)
  - `<%! declarations %>.`

LE 30
47

JSP-Syntax

- Ausdruck
  - Der angegebene Ausdruck wird – in eine Zeichenkette umgewandelt – von der JSP ausgegeben
  - `<%= expression %>`
- *Scriptlet*
  - Ein in der Skriptsprache formuliertes Quellcodefragment
  - `<% code fragment %>`
- *include*-Direktive
  - Fügt eine Text- oder Quellcodedatei in die JSP-Datei ein
  - `<%@ include file="relativeURL" %>.`

LE 30
48

JSP-Syntax

- *page*-Direktive
  - Globale Einstellungen für die gesamte JSP

```
<%@ page [ language=" java " ]
[ extends=" package. class " ]
[ import= "
{ package. class | package .* } , ... " ]
[ session=" true |false" ]
[ buffer=" none| 8kb | size kb" ]
[ autoFlush=" true |false" ]
[ isThreadSafe=" true |false" ]
[ info=" text " ]
[ errorPage=" relativeURL "]
[ contentType=" mimeType [ ; charset =characterSet
  ]"|" text/ html ; charset= ISO- 8859- 1 "]
[ isErrorPage=" true| false "] %>.
```

LE 30
49

JSP-Syntax

- *taglib*-Direktive
  - Markierungen, die von einer JSP *Engine* verarbeitet werden, können durch eine sog. *tag*-Bibliothek erweitert werden
  - Über *taglib* ist eine *tag*-Bibliothek einbindbar

```
<%@ taglib uri=" URIToTagLibrary " prefix="
tagPrefix "%>
custom tag:
< tagPrefix : name attribute =" value "+ ... />
< tagPrefix : name attribute =" value "+ ... > other
  tags </ tagPrefix : name >
```

- `<jsp:forward>`
  - Leitet eine *Client*-Anfrage weiter

```
<jsp: forward page=
"{ relativeURL | <%= expression %> }" />.
```


LE 30
50

JSP-Syntax

- `<jsp:getProperty>`
  - Liefert den Eigenschaftswert einer *JavaBean*, so dass er in der JSP benutzt werden kann
  - `<jsp: getProperty name=" beanInstanceName " property=" propertyName "/>`
- `<jsp:include>`
  - Fügt Daten aus einer externen Datei in die JSP ein
  - Daten werden nicht durch einen Parser überprüft
  - `<jsp: include page="{ relativeURL | <%= expression %> }" flush=" true" />.`

LE 30
51

JSP-Syntax

- `<jsp:plugin>`
  - Veranlasst das Herunterladen eines *Applets* oder einer *JavaBean* auf das *Client*-Computersystem

```
<jsp: plugin type=" bean| applet" code=" classFileName
  " codebase=" classFileDirectoryName "
[ name=" instanceName " ]
[ archive=" URIToArchive, ... " ]
[ align=" bottom |top| middle| left| right" ]
[ height="displayPixels "] [ width=" displayPixels " ]
[ hspace=" leftRightPixels " ]
[ vspace=" topBottomPixels "]
[ jreversion=" JREVersionNumber | 1.1 " ]
[ nspluginurl=" URLToPlugin " ] [ iepluginurl="
  URLToPlugin "] >.
```


LE 30
52

JSP-Syntax

```
[ <jsp: params> [ <jsp: param name=" parameterName "
  value=" parameterValue " /> ]+ </ jsp: params> ]
[ <jsp: fallback> text message for user </ jsp:
  fallback> ]
</ jsp: plugin>
```

- `<jsp :setProperty>`
  - Setzt den Wert einer *JavaBean*-Eigenschaft

```
<jsp: setProperty name=" beanInstanceName "
{ property= "*"| property=" propertyName "
[ param=" parameterName "]|
property=" propertyName " value=" { string | <%=
  expression %>}"}/>.
```

LE 30
53

JSP-Syntax

- `<jsp:useBean>`
  - Über diese Markierung kann eine *JavaBean* für die JSP nutzbar gemacht werden.

```
<jsp: useBean id=" beanInstanceName " scope=" page
  |request| session| application"
{ class=" package. class " | type=" package. class " |
  class=" package. class " type=" package. class "|
beanName=" { package. class | <%= expression %> }
" type=" package. class "}
{/>| > other tags </ jsp: useBean> }.
```


LE 30
54

3.10.4 *Java Server Pages*

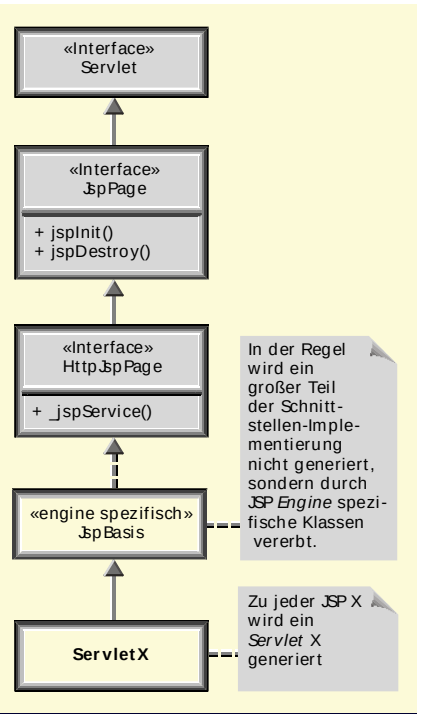
- Neben den vordefinierten Markierungen *(tags)* existieren eine Reihe vordefinierter Objekte, die in den *Scriptlets* benutzt werden können

Implizites Objekt	Typ
`request`	Unterklasse von `javax.servlet.ServletRequest`
`response`	Unterklasse von `javax.servlet.ServletResponse`
`pageContext`	`javax.servlet.jsp.PageContext`
`session`	`javax.servlet.http.HttpSession`
`application`	`javax.servlet.ServletContext`
`out`	`javax.servlet.jsp.JspWriter`
`config`	`javax.servlet.ServletConfig.`

LE 30
55

3.10.4 *Java Server Pages*

- Für JSPs relevante Schnittstellen und Klassen
  - *Scriptlets* werden in die Operation `_jspService` des *Servlets* eingefügt
  - Die in den Deklarationen eingeführten Operationen und Variablen werden in Operationen und Variablen des *Servlets* übersetzt.



LE 30
56

3.10.4 *Java Server Pages*

- Beispiel
  - Der folgende Quellcode stellt dar, wie z.B. die JSP `HelloWorld` in ein *Servlet* übersetzt wird
  - Zunächst wird die in der JSP-Deklaration eingeführte Variable als Attribut der *Servlet*-Klasse deklariert
  - In der Operation `_jspService` werden als erstes die vordefinierten Objekte initialisiert
  - Anschl. wird der eingefügte *Scriptlet*-Quellcode ausgeführt
  - Zum Schluss wird noch der in der JSP angegebene Ausdruck `<%= HelloWorld %>` ausgegeben
  - Er wird in die Anweisung `out.print(HelloWorld)` umgesetzt.

LE 30 57

3.10.4 *Java Server Pages*

```
public class
  jsp_0005cHelloWorld_0005cHelloWorld_0002ejspHelloWor
  ld_jsp_1 extends HttpJspBase
{   static char[][] _jspx_html_data = null;
    // begin [file="D:\\LE30\\jswdk
    //-1.0.1\\examples\\jsp\\HelloWorld\\
      HelloWorld.jsp";from=(11,3);to=(11,23)]
       String HelloWorld;
      // end    ...
    public final void _jspx_init() throws JspException
    { ...   }
    public void _jspService(HttpServletRequest
            request, HttpServletResponse  response)
            throws IOException, ServletException
    {...PageContext pageContext = null;
        HttpSession session = null;.
```


LE 30 58

3.10.4 *Java Server Pages*

```
ServletContext application = null;
ServletConfig config = null;
JspWriter out = null;
Object page = this;
...
try { ...
response.setContentType("text/html");
pageContext = _jspxFactory.getPageContext
(this, request,response,"", true, 8192, true);
application = pageContext.getServletContext();
config = pageContext.getServletConfig();
session = pageContext.getSession();
out = pageContext.getOut();
// begin [file="D:\\LE30\\jswdk
//-1.0.1\\examples\\jsp\\HelloWorld\\
  HelloWorld.jsp";from=(14,2);to=(14,31)].
```


LE 30 59

3.10.4 *Java Server Pages*

```
            HelloWorld = "Hello World";
            // end
        ...
           // begin
  [file="D:\\LE30NeueNumerierung\\jswdk
  //-1.0.1\\examples\\jsp\\HelloWorld\\
  HelloWorld.jsp";from=(17,3);to=(17,15)]
             out.print( HelloWorld );
           // end
          ...
        }
      ...
    }
}.
```


LE 30 60

3.10.4 *Java Server Pages*

- **Resümee: JSPs**
  - JSPs sind eine **sinnvolle Erweiterung** von *Servlets*
  - JSPs sind aber **kein** *Servlet*-Ersatz
  - Nachteile
    - Durch Vermischung von HTML & Skriptsprache werden JSPs schnell **unübersichtlich**
    - Das Aufspüren von Syntax- und Laufzeitfehlern der *Scriptlets* kann **schwierig** sein
    - Möglichst viel Skriptsprachen-Quellcode in andere Komponenten *(JavaBeans)* auszulagern
    - Bei einem komponentenorientierten Ansatz hervorragend als **»Klebstoff«** verwendbar, um die einzelnen Funktionseinheiten zusammenzufügen
    - Keine großen *Scriptlets* in JSPs einbetten.

LE 30
61

3.10.5 *Active Server Pages*

- ASP *(Active Server Pages)*
  - *Server*seitiges Skript-Konzept von Microsoft
  - Setzt auf den Microsoft *Internet Information Services* (IIS) auf
  - Der *Web-Server* von Microsoft kann mit ASP umgehen
  - Eine *Active Server Page* (Dateiendung: `.asp`) ist eine HTML-Datei, die neben den üblichen Elementen, wie HTML, Bildern oder Java-*Applets*/*ActiveX*-Steuerelemente auch Skript-Programme enthält
  - Ein *Web-Browser* fordert eine ASP genau so an wie eine gewöhnliche HTML-Seite.

LE 30
62

3.10.5 *Active Server Pages*

- Die Skript-Programme werden vom *Web-Server* ausgeführt, bevor die Seite an den *Browser* übertragen wird
- Sie verändern meistens die HTML-Seite, sind in ihrer Funktionalität jedoch *nicht* darauf beschränkt
- Aus einem Skript heraus kann auf *server*seitige Komponenten (z.B. Geschäftsobjekte) zugegriffen werden
- Damit ist es z.B. möglich, über ADO-Objekte eine Datenbank-Anfrage zu stellen und das Ergebnis dieser Anfrage in Form von HTML-Code in die Seite zu integrieren
  - ADO: *ActiveX Data Objects*, eine COM-basierte, objektorientierte Schale um eine meist relationale Datenbank.

LE 30
63

3.10.5 *Active Server Pages*

- Meist benutzte Skript-Sprachen:
  - *Visual Basic Script* und *JScript*
- Grundsätzlich kann aber jede Sprache verwendet werden, die *ActiveX-Scripting* unterstützt
- Dies bedeutet, dass der Interpreter den Zugriff auf COM-Objekte durch Skript-Code ermöglichen und selbst einige Schnittstellen implementieren muss, damit der *Web-Server* ihn mit der Interpretation eines Skript-Programms beauftragen kann
- Die Skript-Programme werden zwischen spezielle Markierungen »`<% Skript-Programm %>`« geschrieben, damit der *Web-Server* feststellen kann, was gewöhnliches HTML ist, und welche Teile der ASP zu interpretieren sind.

LE 30
64

3.10.5 *Active Server Pages*

- Beispiel
  - Begrüßungstext in eine HTML-Seite
  - Je nach Tageszeit ein passender Text

```
<%@ language=VBScript %>
<html> <head>
<title>Ein Beispiel für ASP</title> </head>
<body>
<% if hour(time) > 5 and hour(time)<12 then %>
      Guten Morgen!
<% elseif hour(time)>=12 and hour(time)<18 then %>
      Guten Tag!
<% elseif hour(time)>=18 and hour(time)<=23 then %>
      Guten Abend!
<% else %> Hallo Nachtschwärmer! <% end if %>
</body> </html>.
```

LE 30
65

3.10.6 CGI

- CGI *(Common Gateway Interface)*
  - *Web-Browser* kann die Ausführung von Programmen auf einem *Web-Server* veranlassen und Parameter an die Programme übergeben
  - CGI spezifiziert die Kommunikation zwischen dem *Web-Server* und einem externen CGI-Programm (meist als CGI-Skript bezeichnet)
  - Protokoll stammt aus den Anfangszeiten des Web
    - + Herstellerunabhängiger, nicht-kommerzieller Standard
    - + Sprachen*un*abhängig, d.h. CGI-Skripte können in beliebigen Sprachen geschrieben werden, am häufigsten wird Perl verwendet.

LE 30
66

3.10.6 CGI

- + Es existieren eine Vielzahl von *Web-Servern* und Werkzeugen – viele davon als *freeware*
- – Der *Web-Server* startet für jede Ausführung eines CGI-Skriptes einen neuen Prozess auf dem *Server*
  - ○ Dies ist speicher- und zeitaufwendig
- Die schlechte *Performance* von CGI-Skripten wird dazu führen, dass diese Technik mittelfristig von den bisher vorgestellten Konzepten (*servlets*, JSP, ASP) abgelöst wird
- Derzeit ist CGI aber noch weit verbreitet.

LE 30
67

3.10.6 CGI

- Bei CGI wird eine HTTP-Anfrage des *Web-Browsers* an den *Server* als Kommandozeile genutzt
- Die URL, die der *Browser* an den *Server* schickt, bezeichnet in diesem Fall keine HTML-Seite, sondern ein ausführbares Programm
- Eine solche URL sieht z.B. so aus:
- `www.server-name.de/cgi-bin/programm-name`
- Das Verzeichnis `cgi-bin` ist als Programmverzeichnis nicht vorgeschrieben, wird jedoch von vielen *Servern* verwendet.

LE 30
68

3.10.7 XML

- HTML
  - Markierungen sind Teil der Sprache
  - Ein Autor kann sich zwar überlegen, ob er in einem Brief den Betreff und die Anrede z.B. in eigene Absätze verlegt, jedoch kann er diese Absätze nicht direkt als Anrede bzw. Betreff kenntlich machen

```
<html> <body>
    <p>Ihre Anfrage vom 26.07.2000 bezüglich einer
Schulung </p>
    <p>Sehr geehrte Damen und Herren</p>
    <p>Der Text des Briefes, der genau wie Betreff
und Anrede als Absatz geschrieben wird
      und sich strukturell nicht von ihnen
unterscheidet </p>
</body> </html>.
```

LE 30
69

3.10.7 XML

- XML *(Extension Markup Language)*
  - Beschreibung logischer Dokumenten-Strukturen
  - Sehr geringer Sprachumfang
  - Definiert keine einzige Markierung
  - Autors überlegt sich selbst passende Markierungen
  - Element
    - Der von einer öffnenden und schließenden Markierung eingeschlossene Teil eines XML-Dokuments
    - Ein Element ohne Inhalt besteht nur aus einer einzelnen Markierung
      - `<Einschreiben/>` zeigt z.B. an, dass der Brief als Einschreiben verschickt werden soll
      - Es bedarf keines Inhalts.

LE 30
70

3.10.7 XML

- Wurzel-Element
  - Element, das alle anderen Elemente enthält
  - Am Anfang eines Dokuments
    - Angaben zur verwendeten XML-Version
  - Jedes Element kann zusätzlich Attribute enthalten, die hinter dem Element-Namen als Name-Wert-Paare geschrieben werden
  - `<Brief Druckdatum="2000-07-5">...</Brief>`
  - XML unterscheidet zwischen Groß- und Kleinschreibung.

LE 30
71

3.10.7 XML

- Beispiel: Brief als XML-Dokument

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Brief>
   <Adresse>
      <Name>Meier</Name>
      <Strasse>Elisenstraße</Strasse>
      <PLZ>80335</PLZ>
      <Ort>München</Ort>
   </Adresse>
   <Betreff>Ihre Anfrage vom 26.07.2000 bezüglich
  einer Schulung </Betreff>
   <Anrede>Sehr geehrte Damen und Herren</Anrede>
   <Text>der Text des Briefes, der sich jetzt
  strukturell eindeutig von Betreff und Anrede
  abhebt.</Text>
</Brief>.
```


LE 30
72

3.10.7 XML

- Suchanfragen möglich
  - Mit einem Suchprogramm ganz neue Anfragen möglich sind
    - Alle Briefe, die im Betreff das Wort »Schulung« enthalten
- XML-Dokument = Textdatei
  - XML-Dokumente sind wie Java-Quelltexte Unicode-Dateien
  - XML-Dokumente bedürfen der Unterstützung durch Werkzeuge, mit denen sie erstellt werden können.

LE 30
73

3.10.7.2 Dokumenttyp-Definitionen

- Dokumenttyp-Definitionen (DTDs)
  - Beschreiben die Struktur von XML-Dokumenten
  - Angabe, welche Elemente wie im Dokument
  - In das Dokument integriert oder das Dokument enthält die URL, wo die DTD steht

```
<!ELEMENT Brief(Adresse, Betreff, Anrede+, Text)>
<!ELEMENT Adresse(Firma?, Name, Strasse, PLZ, Ort)>
<!ELEMENT Name(#PCDATA)>
<!ELEMENT Strasse(#PCDATA)>
<!ELEMENT PLZ(#PCDATA)>
<!ELEMENT Ort(#PCDATA)>
<!ELEMENT Betreff(#PCDATA)>
<!ELEMENT Anrede(#PCDATA)>
<!ELEMENT Text(#PCDATA)>.
```


LE 30
74

3.10.7.2 Dokumenttyp-Definitionen

- Zum Datenaustausch
  - Auf der Basis von XML wurden inzwischen eine ganze Reihe von Austausch-Formaten spezifiziert
    - XSL *(XML Stylesheet Language)* zur Transformation von XML-Dokumenten
    - XMI *(XML Metadata Interchange)*
      - Textuelle Beschreibung von UML-Modellen
  - OASIS *(Organization for the Advancement of Structured Information Standards)*
  - Mit Hilfe von DTDs lassen sich Dokumente nur oberflächlich spezifizieren
  - Ein DTD ist selbst kein XML-Dokument, d.h. die Struktur von DTDs kann nicht mit DTDs beschrieben werden.

LE 30
75

3.10.7.3 XML-Schemata

- XML-Schemata
  - XML-basierte Sprache, die mächtigere Konstrukte zur Spezifikation von Struktur, Inhalt und Semantik von XML-Dokumenten enthält als DTDs
    - XML-Schemata sind also wieder XML-Dokumente
  - Im Bereich der Datentypen bieten XML-Schemata umfangreiche Möglichkeiten
    - Einige Datentypen sind bereits fest vorgegeben, z.B. `Date` für Datumsangaben
    - Weitere können selbst definiert werden
    - XML-Schemata werden DTDs in Zukunft ablösen.

LE 30
76

3.10.7.3 XML-Schemata

- Beispiel: XML-Schema für einen Brief

```
<schema>
<datatype name="PLZT">
   <basetype name="string"/>
   <lexicalRepresentation>
      <lexical>99999</lexical>
   </lexicalRepresentation>
</datatype>
<elementType name="PLZ">
   <datatypeRef name="PLZT"/>
</elementType>
<elementType name="Adresse">
   <sequence>
     <elementTypeRef name="Firma" minOccur="0"
                                  maxOccur="1"/>.
```

LE 30
77

3.10.7.3 XML-Schemata

```
      <elementTypeRef name="Name" minOccur="1"
                                  maxOccur="1"/>
       <elementTypeRef name="Strasse" minOccur="1"
                                  maxOccur="1"/>
       <elementTypeRef name="PLZ" minOccur="1"
                                  maxOccur="1"/>
       <elementTypeRef name="Ort" minOccur="1"
                                  maxOccur="1"/>
    </sequence>
  </elementType>
  <elementType name="Firma">
    <datatypeRef name="string"/> </elementType>
  <elementType name="Name">
    <datatypeRef name="string"/> </elementType>
  <elementType name="Strasse">
    <datatypeRef name="string"/> </elementType>.
```


LE 30
78

3.10.7.3 XML-Schemata

```
<elementType name="Ort">
  <datatypeRef name="string"/> </elementType>
<elementType name="Brief">
   <sequence>
      <elementTypeRef name="Adresse" minOccur="1"
                                  maxOccur="1"/>
      <elementTypeRef name="Betreff" minOccur="1"
                                  maxOccur="1"/>
      <elementTypeRef name="Anrede" minOccur="1"
                                  maxOccur="2"/>
      <elementTypeRef name="Text" minOccur="1"
                                  maxOccur="1"/>
   </sequence>
</elementType>.
```


LE 30
79

3.10.7.3 XML-Schemata

```
<elementType name="Anrede">
  <datatypeRef name="string"/> </elementType>
<elementType name="Text"> <mixed/> </elementType>
<elementType name="Betreff">
  <datatypeRef name="string"/> </elementType>

</schema>.
```


LE 30
80

3.10.7.4 Namensbereiche

- XML
  - Definiert selbst keine Markierungen
  - Der Autor muss sich selbst Namen ausdenken
  - Kann leicht zu Namenskonflikten führen
- Namensbereiche
  - In XML lassen sich daher Namensbereiche definieren
  - Namensbereiche müssen weltweit eindeutig sein, weshalb meistens URLs verwendet werden
  - Beispiel

```
<brief:Brief xmlns:
brief="Web.swt.ruhr-uni-bochum.de/Brief-DTD">
<brief:Adresse>..</brief:Adresse>..</brief:Brief>.
```

LE 30
81

3.10.7.5 Anzeige von Dokumenten

- **Transformation in ein HTML-Dokument**
  - In der Praxis beschreibt ein *Stylesheet* häufig eine Transformation eines XML-Dokuments in ein HTML-Dokument, das ein *Web-Browser* dann anzeigen kann
  - 2 Varianten:
    - Die Konvertierung findet auf dem *Server* statt, und es wird eine HTML-Datei an den *Browser* geschickt
    - Die Konvertierung findet erst auf dem *Client* statt
      - Es wird das XML-Dokument und das *Stylesheet* an den XML-fähigen *Browser* übertragen, der die Konvertierung selbst vornimmt
    - Welche Variante gewählt wird, hängt von der Zugriffsart ab.

LE 30
82

3.10.7.5 Anzeige von Dokumenten

- **Beispiel: Brief konvertiert nach HTML**

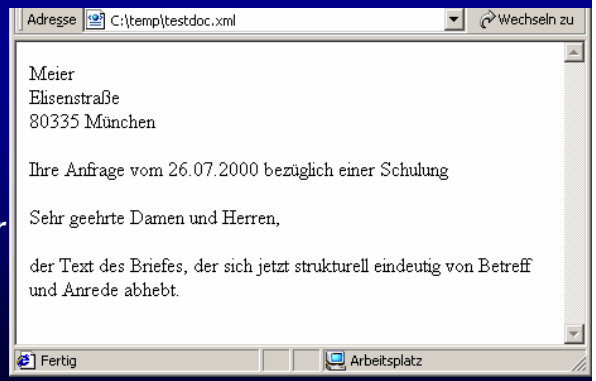
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-
  xsl">
<xsl:template match="/">
<html>
<head><title>Ein Brief</title></head>
<body>
 <p><xsl:apply-templates select="Brief/Adresse"/></p>
 <p><xsl:value-of select="Brief/Betreff"/></p>
 <p><xsl:for-each select="Brief/Anrede">
   <xsl:value-of select="."/>,<br/></xsl:for-each></p>
 <p><xsl:value-of select="Brief/Text" /></p>
</body>
</html>
</xsl:template>.
```


LE 30
83

3.10.7.5 Anzeige von Dokumenten

```
<xsl:template match="Adresse">
   <xsl:value-of select="Name"/><br/>
   <xsl:value-of select="Strasse"/><br/>
   <xsl:value-of select="PLZ"/>
   <xsl:value-of select="Ort"/>
</xsl:template>
```

- Darstellung des XML-Dokuments in einem XML-fähigen *Web-Browser*



LE 30
84

3.10.7.6 XML-*Parser*

- **XML-*Parser***
  - Programme, die ein XML-Dokument lesen und die einzelnen Markierungen herausfiltern können
  - Andere Programme können auf diese *Parser* aufsetzen und damit auf einem höheren Abstraktionsniveau arbeiten
  - Einfach Operation `liefereInhaltvonMarkierung("Betreff")` aufrufen
  - Alternative: Algorithmus formulieren, der Zeichenkette `<Betreff>` sucht und dann alle Zeichen bis zur Zeichenkette `</Betreff>` extrahiert
  - XML-Parser können als Bibliotheken entwickelt und in anderen Programmen eingesetzt werden.

LE 30
85

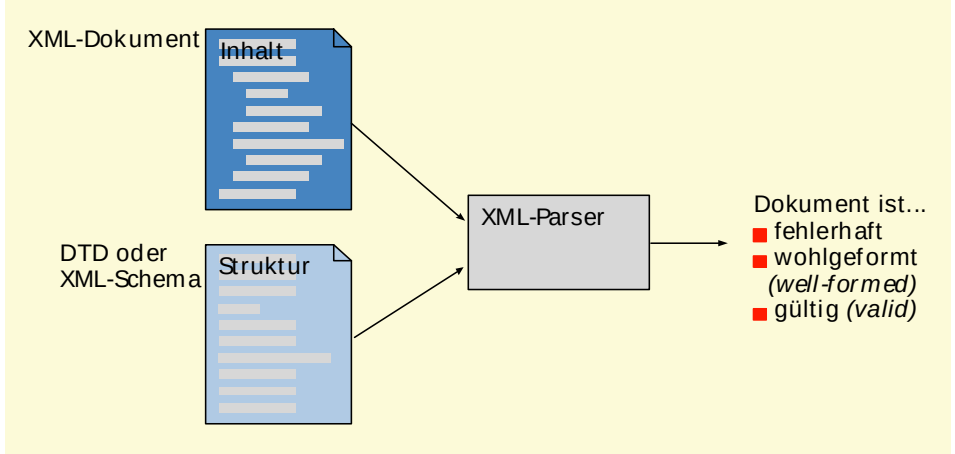
3.10.7.6 XML-*Parser*

- 2 Stufen der Korrektheit
  - XML-Parser prüfen ein Dokument beim Einlesen auf Korrektheit
  - XML-Spezifikation: 2 Stufen der Korrektheit
    - Ein Dokument ist wohlgeformt *(well-formed)*, wenn es sich an die syntaktischen Regeln von XML hält
    - Ein Dokument ist gültig *(valid)*, wenn seine Struktur den Vorgaben eines Dokument-Typs oder eines XML-Schemas entspricht
      - Das Programm kann sich bei einem gültigen Dokument darauf verlassen, dass es bestimmte Elemente in einer bestimmten Reihenfolge vorfindet, was eine Menge Ausnahmesituationen und Fehlerbehandlungen im Anwendungs-Code überflüssig macht.

LE 30
86

3.10.7.6 XML-*Parser*

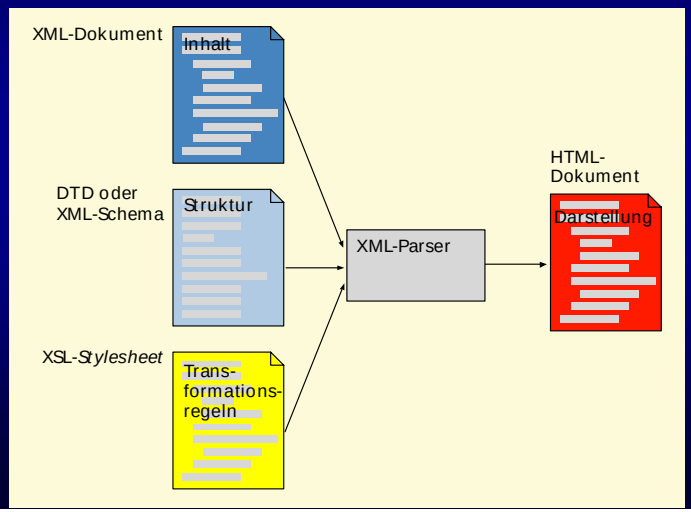
- Prüfung der Korrektheit eines Dokuments durch einen XML-*Parser*



LE 30
87

3.10.7.6 XML-*Parser*

- Konvertierung eines XML-Dokuments in ein HTML-Dokument



LE 30
88

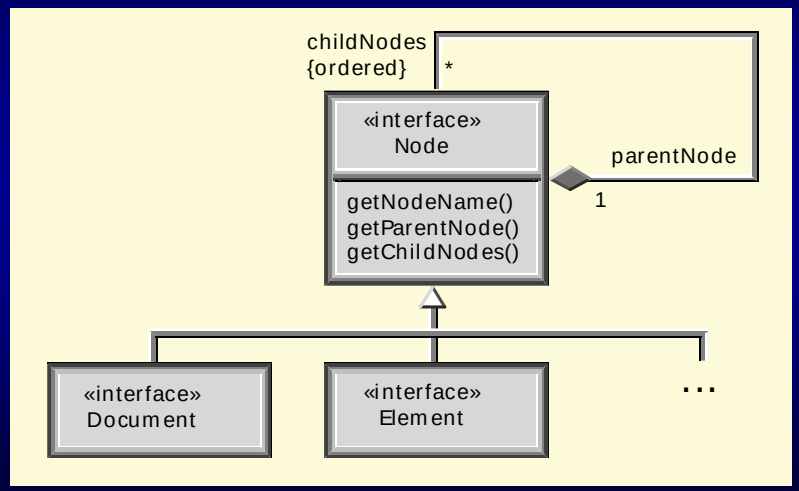
3.10.7.7 DOM

- DOM *(Document Object Model)*
  - Einheitliche und mächtige Schnittstelle zwischen XML-*Parser* und einer Anwendung
  - XML-Dokumente stellen eine Baumstruktur dar
    - Jedes Element Kind-Elemente kann enthalten
    - Jedes Element (mit Ausnahme der Wurzel) liegt in genau einem Eltern-Element
  - DOM definiert Klassen bzw. Schnittstellen, mit denen aus einer OO-Sprache auf die Baumstruktur zugegriffen werden kann
  - Aufgabe des *Parsers*
    - Zeichen eines XML-Dokuments zu interpretieren und daraus DOM-Objekte zu erzeugen, auf die eine Anwendung zugreifen kann.

LE 30
89

3.10.7.7 DOM

- Ein Ausschnitt aus der Struktur des DOM



LE 30
90

3.10.7.7 DOM

- Beispiel: XML-Dokument mit Java bearbeitet

```
import org.w3c.dom.*; import javax.xml.parsers.* ;
public class HauptKlasse
{  public static void main(String[] args)
   { DocumentBuilderFactory eineFabrik =
            DocumentBuilderFactory.newInstance();
     DocumentBuilder einErbauer =
            eineFabrik.newDocumentBuilder();
     Document einXmlDokument =
            einErbauer.parse(new File("Dateiname"));
     //eine Liste aller Elemente im Dokument anfordern
     NodeList dieKindElemente =
            einXmlDokument.getChildNodes();
     int anzahlKindElemente =
            dieKindElemente.getLength();.
```


LE 30
91

3.10.7.7 DOM

```
     //das erste Kind-Element ermitteln
     Node erstesElement = dieKindElemente.item(0);
     //den Namen des Knoten ermitteln
     String derElementName =
            erstesElement.getNodeName();
        …
   }
}.
```


LE 30
92

- Danke!
- Aufgaben

- Diese Präsentation bzw. Teile dieser Präsentation enthalten Inhalte und Grafiken des **Lehrbuchs der Software-Technik** (Band 1), **2. Auflage** von Helmut Balzert, Spektrum Akademischer Verlag, Heidelberg 2001