Verteilte Systeme
Betriebssysteme II

Kapitel 7: Webtechnologien

Prof. Dr. Wolfgang Küchlin

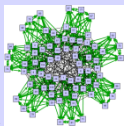
Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

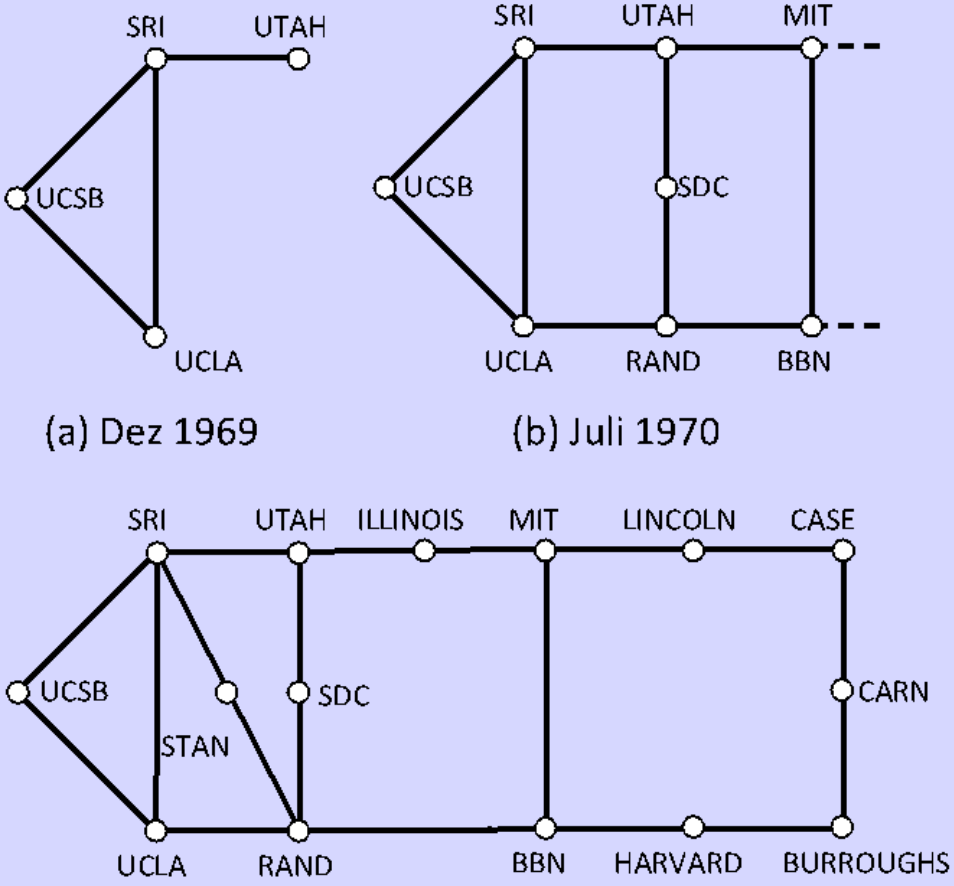
Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

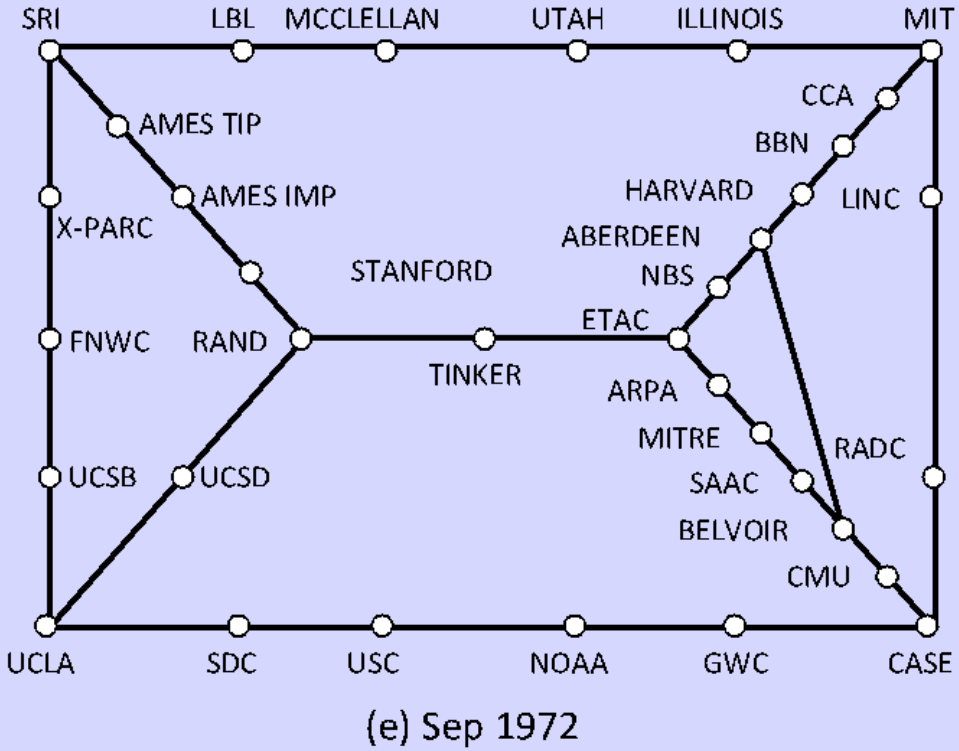
SR 

Das Web – Vorgänger: ARPANET

- 1969: Experimentelles ARPA-Netz mit Knoten an vier Universitäten
  - DARPA = Defense Advanced Research Projects Agency
- 1974: Erfindung des TCP/IP-Protokolls
  - Implementierung in Berkeley UNIX (BSD)
  - Verwendung von Sockets
- 1984: Organisation mit *Domain Naming System (DNS)*
- Ende 70er: Vom Militär unabhängiges NSFNET
  - NSF = National Science Foundation

Das Web – Vorgänger: ARPANET



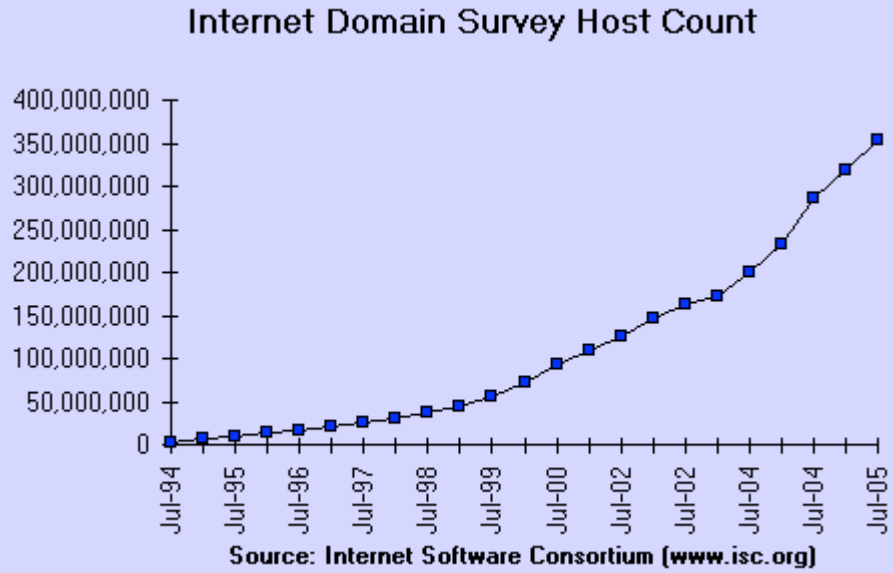


schnelles Wachstum des ARPANET (Abb.1-25, Computernetzwerke, Tanenbaum)

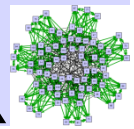
SR

Das Web – Internet

- Mitte 80er: Sammlung von Netzen wurde als ein Netzverbund (das Internet) betrachtet
- 1990: erster Browser für das World Wide Web wird entwickelt



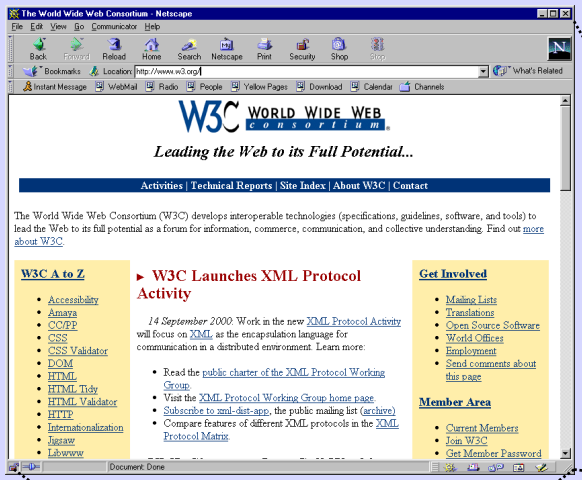
Quelle: http://www.isc.org/index.pl?/ops/ds/

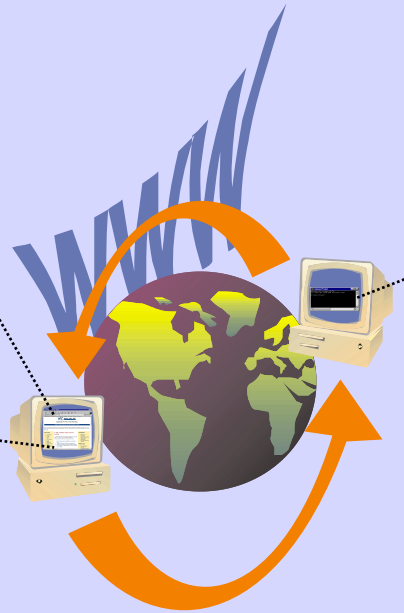


Das Web – Standards und Technologien

- grundlegend:
  - HTTP, HTML (mit Hyperlinks) und URLs
- Weiterentwicklungen
  - HTTPS, CSS, XML, URI
- Dynamische Webseiten und Webanwendungen
  - Serverseitig
    - Skriptsprachen: Perl, PHP, Python, …
    - Servlets, JSPs
  - Clientseitig
    - Java (*Applets*)
    - Javascript

Das Web – Funktionsweise

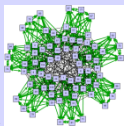




HTTP-Server

Application
Presentation
Session
Transport
Network
Data Link
Physical

Application
Presentation
Session
Transport
Network
Data Link
Physical



Markup

Markup: Informationen, die einem Dokument beigefügt werden, selbst aber nicht unmittelbar dargestellt werden.

Bsp.: HTML-Markup:

```
<h2>Markup zur Textformatierung</h2>
Unter <i>Markup</i> versteht man Informationen,
die einem Textdokument beigef&uuml;gt,
```

SGML – Standardized General Markup Language

- Ende 80er Jahre bei IBM entwickelt (Goldfarb).
- ISO Standard 1986
- Tags zur Annotation (mark-up) eines Textes
  - zur Textformatierung
    - `<i>kursiv</i>`
- Ziel: repräsentiere Industrie-Dokumentationen
  - elektronisch
  - unabhängig von konkreten Text-Satzsystemen
  - bilde auf jeweiliges Medium ab (z.B. Web oder Druck)
  - Industriequalität (Novell-Handbücher: 150.000 Seiten)
- Problem: Extrem mächtig, extrem komplex

SR

SGML, HTML, XML

- Idee prima, aber SGML zu komplex
- HTML: Formatiere Dokumente für das Web
  - Vordefinierte Menge von Tags, Semantik fixiert
  - Werden von Browsern interpretiert
  - bilden *Document Type*, der mittels SGML definiert ist
- XML: definiere Dokumentstruktur allgemein
  - minimale Sprache, extremes subset von SGML („SGML- -“)
  - erweiterbar
  - Abbildung Struktur → layout separat über style sheets
  - → auch für Datenserialisierung nützlich (ohne publishing)

HTML – Hypertext Markup Language

- HTML definiert primär Layout eines Web-Dokuments
  - nur sekundär auch Struktur
- Auszeichnungssprache
  - paarweise öffnende und schließende Tags
  - Hierarchische Gliederung
- Tags
  - zur Textformatierung
    - `<i>kursiv</i>`
  - zur Spezifikation von Hypertext-Links
    - `<A HREF="http://www.w3c.org"> W3 Konsortium </A>`
  - zum Einbinden von Multimediaobjekten und Applets
    - `<img src="logo.gif" alt="Alternativtext">`
  - für Formulare

HTML – Beispiel

```
<HTML>
  <HEAD>
    <TITLE>Verteilte Systeme</TITLE>
  </HEAD>
  <BODY>
    <H1>Gliederung</H1>
    <ul>
      <li><i>Einleitung</i></li>
      <li><b>Netze</b></li>
      <li><tt>Programmiermodelle</tt></li>
      <li>...</li>
    </ul>
    <img src="logo.gif">
  </BODY>
</HTML>
```

Von HTML zu XML: eXtensible Markup Language

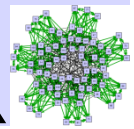
HTML:

```
D. B&uuml;hler and W. K&uuml;chlin: <br>
<i>“Remote Fieldbus ...“</i>, ISIE 2000.
```

XML:

```
<publication>
  <proceedings>
    <author>D. Bühler</author>
    <author>W. Küchlin</author>
    <title>Remote Fieldbus ...</title>
    <conference year="2000">ISIE</conference>
  </proceedings>
</publication>
```

D. Bühler and W. Küchlin: “Remote Fieldbus ...”, ISIE 2000.



XML – eXtensible Markup Language

- **Beschreibung strukturierter Daten** (kein Layout)
- Maschinen- und Menschen-lesbar (ASCII-Text)
- wohlgeformte XML-**Dokumente**
  - paarweise öffnende und schließende Tags
    - HTML erlaubt auch Ausnahmen:
  - keine Überlappung der Tags erlaubt
  - alle Tags kleingeschrieben
  - Attributwerte in doppelten Anführungszeichen
- Inline-Schreibweise mit Attributen
  - `<autor nachname="Stevens", vorname="Richard"/>`
- gemischte Schreibweise
  - ```
    <autor geschlecht="männlich">
        Stevens, W. Richard
    </autor>
    ```


    ```

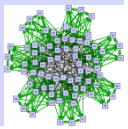
XML – eXtensible Markup Language

- Von W3C standardisiert (www.w3.org)
- Strukturdefinition (Grammatik einer Dokumentfamilie)
  - Document Type Definition (DTD)
  - XML-Schema

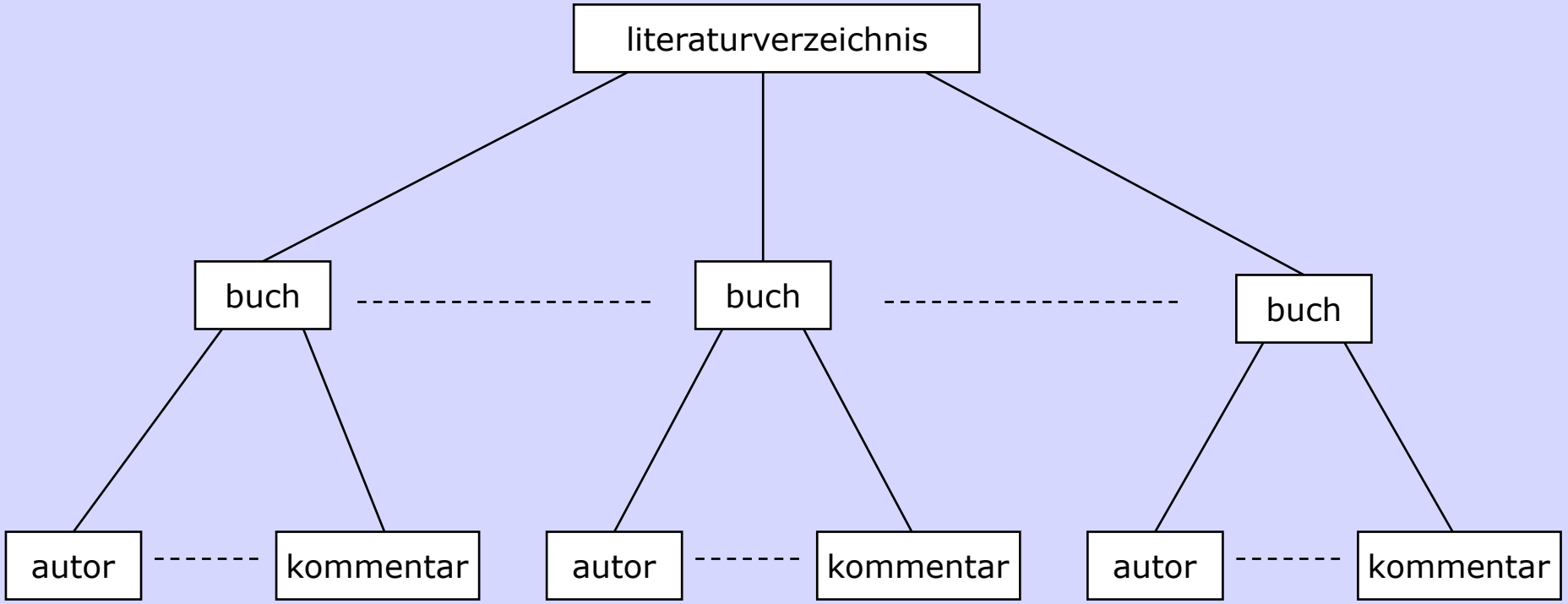
- Vorteile von XML
  - Allgemeinheit
  - Leichte maschinelle Zugänglichkeit der Information
    - Textbasiert
    - Strukturiert
  - Flexibles Layout
  - Spezialisierte Sichten

why XML ?

- open standard (W3C)
- platform independence (non-binary)
- tool support
  - editors, viewers, ...
  - databases (native or relational extenders)
  - parsers / APIs
- universality
- persistence
- web features
- internationalization (i18n)



XML – Beispiel



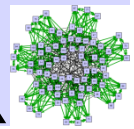
XML – Beispiel

Kopf

```
<?xml version="1.0"?>
```

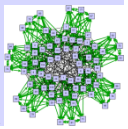
Rumpf

```
<literaturverzeichnis>
        <buch>
                <autor>Stevens, W. Richard</autor>
                <titel>UNIX Network Programming</titel>
                <verlag>Prentice Hall</verlag>
                <erscheinungsjahr>1990</erscheinungsjahr>
                <isbn>0-13-949876-1</isbn>
                <stichwort>Netzwerk</stichwort>
                <stichwort>Netzwerk-Programmierung</stichwort>
        </buch>
        <buch>
                ....
        </buch>
</literaturverzeichnis>
```



XML – Syntax

- Prolog
  - Beispiel: `<?xml version="1.0"?>`
  - verwendete XML-Version
  - verwendeter Zeichensatz (optional)
  - Bedarf an weiteren Dokumenten (optional)
- Kommentare
  - Beispiel: `<!-- Kommentar -->`



XML – Syntax: Namensräume

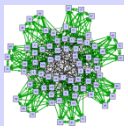
- Ziel: Vermeidung von Mehrdeutigkeiten bei Tags
- Definition mittels `xmlns`-Attribut oder dem `xmlns:`-Präfix
- Wert des Attributs ist Name des Namespaces
  - URI muss auf nichts zeigen

Beispiel 1:

```
<?xml version="1.0"?>
<literaturverzeichnis
            xmlns="http://www.uni-tuebingen.de/diplArb">
    <buch> ... </buch>
</literaturverzeichnis>
```

Beispiel 2:

```
<?xml version="1.0"?>
<da:literaturverzeichnis
            xmlns:da="http://www.uni-tuebingen.de/diplArb">
    <da:buch> ... </da:buch>
</da:literaturverzeichnis>
```



XML Document Type Definition (DTD)

```
<!ELEMENT publication (proceedings | article)>
<!ELEMENT proceedings ((author)*, title,
  conference)>
...
<!ELEMENT conference (#PCDATA)>
<!ATTLIST conference
  year CDATA #required>
```

- meta language
- formal grammar
- known tags, element nesting, attributes, ...
- document classes → document validation

Document Type Definition (DTD)

DTD definiert

- **Elemente**
  - `<!ELEMENT`
  - mit Liste möglicher Kindelemente und deren Kardinalitäten
    - ohne : Pflichtelemente
    - ? : Null- oder einmal
    - * : null bis n-mal
    - + : ein- bis n-mal
- **Attribute**
  - `<!ATTLIST name_des_Elements Attribut1 Typ1 Modifier1`
    `Attribut2 Typ2 Modifier2 >`
    - `<!ATTLIST autor geschlecht PCDATA #REQUIRED>`
- **Entities**
  - Platzhalter für immer wieder verwendete Zeichenketten
    - `<!ENTITY &copyright "Prentice Hall 1990" >`

Document Type Definition (DTD)

- Einbindung
  - im selben Dokument
  - ausgelagert in externes Dokument

```
<?xml version="1.0"?>
<!DOCTYPE literaturverzeichnis SYSTEM "literaturverz.dtd">
<literaturverzeichnis> </literaturverzeichnis>
```

- HTML lässt sich als XML Sprache mit spezieller DTD auffassen
- Maschinelle Validierung möglich (Dokument genügt der DTD)
- Nachteile
  - DTDs sind keine XML-Dokumente
  - neue Syntax
  - nur wenige Standardtypen
  - keine Standardwerte

DTD – Beispiel

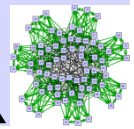
```
<!ELEMENT literaturverzeichnis (buch+)>
   <!ELEMENT buch (autor+, titel, untertitel?, verlag,
       erscheinungsjahr, isbn, stichwort+, abstract?,
       kommentar?)>
       <!ELEMENT autor (#PCDATA)>
       <!ELEMENT titel (#PCDATA)>
       <!ELEMENT untertitel (#PCDATA)>
       <!ELEMENT verlag (#PCDATA)>
       <!ELEMENT erscheinungsjahr (#PCDATA)>
       <!ELEMENT isbn (#PCDATA)>
       <!ELEMENT stichwort (#PCDATA)>
       <!ELEMENT abstract (#PCDATA)>
       <!ELEMENT kommentar (#PCDATA)>
```

PCDATA (Parsed Character Data)
Zeichenketten werden vom XML-Parser interpretiert und Entities aufgelöst.

CDATA (Character Data)
Zeichenketten werden vom Parser eingelesen, aber nicht interpretiert.

XML-Schema

- Basis-Datentypen
  - `string, date, time, int, long, decimal, double, boolean, byte, base64Binary, …`
- komplexere Strukturen
  - Einschränkung des Wertebereichs
  - Listen
  - Vereinigung und Kombination verschiedener Typen
  - Vererbung



Base-64 Codierung

- Codierung von Binärdaten als druckbare ASCII-Zeichenketten
- Prinzip
  - Bilde Blöcke zu je 3 Bytes = 24 Bits
  - Teile jeden Block in 4 Fragmente zu je 6 Bits auf
  - 6 Bits können mit 64 druckbaren ASCII-Zeichen codiert werden
    - 26 Großbuchstaben: A, B, …, Z
    - 26 Kleinbuchstaben: a, b, …, z
    - 10 Ziffern: 0, 1, …, 9
    - 2 Sonderzeichen: + und /
  - Evtl. müssen im Quelltext 1 oder 2 Füllbytes ergänzt und bei der Dekodierung bis zu k ≤ 3 Codes wieder entfernt werden. Dies wird durch k `=´-Zeichen am Ende angezeigt.

XML-Schema – Beispiel

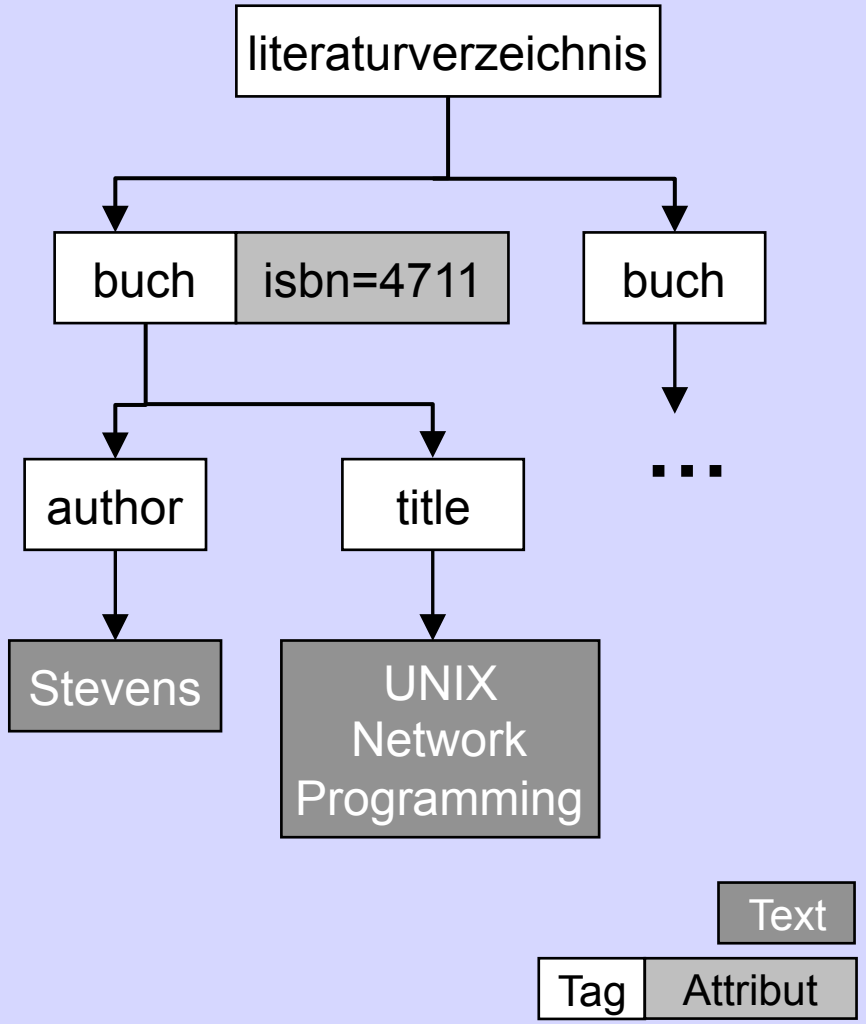
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
    <xs:element name="literaturverzeichnis">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="buch" maxOccurs="unbounded">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="autor" type="xs:string" maxOccurs="unbounded"/>
                        <xs:element name="titel" type="xs:string"/>
                        <xs:element name="untertitel" type="xs:string" minOccurs="0"/>
                        <xs:element name="verlag" type="xs:string"/>
                        <xs:element name="erscheinungsjahr" type="xs:string"/>
                        <xs:element name="isbn" type="xs:string"/>
                        <xs:element name="stichwort" type="xs:string" maxOccurs="unbounded"/>
                        <xs:element name="abstract" type="xs:string" minOccurs="0"/>
                        <xs:element name="kommentar" type="xs:string" minOccurs="0"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:sequence>
    </xs:complexType>
    </xs:element>
</xs:schema>
```

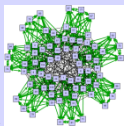
Vom W3C im XML-Schema `http://www.w3.org/2001/XMLSchema` vorgegeben:

- Basistypen `element, complexType, sequence, …`
- Datentypen `string, integer, …`

XML – APIs zur Verarbeitung

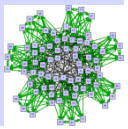
- *Java API for XML Processing (JAXP)*
  - seit J2SE
  - enthält u.a. DOM und SAX-Parser
- DOM
  - *Document Object Model*
  - DOM-Parser erzeugt parse-tree
    - Parser ist *document builder*
- SAX
  - *Simple API for XML*
  - ereignisorientierter Ansatz
  - Dokument wird komplett durchlaufen
  - Beginn / Ende jedes Tags wird über Callback-Methoden mitgeteilt





XML – APIs zur Verarbeitung

- *Java Architecture for XML binding (JAXP)*
  - seit J2EE
  - Automatische Abbildung von XML-Schema Elementen auf Java-Klassen
  - Sehr vorteilhaft, wenn die XML-Schemadefinition komplex ist und sich häufig ändert
    - Jede Schema-Änderung bedingt Code-Änderung
    - Bsp: Neues Sub-Element im Schema bedingt neues Attribut in der zugeordneten Klasse
- Marshalling
  - Java Objekte werden als XML-Dokumente serialisiert
- Unmarshalling
  - XML-Dokumente eines Schemas werden in Java-Objekte de-serialisiert

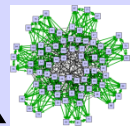


XML Standard Familie

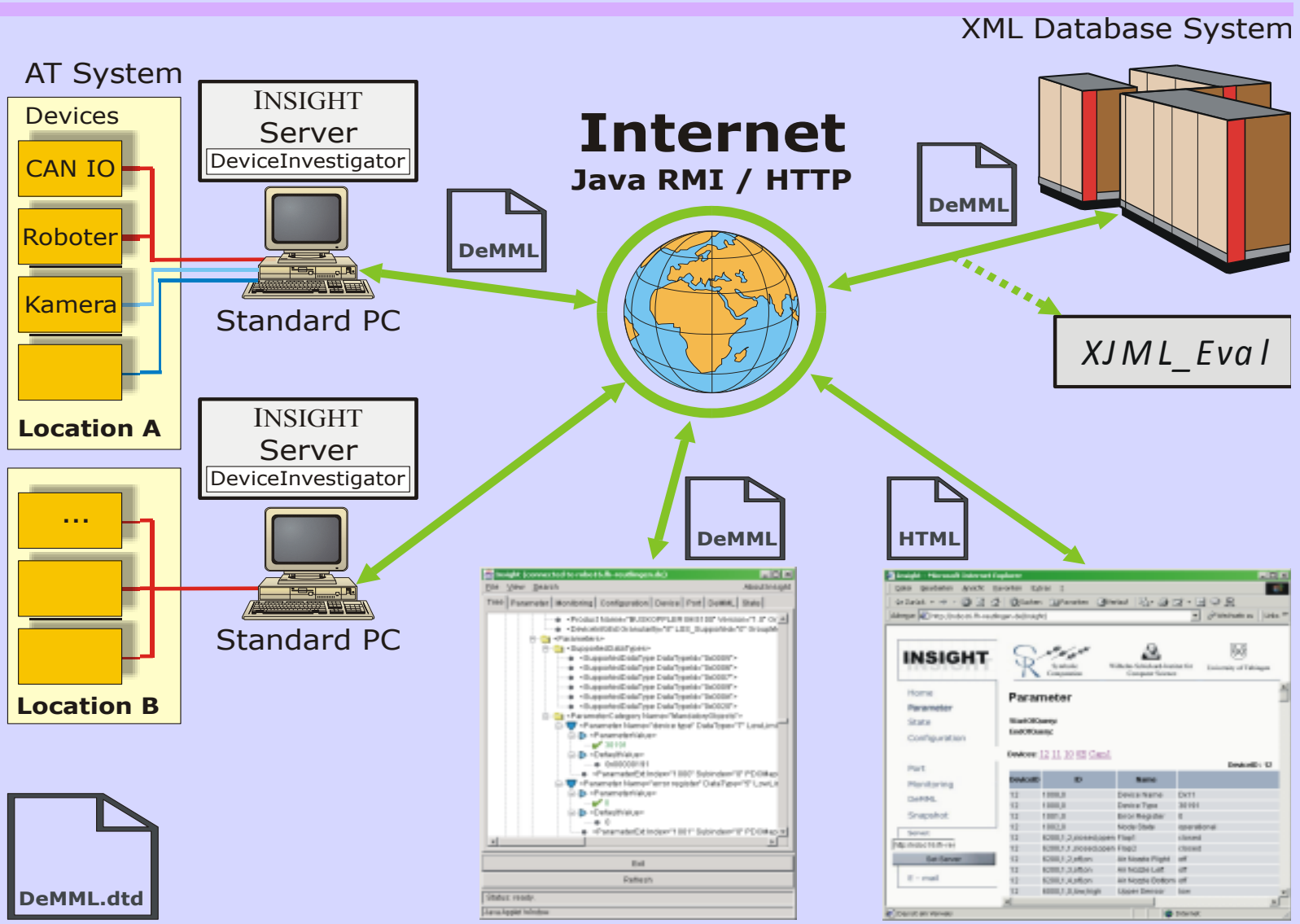
- Von W3C standardisiert (www.w3.org)
- **CSS** (1998): Cascading Style Sheets (layout)
- **XML 1.0**
- **Namespaces** (1999)
- **XSLT 1.0** (1999) XSL Transformations (layout Transformation ohne Profi-Druck))
- **XPath 1.0.** (1999) Zugriff auf Teile eines Dokuments
- **XHTML 1.0** (2000), (Extensible HTML). "A Reformulation of HTML 4 in XML 1.0"
- **DOM Level 2,** Document Object Model, Core Specification (2000)
- **XML Schema** (2001) Grammatik-Sprache für XML-Dokumentfamilien
- **XLink 1.0 (**2001): XML Linking Language (Spez. für Hyperlinks)
- **XML Base** (2001) (Spez. von Datenbank URIs für Dokument-Teile)
- **XSL** 1.0 (2001) Extensible Stylesheet Language (layout)
- **XPointer** (2002) : XML Pointer Language (Spez. von Pfaden in URIs)
- **XQuery 1.0** (2002) XML Query Language (Recherche in XML-Dokumenten)
- **XInclude** (2002) XML Inclusions

XML Zusammenfassung

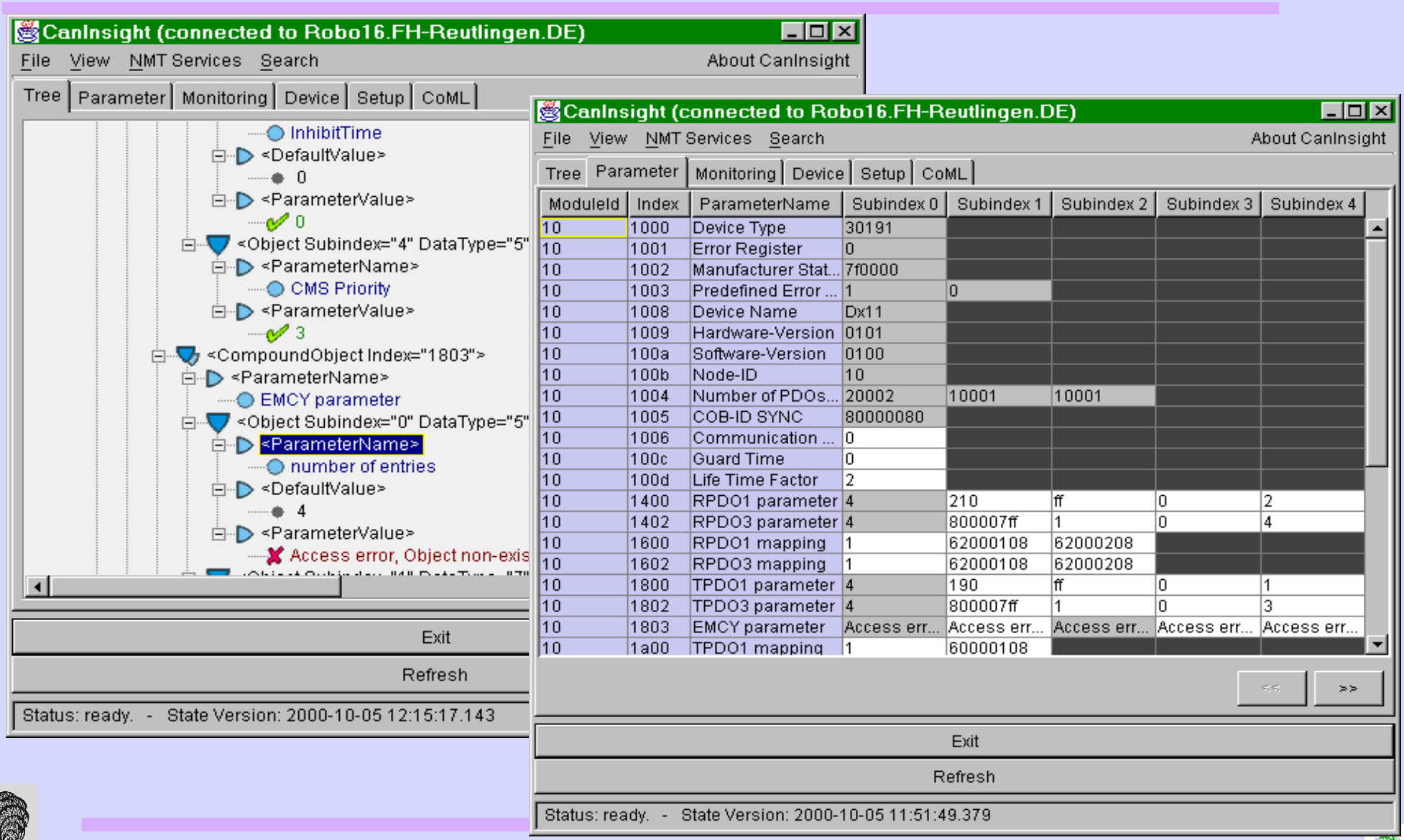
- Umfangreicher Tool Support
  - Java Language bindings: JAXP, JAXB, …
  - XML Datenbanken
  - Schöne, mächtige Editoren (XML Spy)
  - Ein einziges strukturiertes Dokument statt viele einzelne Parameter beim RPC
    - alle Parameter zusammengefasst und visualisierbar
- Generalized Markup
  - Trennung von Struktur und Darstellung
  - CSS, XSL, XSLT ⇒ Darstellung (Layout, Rendering)
  - Sichten ⇒ Flexibilität und Konsistenz
- Document Type Definition (DTD, Schema)
  - Bildung von Dokumentklassen + Validierung



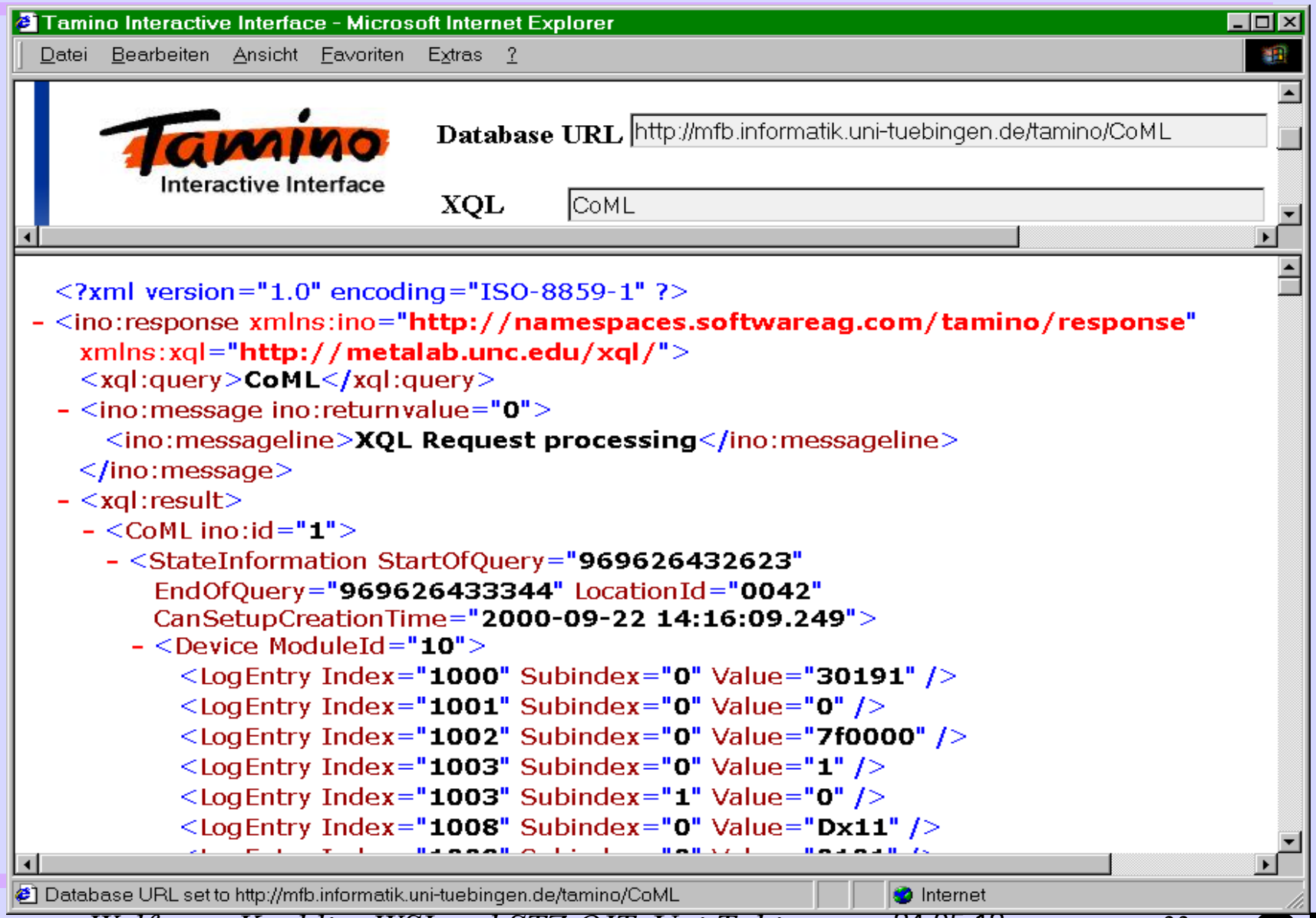
XML Anwendung: Teleservice mit dem Insight System



CanInsight client, XML und HTML Sicht



CoML process data in Tamino XML Datenbank

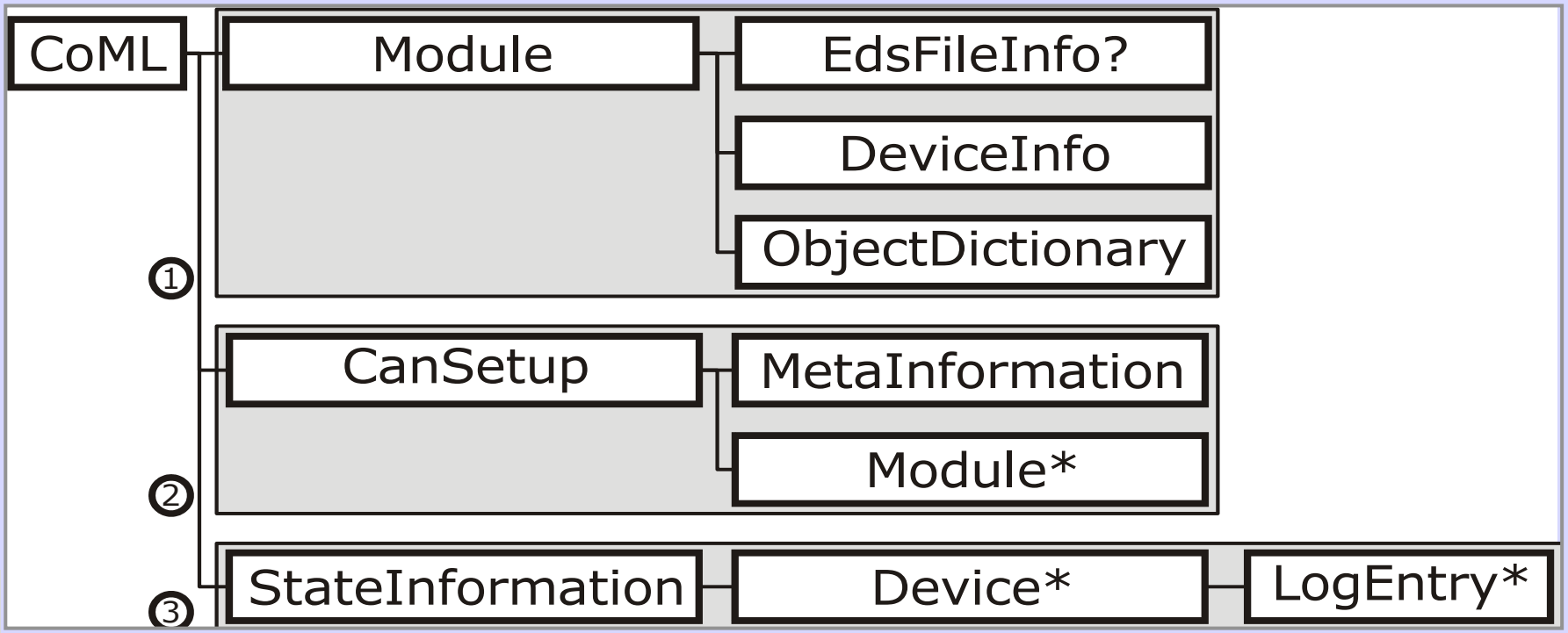


CANopen Markup Language (CoML)

- XML application (CoML DTD)
- Beschreibung von CANopen Geräten in XML
- Allgemeiner: DeMML (Device Management Markup Language)
- Herkömmlich:
  - EDS (Electronic Data Sheet), von CiA für CANopen Geräte definiert.
  - EDS maschinenlesbar aber nur halb formal
- CoML/DeMML: generic data basis for
  - → device profiles
  - → system configuration
  - → process data / state representation
  - →documentation (XSL)
- Vorteil XML:
  - Recherchierbar, überprüfbar, umfassend, integriert (Konfiguration und Prozessdaten in einem Dokument)

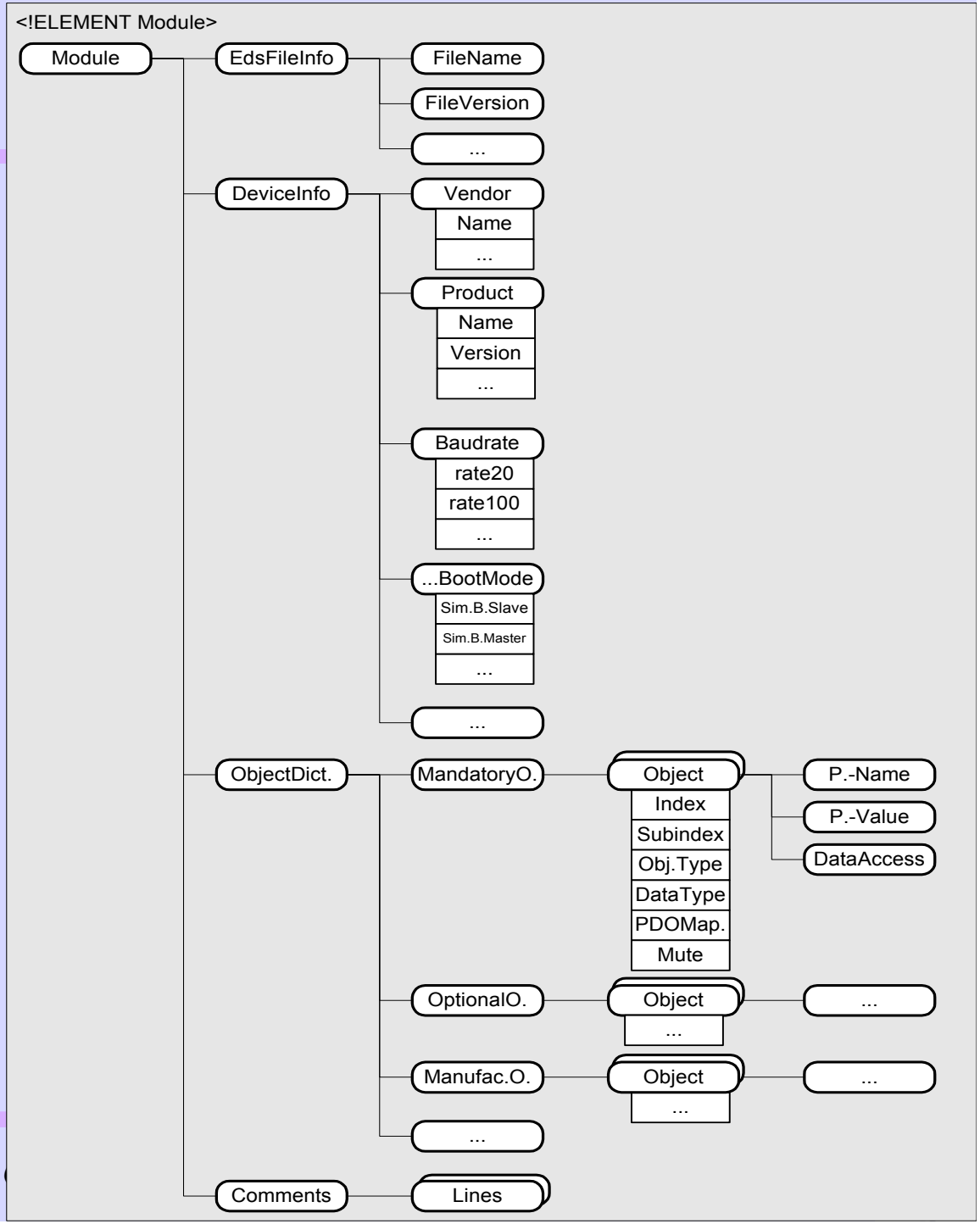
CoML overview

(1) device profiles
(2) systems data
(3) process data



CoML device profiles

- based on EDS
- non-standard EDS information.
- device info.
- parameter
  - description
  - values
  - monitoring configuration



CoML parameter representation

```
<Object
    Index="1000" Subindex="0"
    DataType="7" ObjectType="7"
    AccessType="ro" PDOMapping="0" Mute="0">
    <ParameterName>DeviceType</ParameterName>
    <DefaultValue>0x30191</DefaultValue>
    <ParameterValue>0x30191</ParameterValue>
</Object>
```

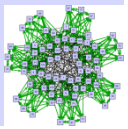
conclusions

- XML/CoML descriptions for
  - CAN device profiles,
  - system data,
  - process data

  make sense.
- tools were implemented for:
  - EDS to CoML translation
  - Completeness and consistency checking of EDS
  - CoML-based system monitoring
  - CoML-based remote system management
  - XML database back-end

CoML / DeMML further readings

- "**The CANopen Markup Language – Representing fieldbus data with XML**" in
  proceedings of the IEEE International Conference on Industrial Electronics, Control and Instrumentation (IECON 2000),
  Oct. 2000, Nagoya, Japan.
- "**Remote fieldbus system management with Java and XML**" in
  proceedings of the IEEE International Symposium on Industrial Electronics (ISIE 2000),
  Dec. 2000, Puebla, Mexico.



HTTP – Hypertext Transfer Protocol

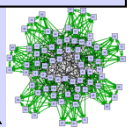
- Aufgabe: Transport von Daten aller Art
- Benötigt nur *reliable transport* (z.B. TCP/IP)
- Kennzeichnung der Datenart mit MIME
  - MIME = Multipurpose Internet Mail Extension
  - Beispiele:
    - `text/html`
    - `text/plain`
    - `image/jpg`
    - `image/gif`
    - `application/msword`

HTTP – Hypertext Transfer Protocol

- Request/response standard of a client and a server.
  - A client is the end-user, the server is the web site. The client making an HTTP request is referred to as the ***user agent***. The responding server—which stores or creates ***resources*** such as HTML files and images—is called the ***origin server***.
- An HTTP client initiates a request.
  - It establishes a reliable connection to a port on a host (TCP and port 80 by default), and sends an HTTP ***request message***.
- An HTTP Server answers the request.
  - An HTTP server listening on that port waits for a client to send a request message. Upon receiving the request, the server sends back a status line, such as "HTTP/1.1 200 OK", and a ***response message***, the body of which is perhaps the requested resource, an error message, or some other information.
- Uniform Resource Identifier (URI)
  - Resources to be accessed by HTTP are identified using Uniform Resource Identifiers (URIs)—or, more specifically, Uniform Resource Locators (URLs)—using the http: or https: URI-schemes.

HTTP – Request and Response Messages

- **GET** Requests a representation of the specified resource. GET should not be used for operations that cause side-effects, such as using it for taking actions in web applications.
- **POST** Submits data to be processed (e.g., from an HTML form) to the identified resource. The data are included in the body of the request. This may result in the creation of a new resource or the updates of existing resources or both.
- **HEAD** Asks for the response identical to the one that would correspond to a GET request, but without the response body. This is useful for retrieving meta-information written in response headers, without having to transport the entire content.
- **PUT** Uploads a representation of the specified resource.
- **DELETE** Deletes the specified resource.
- **TRACE** Echoes back the received request, so that a client can see what intermediate servers are adding or changing in the request.
- **OPTIONS** Returns the HTTP methods that the server supports for specified URL. This can be used to check the functionality of a web server by requesting '*' instead of a specific resource.
- **CONNECT** Converts the request connection to a transparent TCP/IP tunnel, usually to facilitate SSL-encrypted communication (HTTPS) through an unencrypted HTTP proxy.



HTTP – URI

- Uniform Resource Identifier (URI)
  - Identifizierung einer abstrakten oder physischen Ressource

  <*Schema*>:<*Schema-spezifischer Teil*>
  - meist:

  <*Schema*>://[<*Benutzer*>[:<*Passwort*>]@]<*Server*>[:<*Port*>]/[<*Pfad*>][?<*Anfrage*>][#<*Fragment*>]
  - zwei Arten von URIs: URL und URN
- Uniform Resource Locator (URL)
  - (eindeutiger) Name für Ressourcen auf Web Server
  - mit gewöhnlichem Zugriffsmechanismus
- Uniform Resource Name (URN)
  - eindeutiger Name für eine Ressource im `urn` Schema
  - unabhängig vom Ort der Ressource
  - Beispiel: `urn:ietf:rfc:2141`

HTTP – URI Schemata

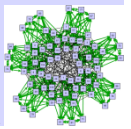
- **http** – Hypertext Transfer Protocol
- **ftp** – File Transfer Protocol
- **mailto** – E-Mail-Adresse
- **sip** – SIP-gestützter Sitzungsaufbau, z.B. für IP-Telefonie
- **urn** – Uniform Resource Names (URNs)
- **tel** – Telefonnummer
- **news** – Newsgroup oder Newsartikel
- **data** – direkt eingebettete Daten
- **fish** – Filetransfer per SSH
- **doi** – Digital Object Identifier

HTTP – URL

➢ URL-Syntax

```
<scheme>:
//[<user>[:<password>]@]<host>[:<port>]/[<path>;][<params>][?
   <query>][#<frag>]
```

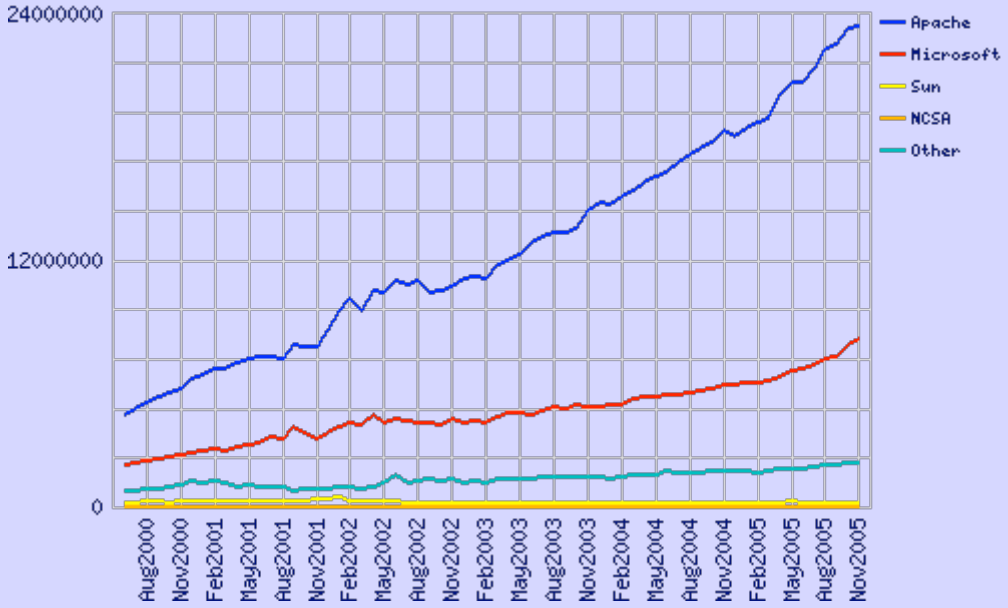
- `scheme`: Zu benutzendes Protokoll
  - z.B. http, https, ftp, file, …
- `host`: Name oder IP des Hosts
- `port`: Default 80 bei HTTP
- `params`: Name-Wert-Paar als Input-Parameter für manche Protokolle
  - z.B. type=animal
- `query`: Parameter für Datenbanken, Suchmaschinen, …
- `frag`: Name für einen Teil der Ressource
  - z.B. Anker in HTML-Dokument: ab wo soll das Dokument gezeigt werden



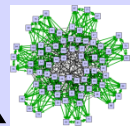
HTTP – Web Server

- Aufgaben
  - Implementierung des HTTP-Protokolls
  - TCP Connection Handling
  - Verwaltung der Ressourcen
  - Administration

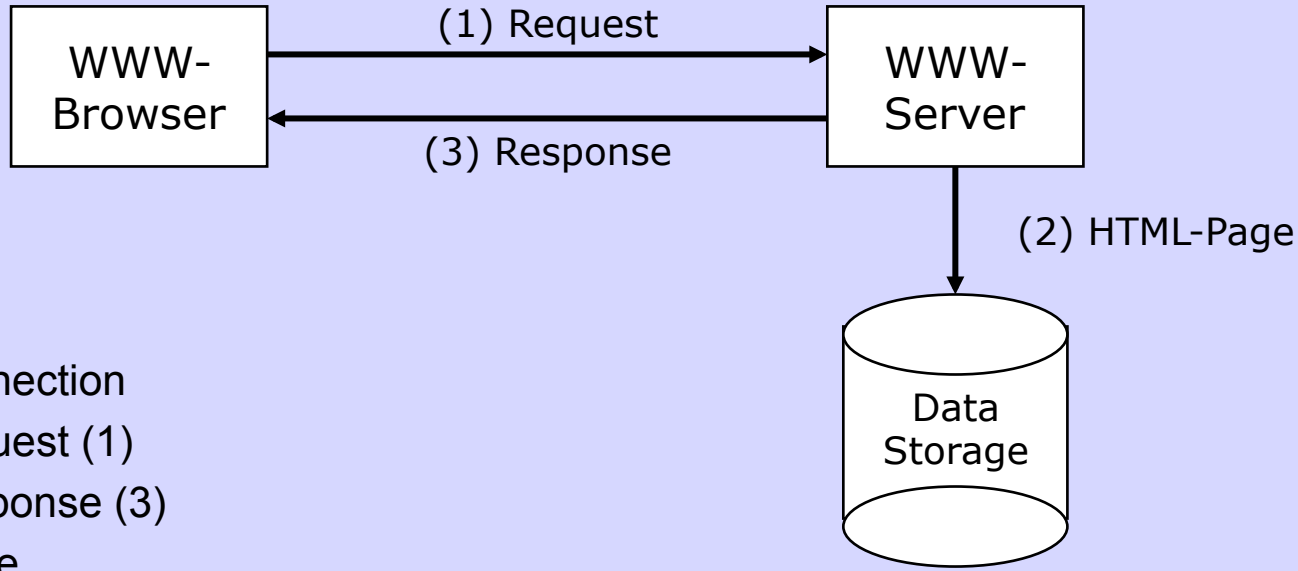
- Marktsituation
  - Totals for Active Servers Across All Domains June 2000 - November 2005



Quelle: http://news.netcraft.com/archives/2005/11/07/november_2005_web_server_survey.html



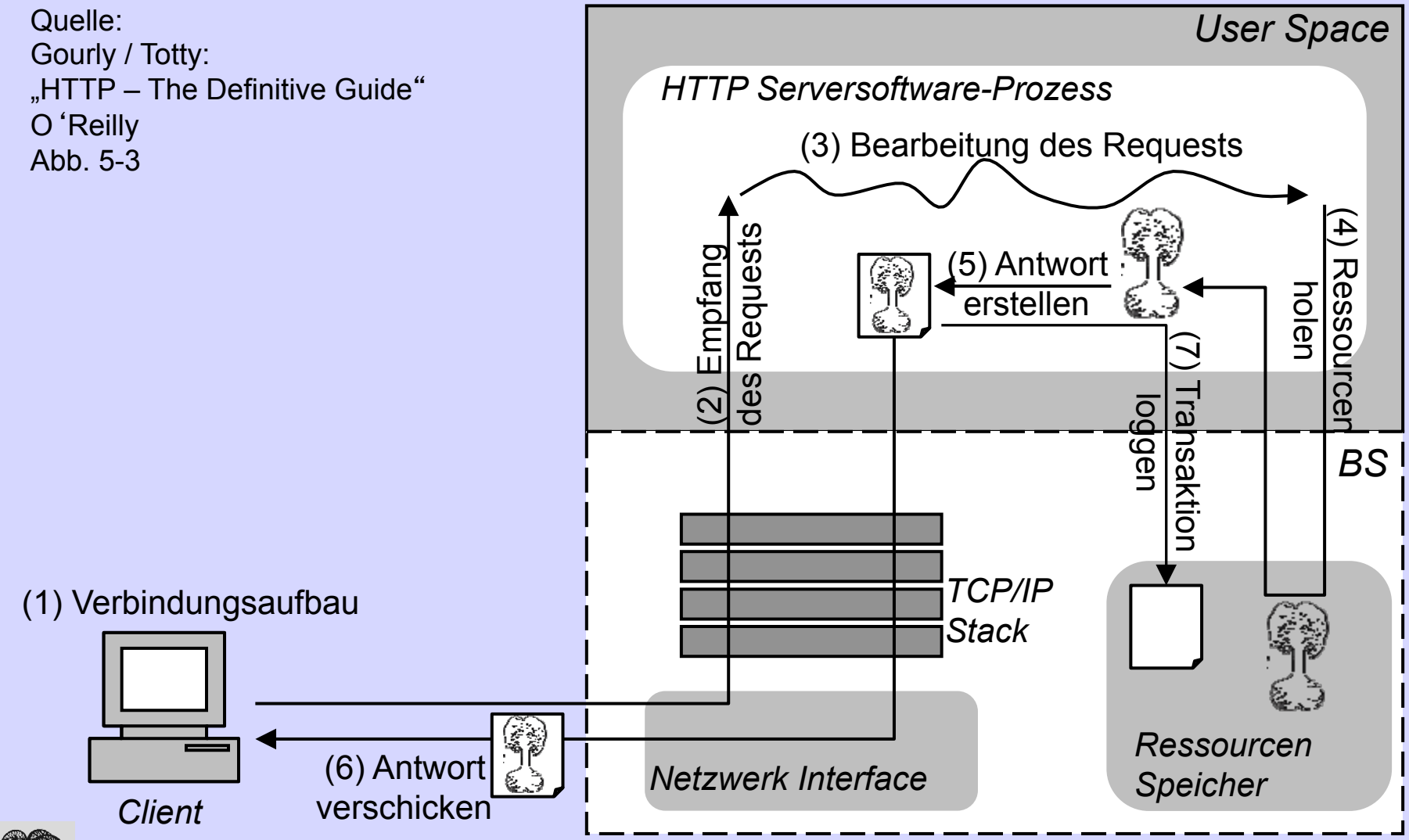
HTTP – Ablauf



- Connection
- Request (1)
- Response (3)
- Close

HTTP – Überblick

Quelle:
Gourly / Totty:
„HTTP – The Definitive Guide“
O ‘Reilly
Abb. 5-3



HTTP – Connection

Verbindungsaufbau zum Server

```
http://www-sr.informatik.uni-tuebingen.de:80/pages/index.html
```

Protokoll | Host | Port | Pfad

(1) Browser extrahiert Host `www-sr.informatik.uni-tuebingen.de`

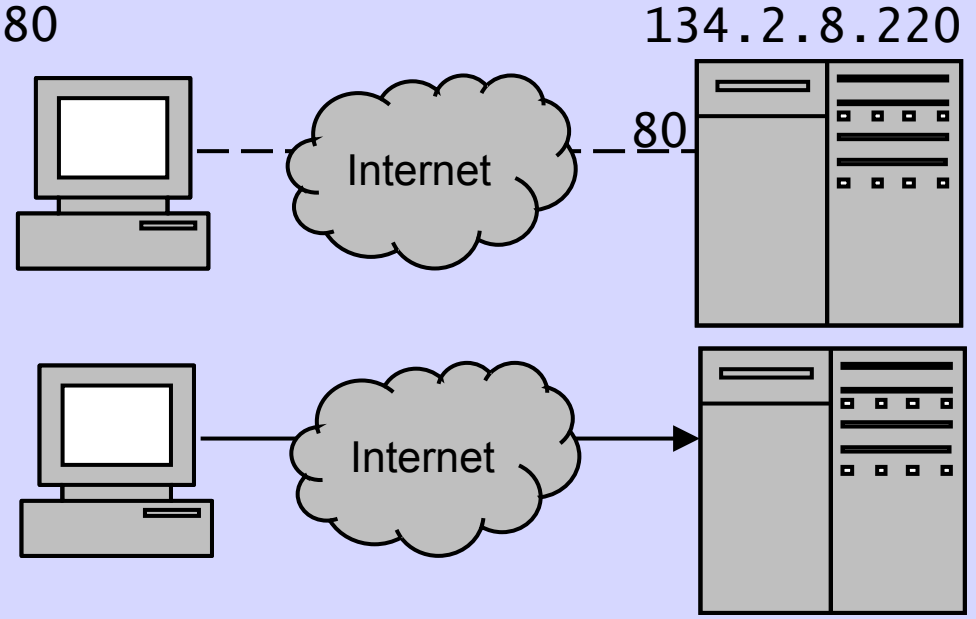
(2) Browser ermittelt IP (DNS) `134.2.8.220`

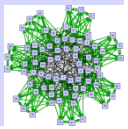
(3) Browser ermittelt Portnummer `80`

(4) Browser baut TCP Verbindung zu 134.2.8.220 Port 80 auf

(5) Browser schickt HTTP Request Nachricht zum Server

(6) …

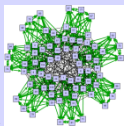




HTTP – Nachrichtenformat

- Format einer HTTP-Nachricht
  - zwischen Header und Body immer eine Leerzeile
- Start Line
  - Client → Server: Request Line
    - gibt an, was der Server tun soll
  - Server → Client: Response Line
    - gibt an, was der Server getan hat
- Headers
  - Zusatzinformationen
  - Name/Wert-Paare
- Body
  - optional
  - enthält „Nutzlast“


```
generic-message = start-line
    *(message-header CRLF
    CRLF
[ message-body ]
```



HTTP – Request

- Format einer Request Nachricht

```
<method> <request-URI> <version>
<headers>

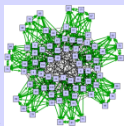
<entity-body>
```

  - request-URI: Pfadname, falls TCP-connection zum origin server besteht, andernfalls volle URI mit Origin-Server-Name, falls connection zu einem Proxy

- Methode GET
  - Sende angefragte Ressource vom Server zum Client

```
GET http://www.heise.de/newsticker/meldung/67324 HTTP/1.1
Accept: image/gif, */*
Host: www.heise.de
```

- Schlüsselwort ACCEPT
  - zeigt Server an, welche Daten Client handhaben kann



HTTP – Response

- Format einer Response Nachricht

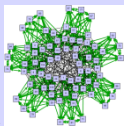
```
<version> <status> <reason phrase>
<headers>

<entity-body>
```

- Beispiel

```
HTTP/1.0 200 OK
Date: Wed, 14 Dec 2005 09:05:35 GMT
Server: Apache/1.3.34
Content-Type: text/html; charset=iso-8859-1
Content-Length: 15456

<html>
...
</html>
```

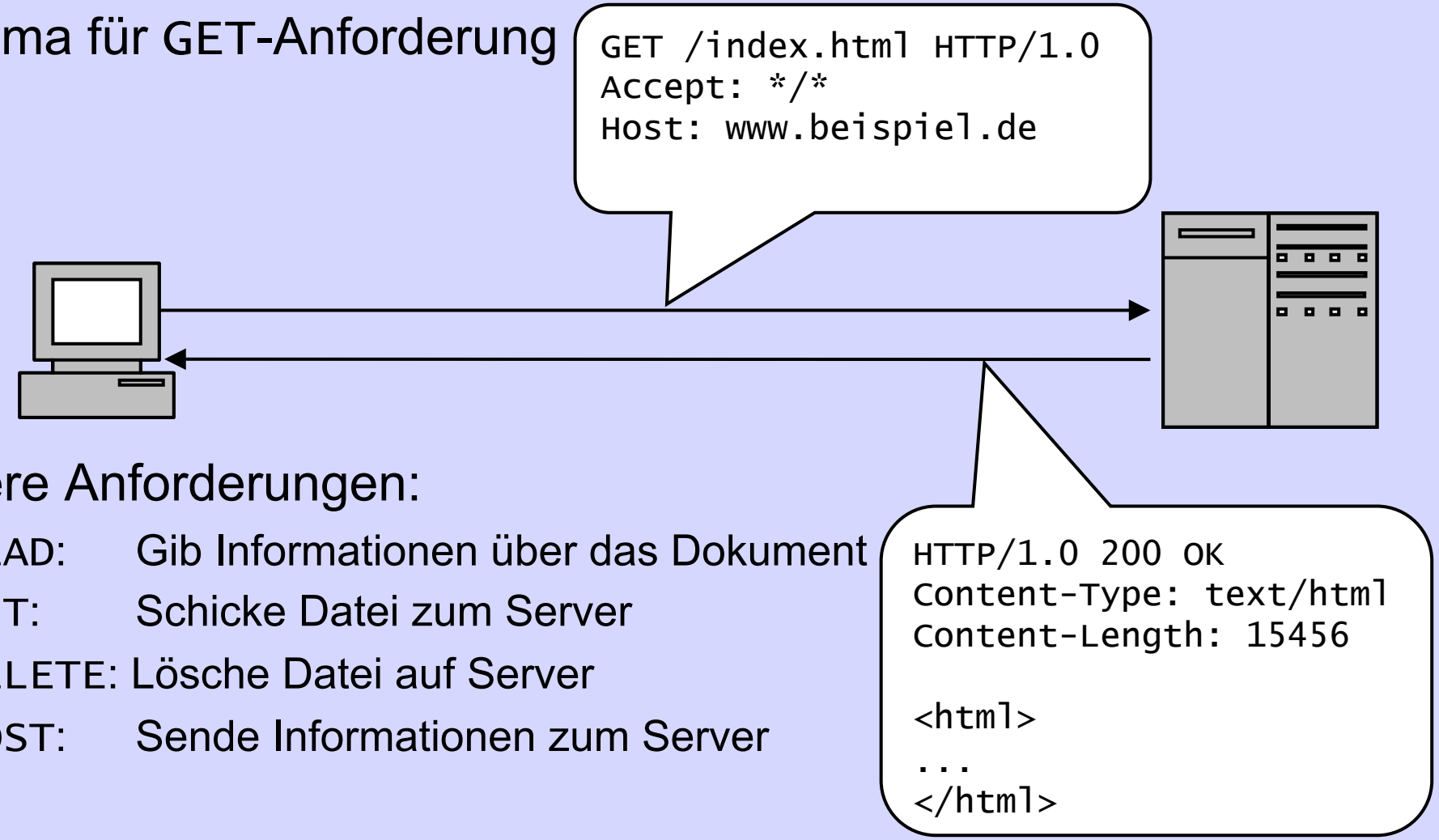


HTTP – Response Antwortcodes

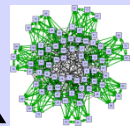
- 1xx: Informationen
- 2xx: Erfolgreiche Operation
  - 200: OK
- 3xx: Umleitung
- 4xx: Client-Fehler
  - 403: Forbidden
  - 404: Not Found
- 5xx: Server-Fehler
  - 504: Gateway Time-out

HTTP – Request

- Schema für GET-Anforderung

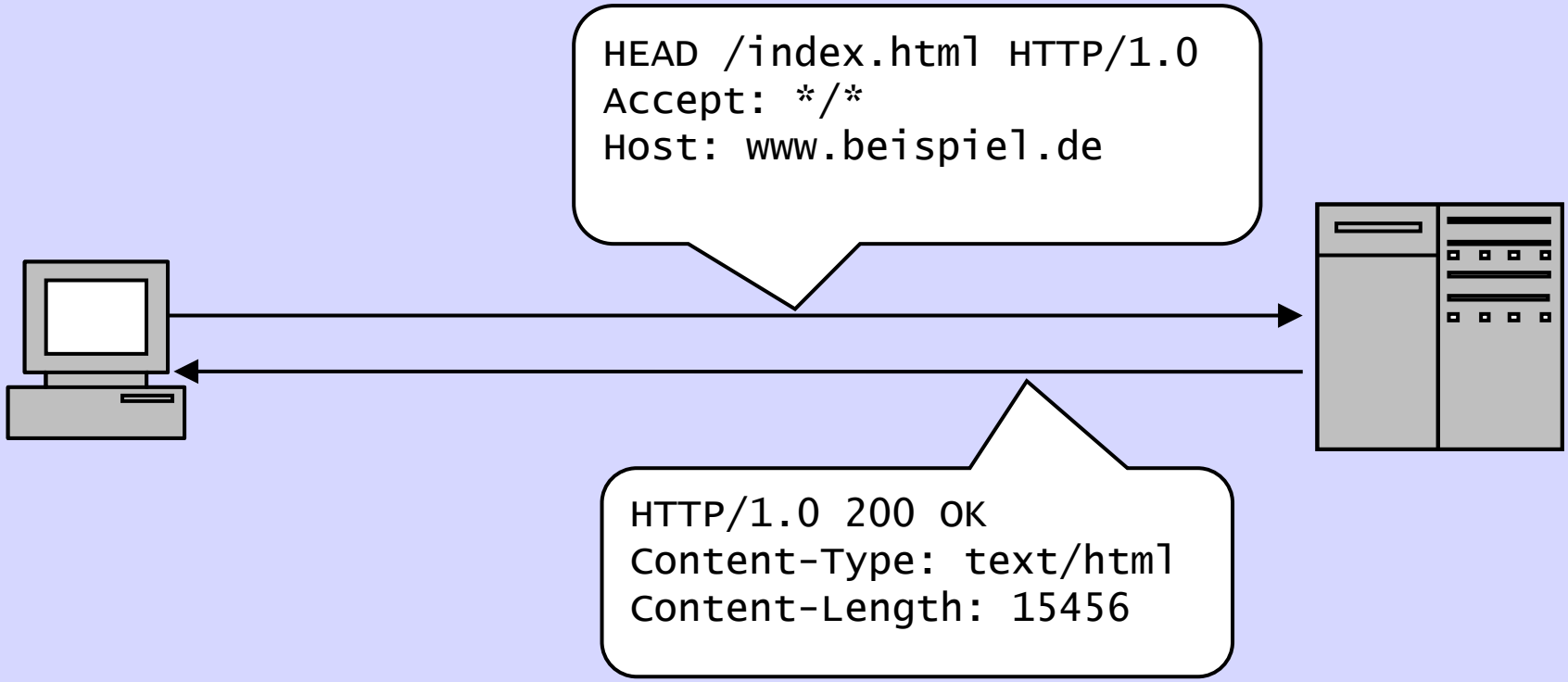


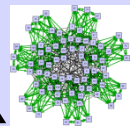
- weitere Anforderungen:
  - HEAD: Gib Informationen über das Dokument
  - PUT: Schicke Datei zum Server
  - DELETE: Lösche Datei auf Server
  - POST: Sende Informationen zum Server
  - …



HTTP – Request HEAD

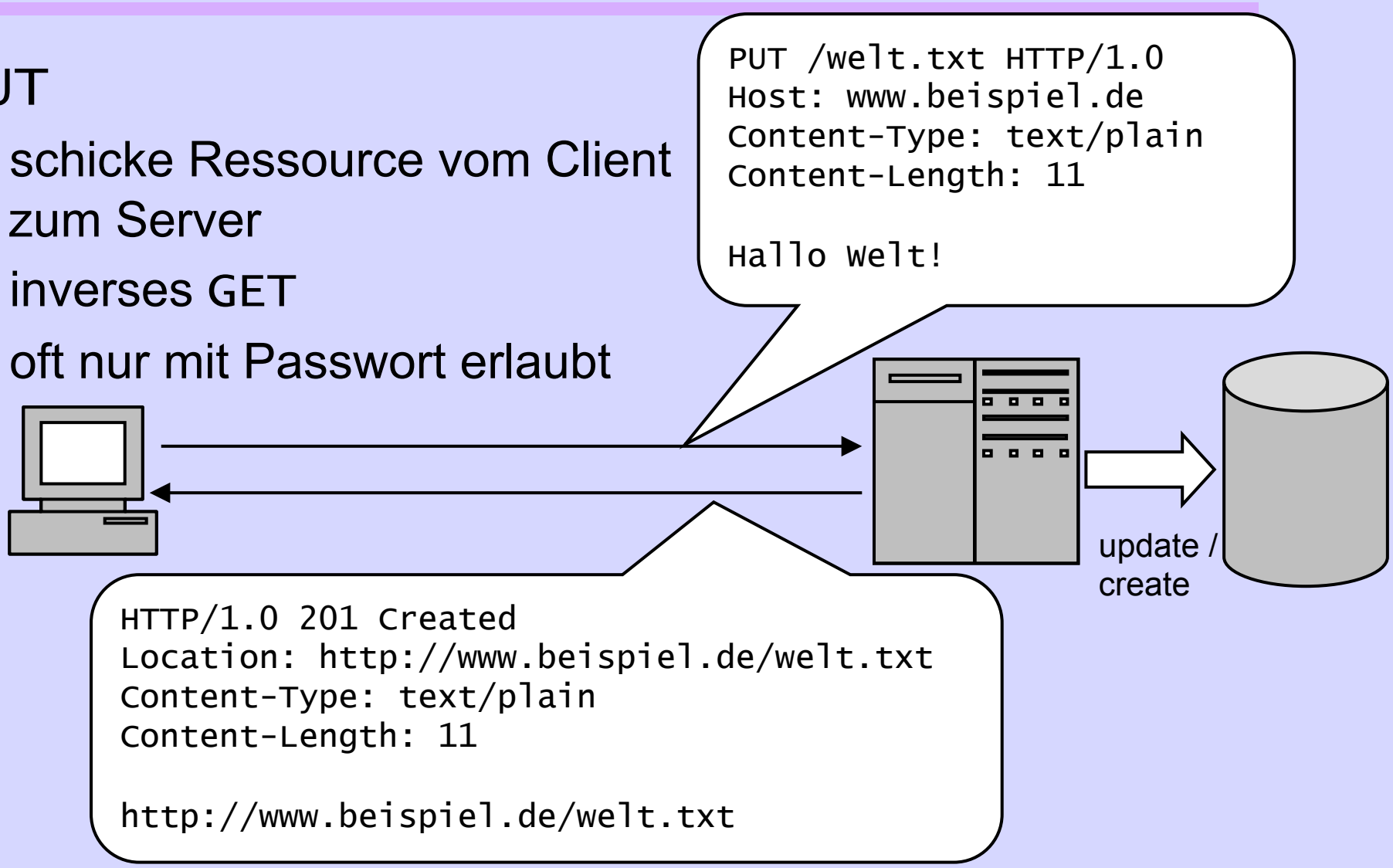
- HEAD
  - Sende Header der angefragten Ressource zum Client
    - wie `GET`, nur ohne Body in der Antwort





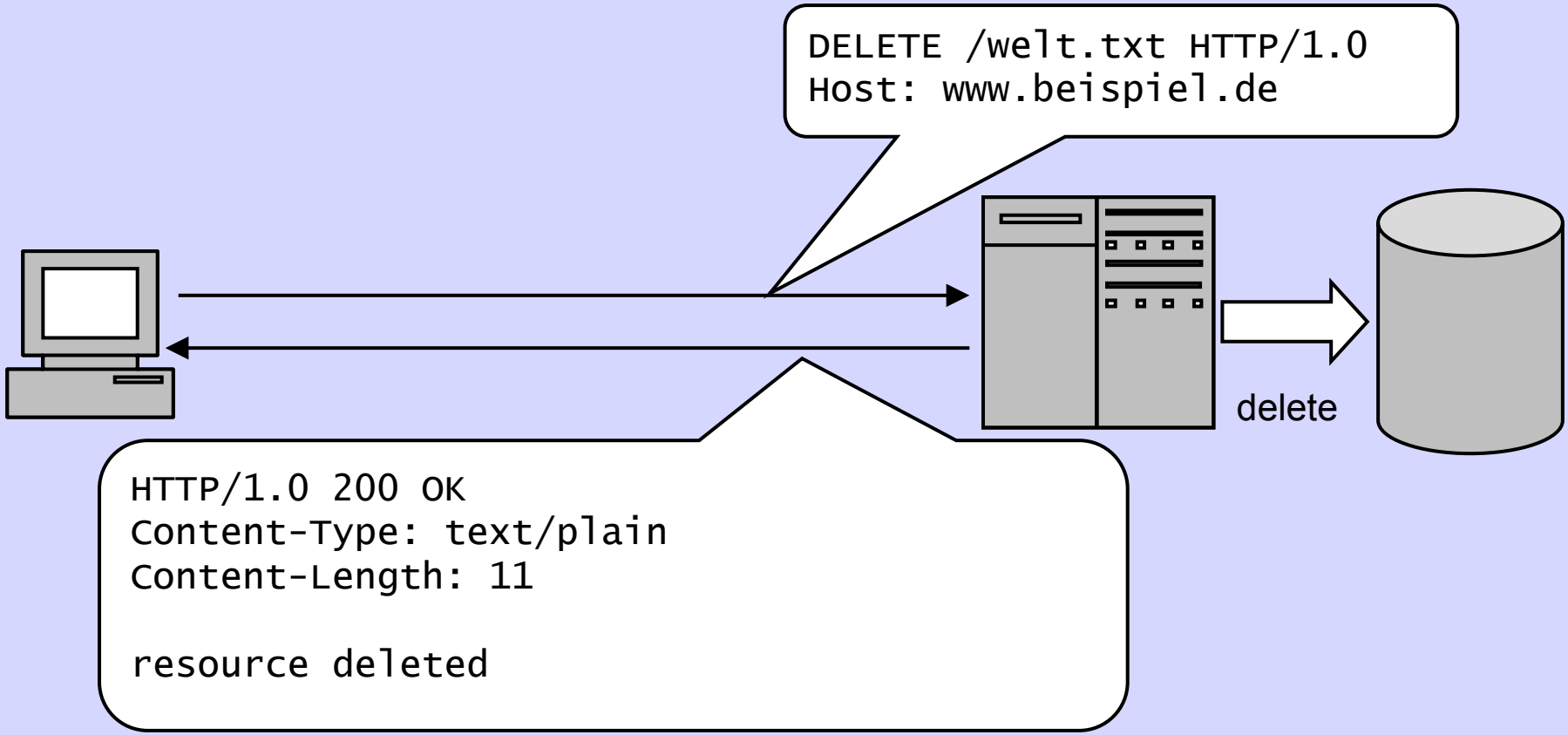
HTTP – Request PUT

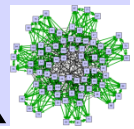
- PUT
  - schicke Ressource vom Client zum Server
  - inverses GET
  - oft nur mit Passwort erlaubt



HTTP – Request DELETE

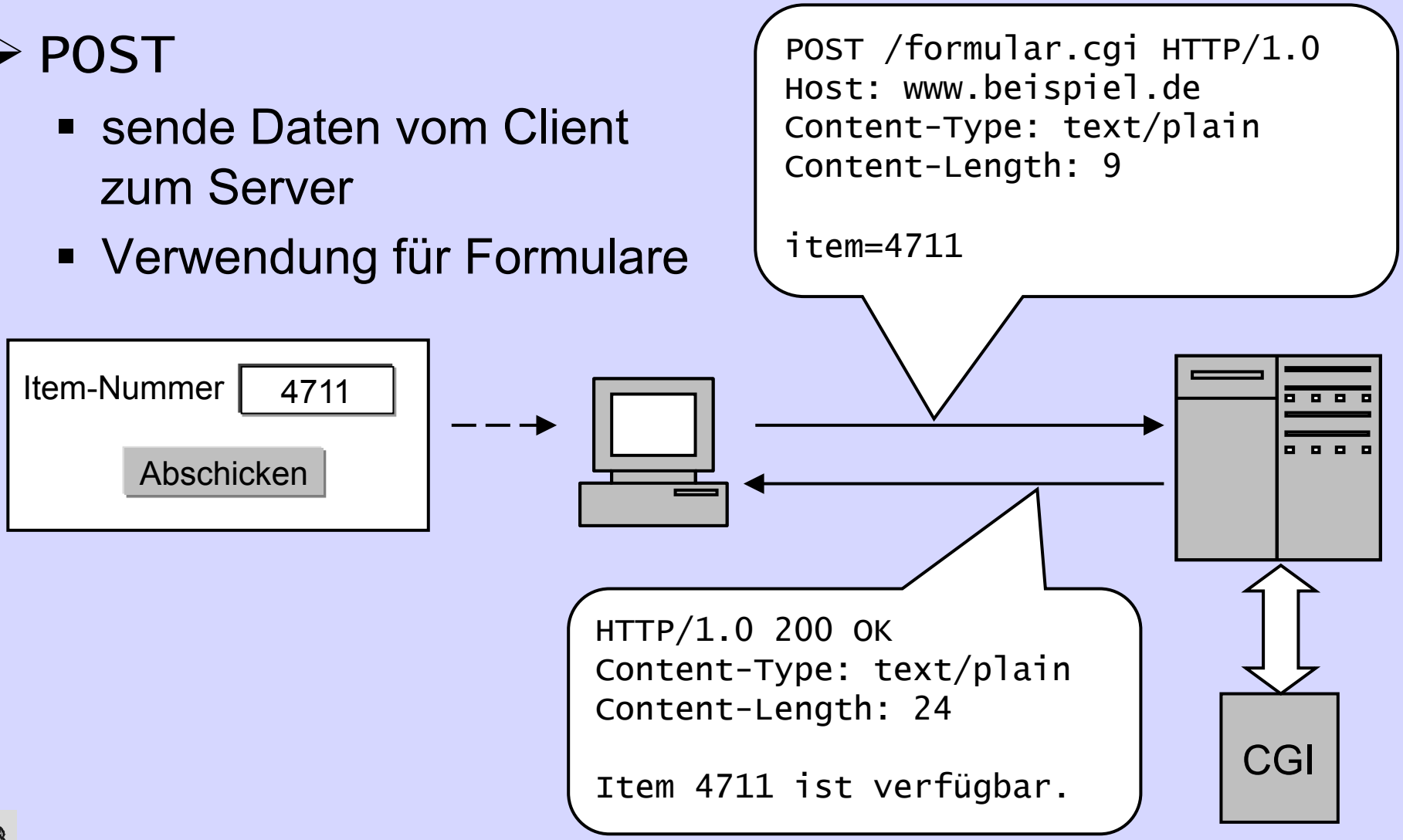
- DELETE
  - lösche Ressource auf Server





HTTP – Request POST

- POST
  - sende Daten vom Client zum Server
  - Verwendung für Formulare



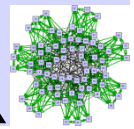
Servlets

- Problem: HTML Dokumente als Files sind statisch
- Idee: Generiere HTML-Dokumente dynamisch durch Programm
- Servlet: Ein Java-Programm, das auf dem Server als Reaktion auf einen http-request gestartet wird
- HTML-Seite wird vom Servlet generiert.

Servlets – Beispiel

```
public class HelloWorld extends HttpServlet {

   public void service(HttpServletRequest request, HttpServletResponse response)
                                                 throws IOException, ServletException {
      // set the contentType
      response.setContentType("text/html;charset=ISO-8859-1");
      // get the PrintWriter
      PrintWriter out = response.getWriter();
      // create content
      StringBuffer content = new StringBuffer();
      content.append("<!DOCTYPE html PUBLIC \"-//w3c//dtd html 4.0  transitional//en
   \" \"http://www.w3.org/TR/REC-html40/strict.dtd\">");
      content.append("<HTML><HEAD><TITLE>JSP-Seite</TITLE></HEAD>");
      content.append("<BODY>");
      // dynamic content: current Date
      content.append("Heute ist der "+new Date());
      content.append("</BODY></HTML>");
      // write out
      out.print (content.toString());
  }
}
```



Java Server Pages (JSP)

- JSP: Eine HTML-Seite mit eingebettetem Java-Code, der auf Server ausgeführt wird und Teile der Seite dynamisch generiert.
- Problem: Es ist umständlich, die statischen Teile der HTML-Seite durch das Servlet in print-statements zu generieren.
- Idee: Schreibe statische Teile in Dokument und bette dynamische Teile darin ein durch Programmcode (spezieller HTML-Kommentar).
- JSP wird von Webserver in Servlet übersetzt

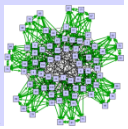
JSPs – Java Server Pages

```
<html>
<body>

<%-- Kommentar --%>
<% //Java-Code %>

</body>
</html>
```

- **Skriptlet <% ... %>**
- **Kommentar <% --...--%>**
- Direktive <% @ ... %>
- Ausdruck <% = ... %>
- Deklaration <%! ...%>



JSPs – Java Server Pages

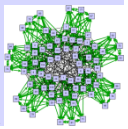
```
<html>
<body>

<%@ page import="java.util.*"%>
<p>Heute ist der
<%= new Date() %>

</body>
</html>
```

Heute ist der
Tue Nov 29 14:48:19 CET 2005

- Skriptlet <% ... %>
- Kommentar <% --...--%>
- **Direktive <% @ ... %>**
- **Ausdruck <% = ... %>**
- Deklaration <%! ...%>



JSPs – Java Server Pages

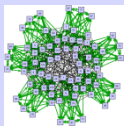
```
<html>
<body>

<%! int x1 = -3, x2 = 4; %>
<%! public int add(int x,int y)
       {return x + y;} %>
<p>Die Summe von
<%= x1%> und
<%= x2%> ist
<%= add(x1,x2) %>
</p>

</body>
</html>
```

Die Summe von -3 und 4 ist 1

- Skriptlet <% ... %>
- Kommentar <% --...--%>
- Direktive <% @ ... %>
- Ausdruck <% = ... %>
- **Deklaration <%! ...%>**



JSPs – Java Server Pages

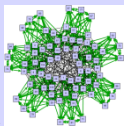
```
<html>
<body>

<%! int x1 = -3 %>
<p>Der Betrag von
<%= x1%> ist
<% if (x1 >= 0){%>
<%= x1%>
<% }else {%>
<%= -x1 %>
<%}%> </p>

</body>
</html>
```

Der Betrag von -3 ist 3

- Skriptlet <% ... %>
- Kommentar <% --...--%>
- Direktive <% @ ... %>
- Ausdruck <% = ... %>
- **Deklaration <%! ...%>**

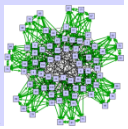


JSPs und Java Beans

- Idee: JSP Code in Komponenten kapseln
  - über Komponenten-Schnittstelle systematisch auf Code zugreifen
  - Einsatz von Java Beans, Komponentenmodell von Java
- Komponente:
  - größerer Software-Modul mit
    - Geheimnis
    - kleiner systematisch aufgebauter Schnittstelle nach außen
- Vorteile von Java Beans gegenüber Variablen
  - Wiederverwendung von Software-Komponenten
  - Verständlichkeit
    - der JSPs auch für Java-Unkundige
  - Kapselung
  - Konzentration zusammengehöriger Daten

Java Beans

- Java Komponenten-Modell
  - Klassen mit methodisch konstruierter Schnittstelle
  - extern konfigurierbare Container
    - speichern Variablen
    - kapseln Logik der Variablen
- (Minimal-) Anforderungen an Java Beans
  - leerer Konstruktor
  - keine öffentlichen Objektvariablen
  - für alle Objektvariablen Getter und Setter-Methoden
    - getX() / setX(…) für Attribut x.
    - isX () bei booleschen Variablen



JSPs und Java Beans – Beispiel Bean

```
package beans;

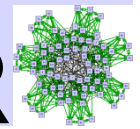
public class RouteBean {

   /** Start der Reise */
    private String start;

    public String getStart () { return start; }
    public void setStart (String start) { this.start = start; }

    /** Das Ziel der Reise */
    private String destination;

    public String getDestination () { return destination; }
    public void setDestination (String dest) {
        this.destination = dest;
   }
}
```



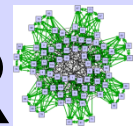
JSPs und Java Beans – Beispiel JSP

```
<!DOCTYPE html PUBLIC "-//w3c//dtd html 4.0 transitional//en"
                      "http://www.w3.org/TR/REC-html40/strict.dtd">

<jsp:useBean id="route" class="beans.RouteBean" />

<jsp:setProperty name="route" property="start" value="Berlin" />
<jsp:setProperty name="route" property="destination"
                                              value="Paris" />


<HTML>
   <BODY>
       Ihre augenblickliche Reiseroute :
       Start: <jsp:getProperty name="route" property="start" />
  <br>
       Ziel: <jsp:getProperty name="route"
                                          property="destination" />
  </BODY>
</HTML>
```

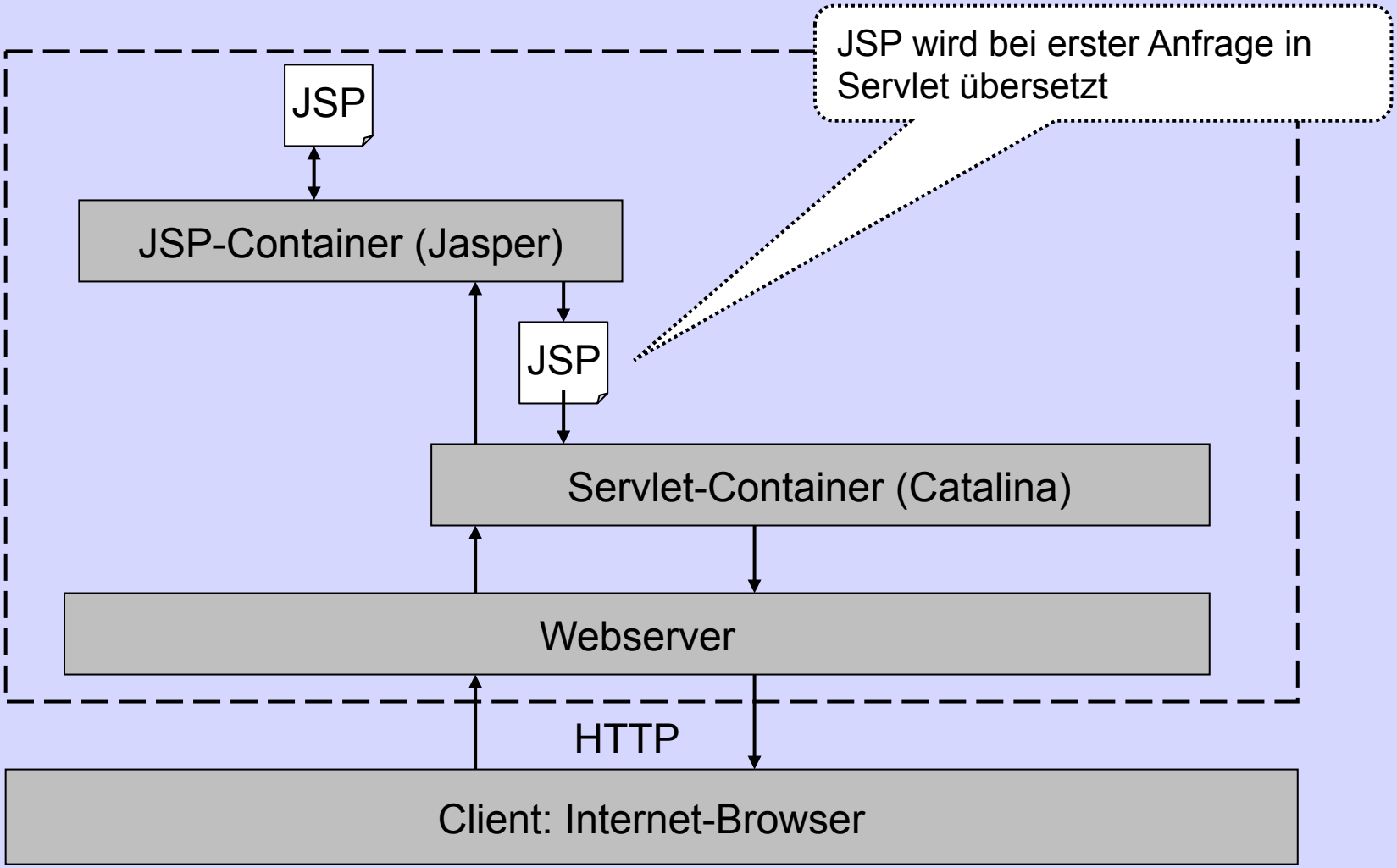


Webserver – Tomcat

Layout für Webanwendungen

- `BspApp\`
  - `css\` (Cascading Stylesheets)
  - `html\` (statische HTML-Dokumente)
  - `img\` (Bilder für die Anwendung)
  - `jsp\` (dynamische JSPs)
  - `WEB-INF\`
    - `classes\` (kompilierte Java-Klassen (Beans und Servlets))
    - `lib\` (benötigte Klassenbibliotheken (.jar-Dateien))
    - `sources\` (optional, Quelldateien (Java Beans))
    - `web.xml` (Web Deployment Descriptor für Setup-Informationen)

Webserver – Tomcat

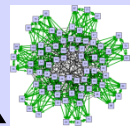


SR

Vergleich von JSPs und Servlets

- Servlets sind komplizierter, erlauben mehr Kontrolle
- JSP sind komfortabler, erlauben weniger Kontrolle

	JSPs	Servlets
Übersetzungszeitpunkt	Zur Laufzeit durch die JSP-Engine	Manuell
Bindung an URL	Unter dem tatsächlichen Pfad, innerhalb der Web-Applikation erreichbar	Wird über Konfigurationsfile (`web.xml`) and eine oder mehrere symbolische URLs gebunden
Services	Beschränkt auf eine einzige Service-Methode	Eigene Service-Methode für jeden Request-Typ (PUT, GET, POST, …) möglich
HTML-Code	Kann direkt eingefügt werden	Muss über `print()` ausgegeben werden

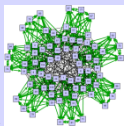


J2EE – Java 2 Enterprise Edition

- offener Standard für verteilte Java-basierte Middleware-Plattform
  - Satz von Spezifikationen und Technologien
  - definiert 1998, EJB 1.1 (1999), EJB 2.0 (2001), EJB 2.1 (2003)
  - für Enterprise-Anwendungen
- Ziel: Schreiben von Applikationsservern in Java
  - **Session Bean**: kapselt Applikationslogik einer Methode
  - **Entity Bean**: kapselt und automatisiert Zugriffslogik auf eine Datenbanktabelle (Abbildung Klasse ⇔ Tabelle)
  - **Message Driven Bean** (EJB 2.0): Reaktion auf asynchr. Nachr.
  - EJB **Container**: Laufzeitumgebung

J2EE – Java 2 Enterprise Edition

- Prinzip
  - J2EE = J2SE + Containertechnologie + Standard Services
  - Komponenten in Container
    - Komponenten: Enterprise Java Beans (EJB)
    - im Gegensatz zu z.B. CORBA mit OO-Paradigma
  - entfernte Methodenaufrufe zwischen verteilten Komponenten
    - Grundlage: Java RMI mit Corba-Protokoll IIOP
- Implementierungen:
  - IBM: Websphere Application Server
  - BEA: WebLogic
  - SUN: J2EE, Java EE 5, Glassfish
  - Open Source: JBoss

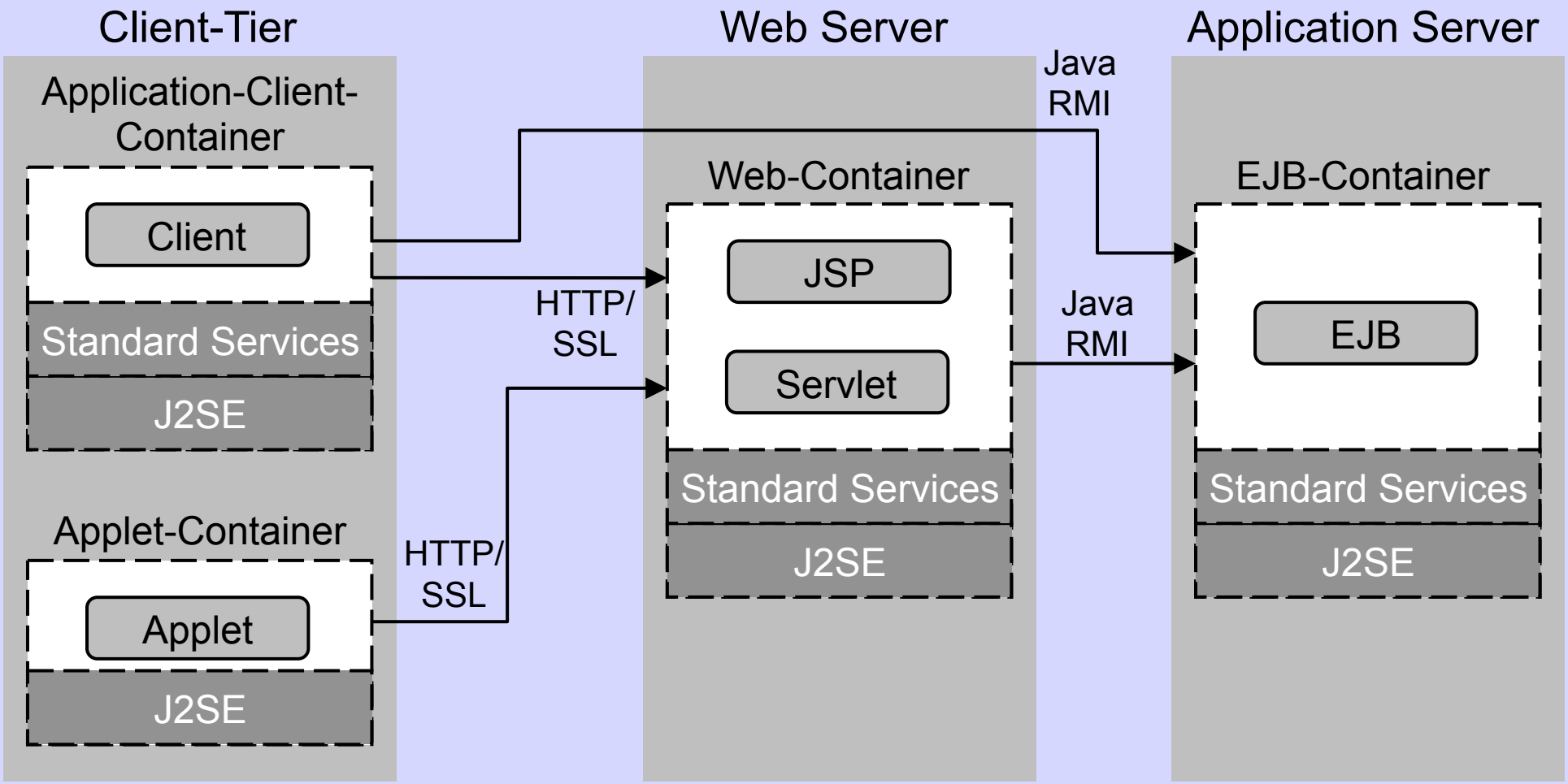


J2EE – Containerarchitektur

- Container
  - Laufzeitumgebung für Komponenten

  → Kommunikation der Komponenten *nur* über Container
- Aufgaben eines EJB-Containers
  - Überwachung des Lebenszyklus von EJBs
  - Instanzen-Pooling
  - Namens- und Verzeichnisdienst
  - Transaktionsdienst
  - Nachrichtendienst
  - Persistenz
- Unterstützung von vier Container-Typen
  - EJB-Container
  - Web-Container mit Servlets
  - Application-Client-Container mit Application Clients
  - Application-Container mit Applets

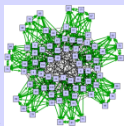
J2EE – Containerarchitektur und EJBs



nach: Hammerschall, Abb.9.2

Standard-Services

- Java Database Connectivity (JDBC)
  - API für den Zugriff auf relationale Datenbanken
- Java Transaction API (JTA)
  - `begin, commit, rollback`
- J2EE Connector Architecture (JCA)
  - Standardarchitektur zur Integration beliebiger Informationssysteme mit einer J2EE-Plattform
  - Beispiel:
    - SAP entwickelt für die Integration Adapter mit JCA
- Java Management Extension (JMX)
  - Architektur zur Verwaltung von Ressourcen
  - Ressourcen: Objekte, Treiber, Dienste, Anwendungen

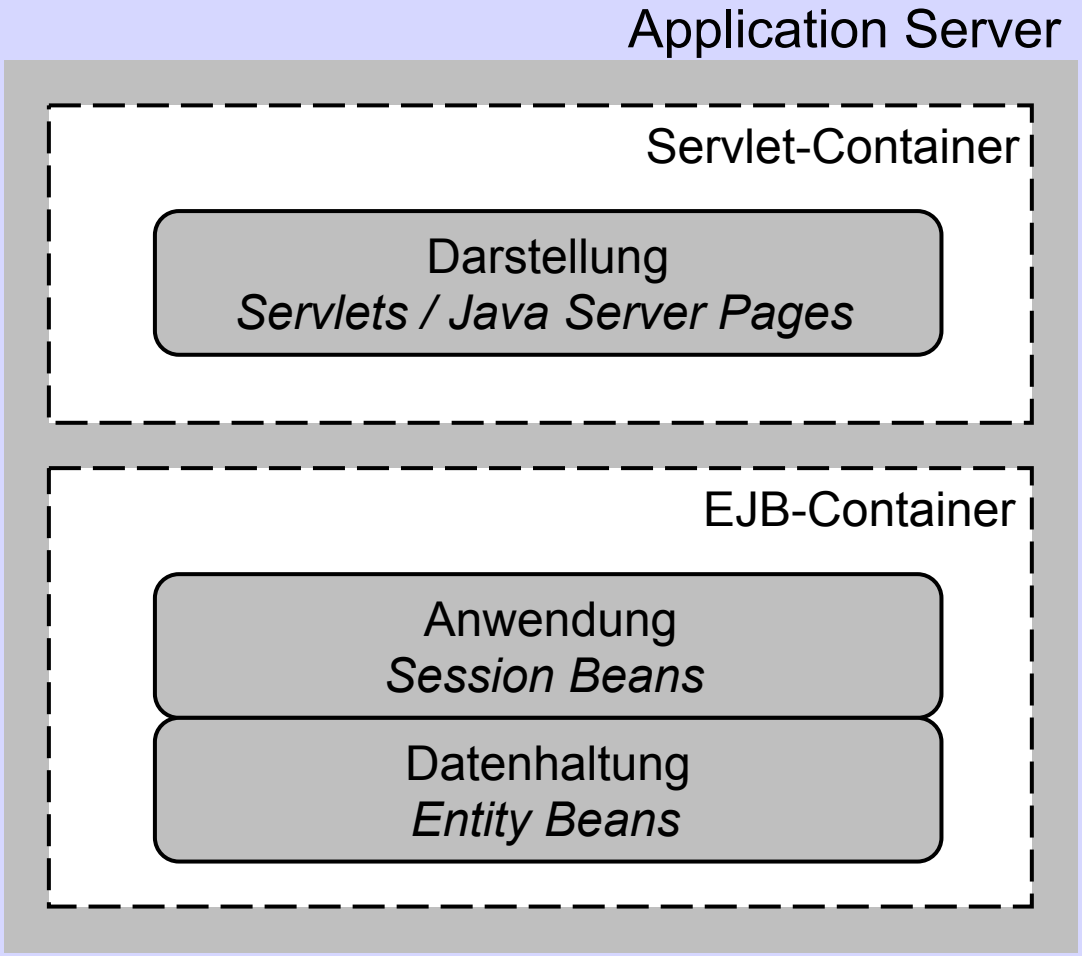


Standard-Services

- Java Message Service (JMS)
  - Versenden von Nachrichten zwischen Komponenten einer J2EE-Anwendung
- Java Mail
  - Versenden von E-Mails
- Java API for XML (JAXP)
  - Bearbeiten von XML-Daten (SAX und DOM)
- Java Authentication and Authorization Service (JAAS)

J2EE – Application Server

- Beispiele:
  - JBoss
  - BEA WebLogic
  - IBM WebSphere AS

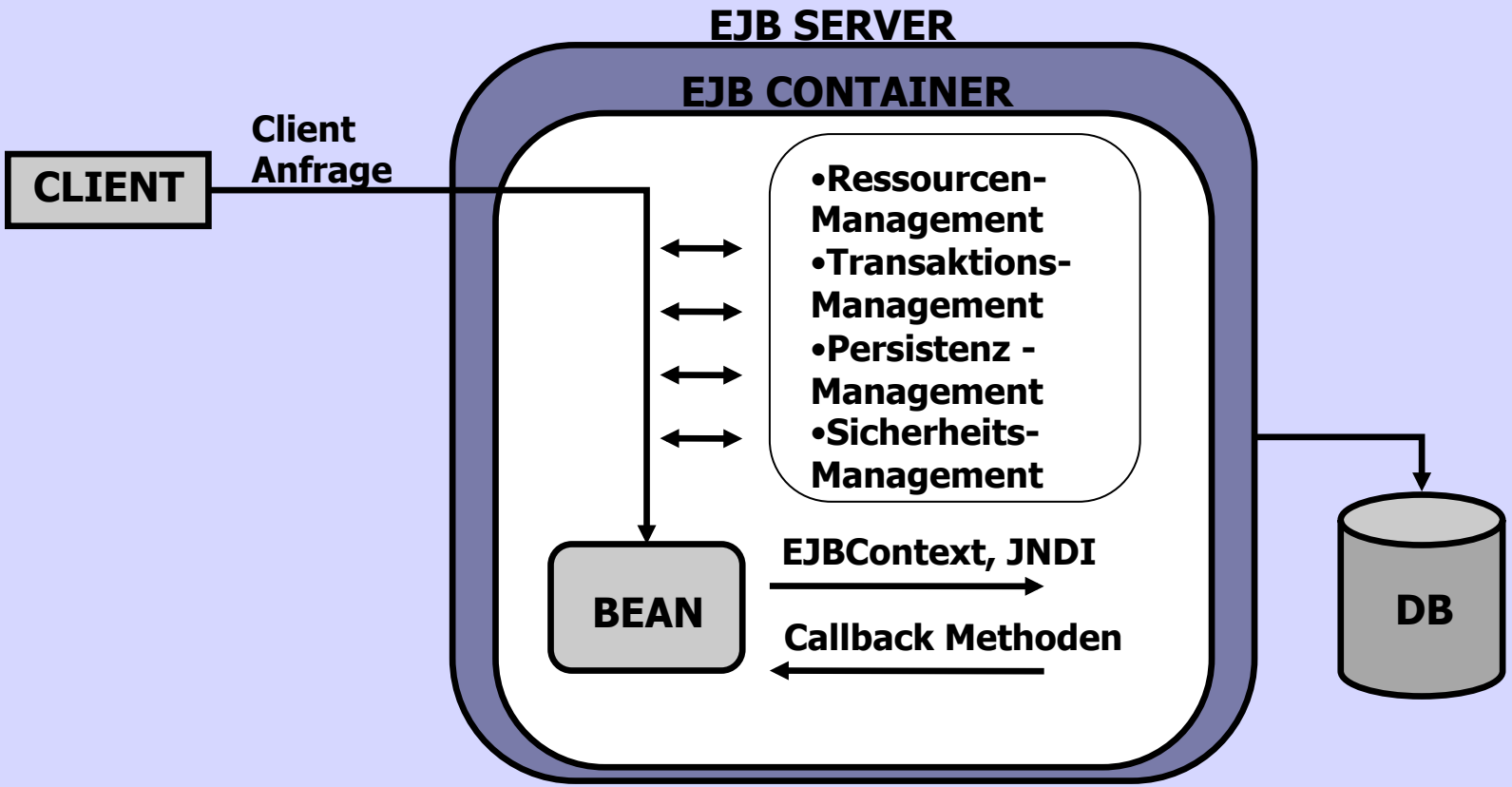


nach: Stark, Abb.7.2

EJB – Enterprise Java Beans

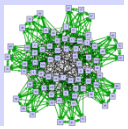
- Teil des J2EE Standard
- serverseitig verteilte Komponenten
- Programmiermodell: Entfernter Methodenaufruf
- durch Standard definiert
  - Art und Struktur der Beans
  - Aufgaben während der Entwicklung
  - verantwortliche Rollen
  - Aufgaben des EJB-Containers

EJB – Enterprise Java Beans

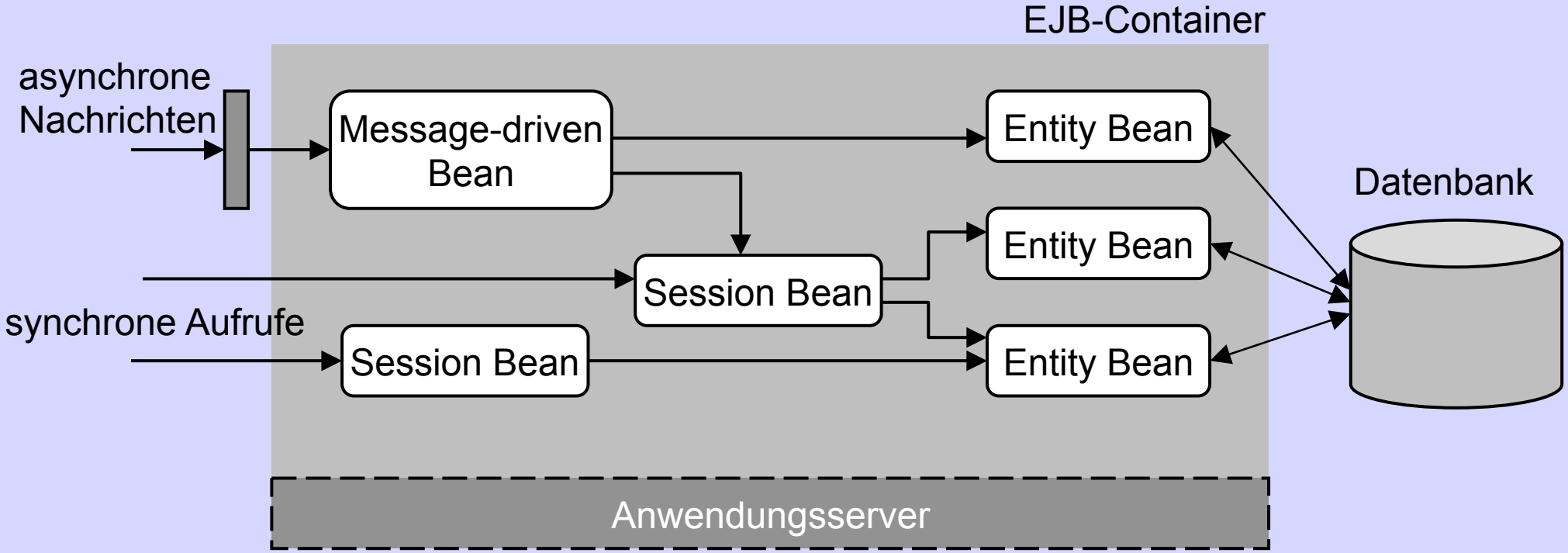


EJB – Arten von Beans

- Entity Beans
  - fachliche Entitäten einer Anwendung (DB-Tabellen)
  - Persistierung der Attribute
- Session Beans
  - Verwaltung von Client-Aufrufen
  - Stateless Session Beans
    - Verwaltung eines Methodenaufrufs
  - Stateful Session Beans
    - Verwaltung einer Client-Sitzung über mehrere Methodenaufrufe hinweg
- Message-driven Beans
  - zustandslos
  - Bearbeitung asynchroner Nachrichten



EJB – Arten von Beans



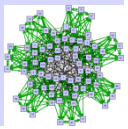
nach: Hammerschall, Abb.9.3

EJB – Aufbau von Beans

- Bestehen aus:
  - verschiedene Klassen und Schnittstellen
  - teils generiert, teils selbst zu programmieren
- Struktur
  - Home- oder Local-Home-Interface
    - Verwaltungsschnittstelle (z.B. create())
  - Remote- oder Local-Interface
    - fachliche Schnittstelle für remote oder local Zugriff
  - Bean-Implementierungsklasse
    - Implementierung aller Schnittstellen
  - Eintrag im Deployment Deskriptor
    - deklarative Beschreibung der Schnittstelle zum Container
  - Primärschlüssel (Zeiger in der DB)
  - Stub, Skeleton, … (verschiedene technische Klassen)

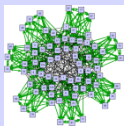
vom Programmierer erstellt

automatisch generiert



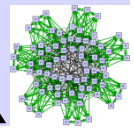
EJB – Deployment Descriptor

- Schnittstelle der Bean zum Container
- Festlegung
  - Name der Bean für Veröffentlichung durch Namensdienst
  - Typ der Bean
  - Name der ausführbaren Dateien
  - Dienste, die Bean von Container erwartet
    - z.B. Abbildung der Bean-Attribute auf Datenbank-Felder
- Angaben in XML



EJB – Entity Beans

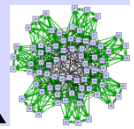
- Repräsentieren Entitäten, auf denen Abläufe arbeiten
  - Instanzdaten liegen als Datensätze in Datenbanken
- Besteht aus
  - Local-Home-Interface
  - Local-Interface
- Local-Home-Interface
  - Verwaltung der Bean zu Laufzeit (auf lokale Anforderung)
  - Methoden:
    - `create()` und `remove()`
    - finder-Methoden
      - Suchabfragen auf der Datenbank
      - liefern Bean-Instanzen mit gesuchten Werten zurück
      - Beispiel: `findByPrimaryKey()`



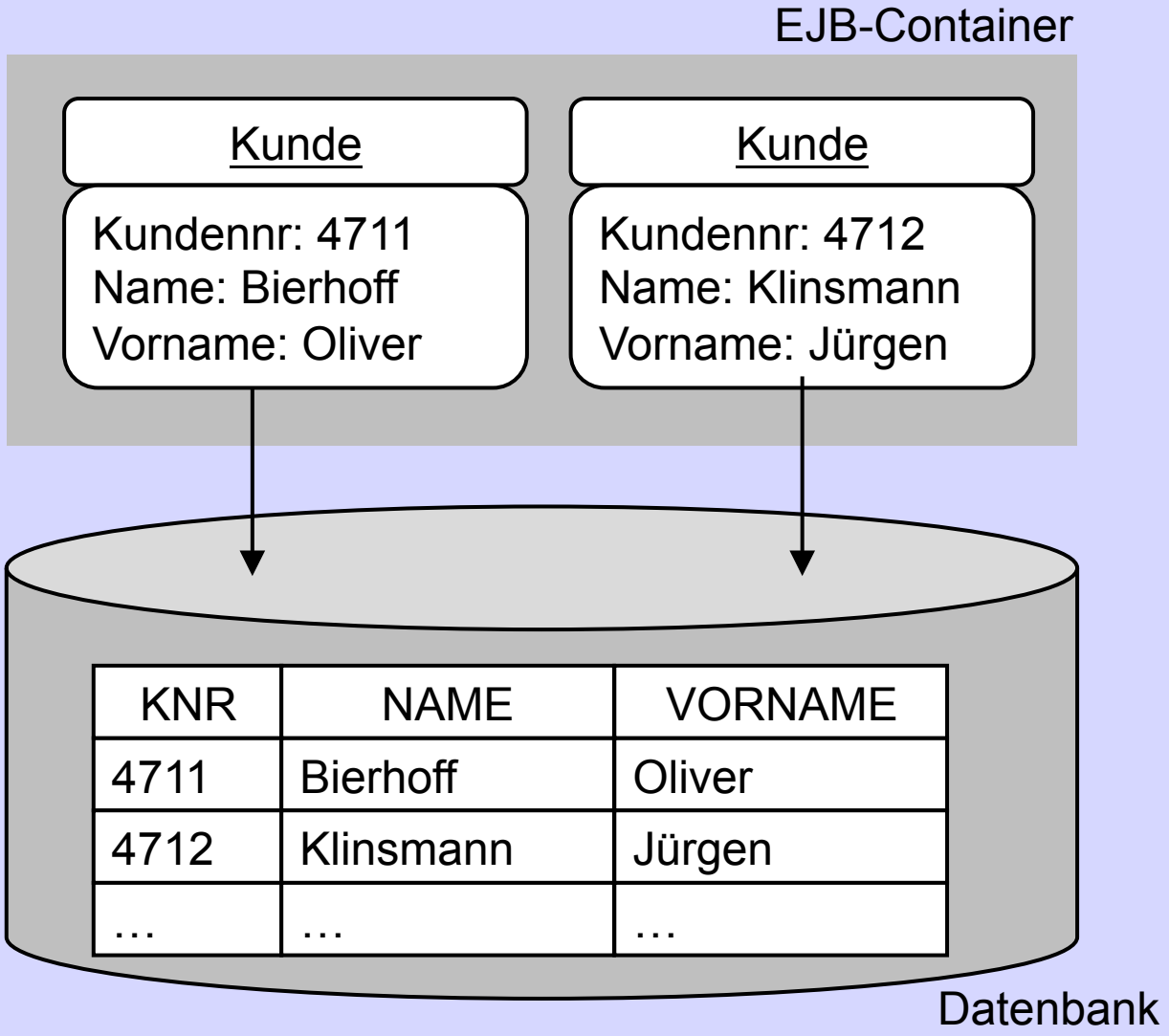
EJB – Entity Beans

- Local-Interface
  - fachliche Schnittstelle für Zugriff aus demselben Container
  - Beispiel
    (http://java.sun.com/j2ee/1.4/docs/tutorial/doc/ Kapitel 27)

```
import java.util.*;
import javax.ejb.*;
public interface LocalPlayer extends EJBLocalObject {
  public String getPlayerId();
  public String getName();
  public Collection getSports() throws FinderException;
}
```



EJB – Entity Beans



EJB – Entity Beans

- Zwei Konzepte zur Erhaltung der Datenkonsistenz
  - Container-managed Persistence (CMP)
    - automatischer Abgleich mit DB-Tabelle
    - erster Ansatz mit einigen Nachteilen (Performance-Probleme)

    → Entwicklung von BMP
    - neue Version: Ausgleichen der Nachteile
  - Bean-managed Persistence (BMP)
    - flexibler und performanter
    - aufwändiger und fehleranfälliger

EJB – Session Beans

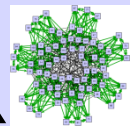
- Repräsentieren Abläufe in Anwendungen
  - Ablauf an Session gebunden
  - Session wird durch Attribute beschrieben
- Arten von Session Beans
  - Stateless und Stateful
  - Gleicher Aufbau
  - unterschiedlicher Eintrag in Deployment Deskriptor

EJB – Session Beans

- Home-Interface
  - Verwaltung der Bean zu Laufzeit durch Container
  - Methoden: `create()` und `remove()`
- Remote-Interface
  - fachliche Schnittstelle für entfernte Zugriffe
  - Beispiel:

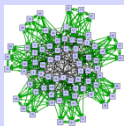
```
public interface Hello extends
                        javax.ejb.EJBObject {
   public String hello()
                        throws java.rmi.RemoteException; }
```

  - `EJBObject` liefert Basismethode für Zugriff auf Bean
    - Beispiel: `isIdentical()`
- Bean-Implementierungsklasse
  - Implementierung der Schnittstellen



EJB – Message-driven Beans

- Stateless Komponenten
- Verarbeiten asynchrone JMS (Java Message Service) Nachrichten von:
  - J2EE Komponente
  - JMS Applikation
  - System ohne J2EE Technologie
- Clients kommunizieren nie direkt mit Message-driven Beans
  - daher: kein Home- und Remote-Interface
- Bean-Implementierungsklasse
  - implementiert Interface `javax.ejb.MessageDrivenBean`
    - dadurch: Festlegung des Bean-Typs
  - implementiert Interface `javax.jms.MessageListener`
    - Empfang von JMS-Nachrichten



EJB – Vorteile

- Gute Performanz: Keine Implementierung von Concurrency Controls, Load Balancing, Fail-Over Support
- Zentrales Deployment: EJB Container
- Zentrale Administration und Sicherheit: in der Business Schicht
- Zentrale Integration von externen Systemen
- Thinner Client Komponenten: Integration von Java Servlets
- EJBs sind komponentenbasiert: besser skalierbar, optimierbar, testbar, verwaltbar
- Freie Auswahl des EJB Servers

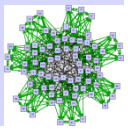
EJB – Nachteile

- Architektonische Zerbrechlichkeit: Ein verteiltes System ist nur so stark wie sein schwächstes Glied
- Evtl. höhere Kosten: Server-Software und Einarbeitung
- Abhängigkeit vom Container / Kosten des Containers
- Lernkurve:
  Einstieg == relativ einfach
  Entwicklung eines guten Systems == sehr komplex

Definition von Web Services

*A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically **WSDL**). Other systems interact with the Web service in a manner prescribed by its description using **SOAP** messages, typically conveyed using HTTP with an **XML** serialization in conjunction with other Web-related standards.*

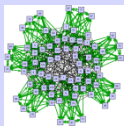
David Booth et al.: *Web Service Architecture*
W3C Working Draft 8 August 2003



Definition von Web Services

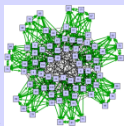
A web service is a piece of business logic, located somewhere on the Internet, that is accessible through standard-based Internet protocols such as HTTP or SMTP. Using a web service could be a simple as logging into a site or as complex as facilitating a multi-organization business negotiation.

Chappell / Jewell:
„Java Web Serivces“
O ‘Reilly, 2002



Web Services

- Komponentenmodell unter Verwendung von Web-Technologien
  - zentral: XML (Serialisierung und Schnittstellenbeschreibung)
- Motivation:
  - Löst ähnliche Probleme wie CORBA
  - Aber offener / allgemeiner als CORBA
- Bisher: große monolitische Informationssysteme
  - Client / Server / Datenbank zu eng verwoben
  - Schlecht dokumentierte Schnittstellen, proprietäre Formate
  - CORBA ORB ‘s nicht überall verfügbar, beschränkt inter-operabel
  - Re-compilation bei kleinsten Änderungen der CORBA IDL

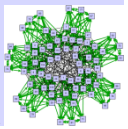


Web Services

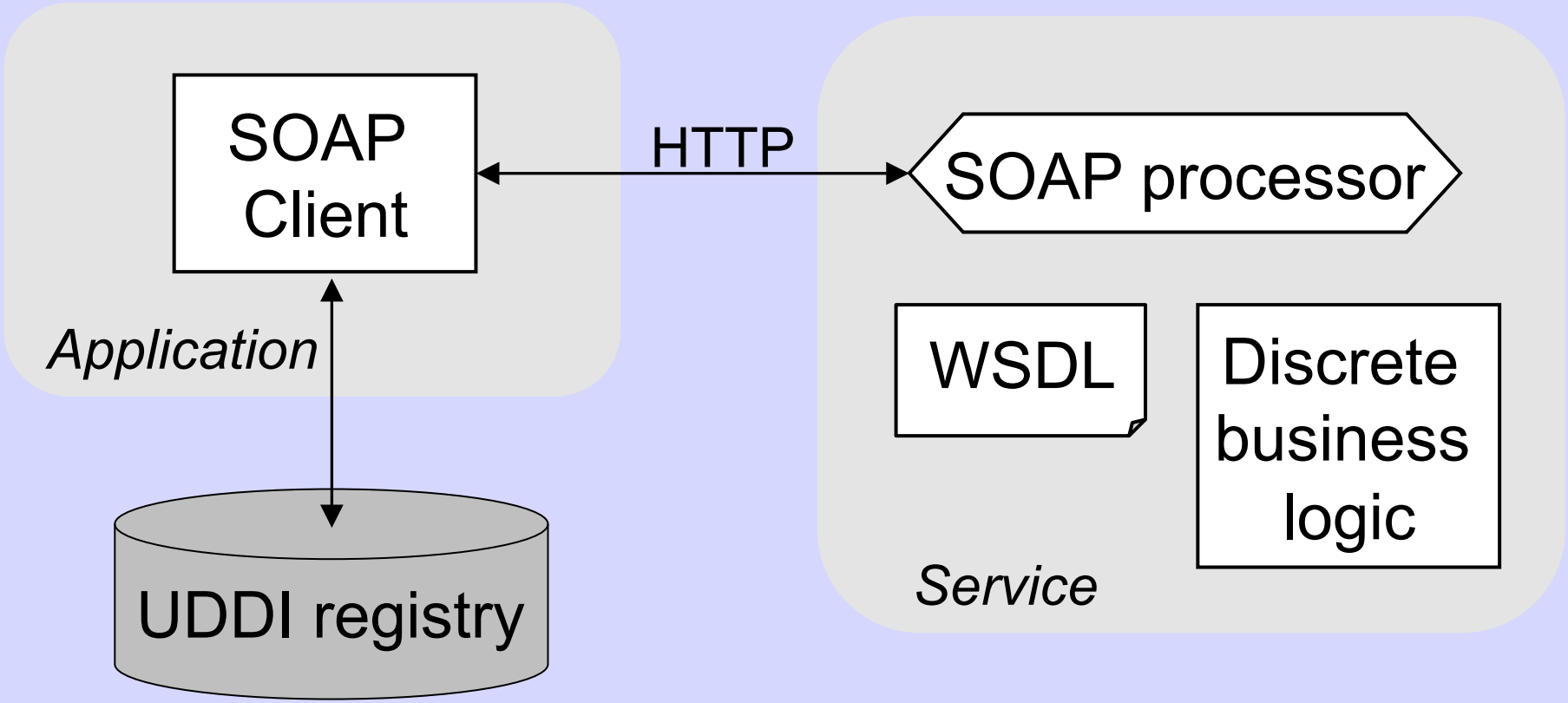
- Vision: Integration verschiedener Business-Komponenten
  - über Abteilungs- und Unternehmensgrenzen hinweg
  - mit light-weight Technologien (XML / HTTP)
- Integration durch Schaffung von Standards
  - vor allem: XML
  - im Unterschied zu z.B. CORBA (z.B. XML statt IIOP)
- Merkmale von Web Services
  - XML-basiert → plattformunabhängig
  - lose Kopplung
  - grobgranular → mehrere Methoden bilden einen Dienst
  - Unterstützung von entfernten Methodenaufrufen
  - Unterstützung von Dokumentenaustausch

Web Services – Schlüsseltechnologien

- **XML**
  - universelle Beschreibungssprache
  - selbst-dokumentierend
  - robust gegen Änderungen: Empfänger überliest irrelevante Einträge
- **WSDL** (Web Service Description Language)
  - Interface Beschreibung von Diensten (analog CORBA IDL)
- **SOAP** (Simple Object Access Protocol)
  - Kommunikation zwischen Diensten („XML-RPC“)
  - Transportiert XML-serialisierte Werte und Methoden-Aufrufe
- **UDDI** (Universal Description, Discovery and Integration)
  - Suchen von Diensten
  - weltweiter Verzeichnisdienst für Web Services

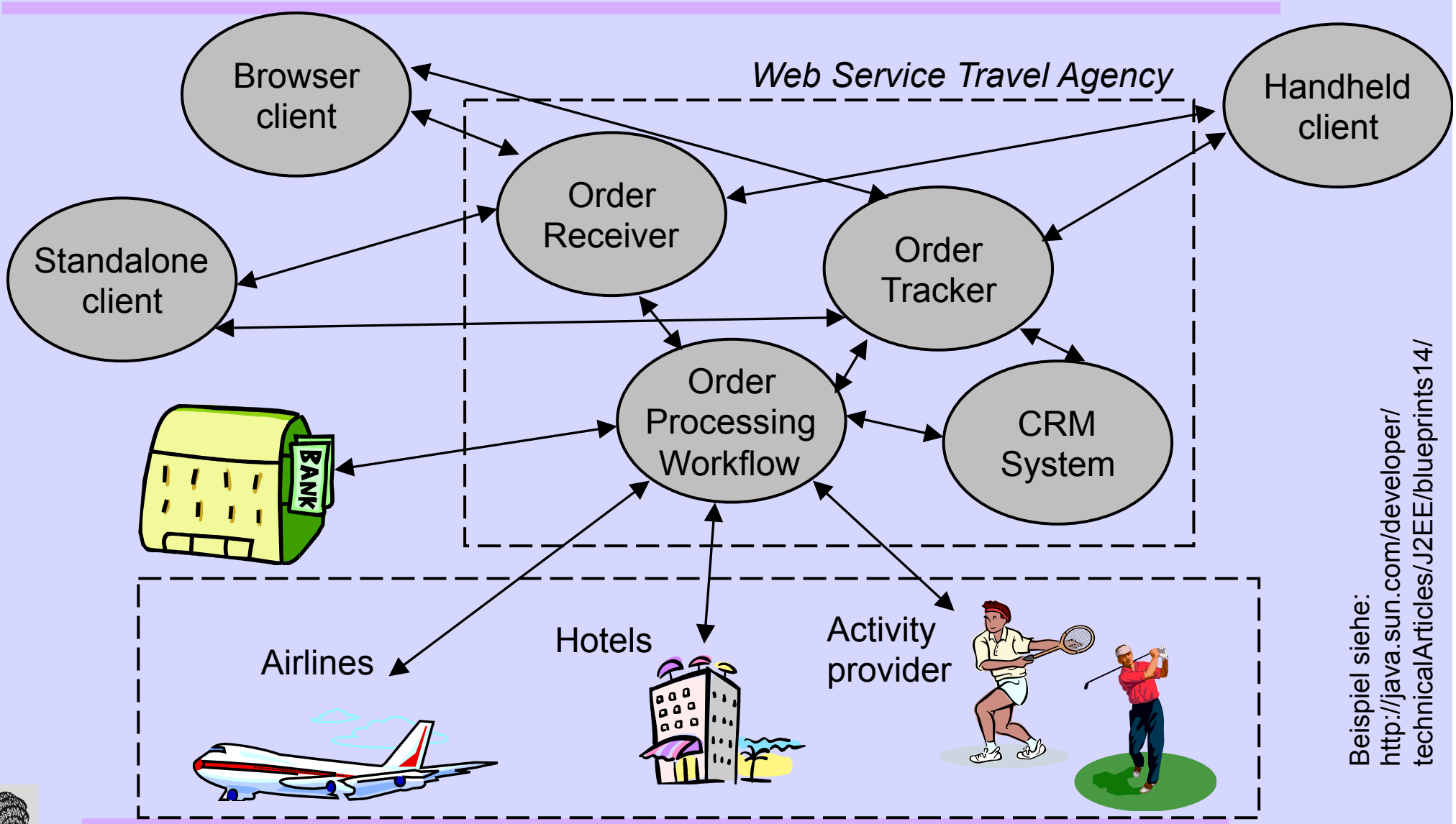


Web Services – einfaches Szenario



Chappell / Jewel: „Java Web Services“, Abb.1-1

Web Services – einfaches Szenario

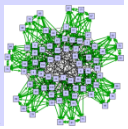


Beispiel siehe:
http://java.sun.com/developer/technicalArticles/J2EE/blueprints14/

SR

XML-RPC

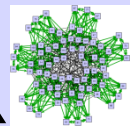
- Idee: Entfernter Methodenaufruf ohne neue Technologien
  - Serialisierung in XML-Dokument
    - Methodenname
    - Parameter
  - Verschicken über HTTP
- Verfügbare Datentypen
  - `<int>`
  - `<double>`
  - `<string>`
  - `<boolean>`
  - `<base64>`
    - Bytefolge
  - `<dateTime.iso8601>`
    - Beispiel: `<dateTime.iso8601>20060125T11:15:12</dateTime.iso8601>`



XML-RPC – Prozeduraufruf

```
HOST /xml-rpc.app HTTP/1.1
Content-type: text/xml
Content-length: 255

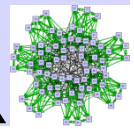
<?xml version="1.0"?>
<methodCall>
   <methodName>calcMaximum</methodName>
   <params>
      <param>
         <value><int>47</int></value>
      </param>
      <param>
         <value><int>23</int></value>
      </param>
   </params>
</methodCall>
```



XML-RPC – Prozedurantwort

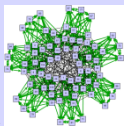
```
HTTP/1.1 200 OK
Content-Type: text/xml
Content-Length: 158

<?xml version="1.0" ?>
<methodResponse>
   <params>
      <param>
         <value><int>47</int></value>
      </param>
   </params>
</methodResponse>
```



XML-RPC – Bewertung

- Vorteil
  - Sehr einfach und schlank
- Nachteile
  - ungenaue Codierung der Datentypen
    - z.B. Probleme mit Datumstyp: keine Zeitzone
  - Aufwändige Codierung binärer Daten (base64)
- Fehlende Metainformation zu den Methoden
  - könnten nur im Request-/Response-Header von HTTP beigefügt werden (→ nicht self-contained)
- → Weiterentwicklung zu SOAP

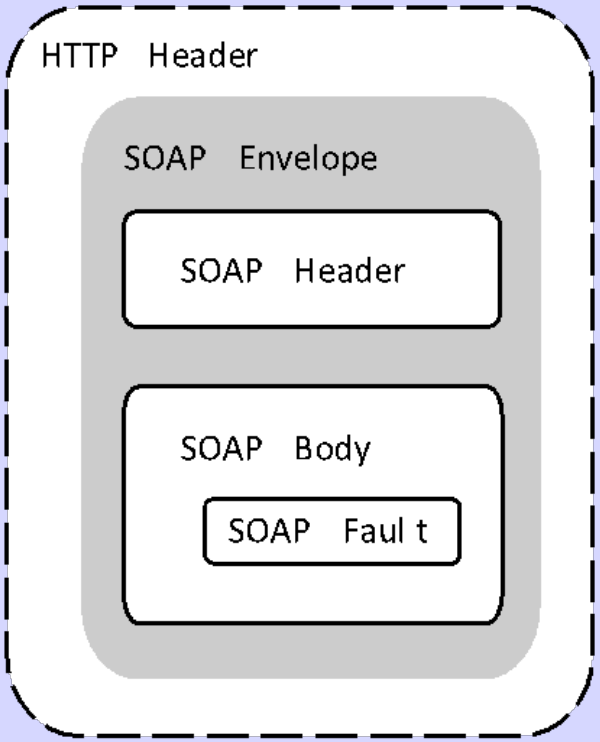


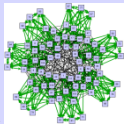
SOAP

- *Simple Object Access Protocol*
- XML-basiertes Nachrichtenprotokoll
- Arbeitet auf bestehenden Transportprotokollen (HTTP, SMTP)
- Aufbau einer SOAP Nachricht
  - Envelope

    Header
    - optional
    - für Metainformationen

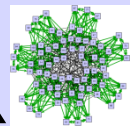
    Body
- kann vollständig im Dokument-Teil eines HTTP-Requests übertragen werden.





SOAP – Nachrichtenformat

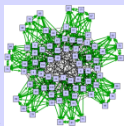
```
<SOAP-ENV:Envelope xmlns:SOAP-ENV= "http://
  www.w3.org/2003/05/soap-envelope/">
    <SOAP-ENV:Header>
          <!-- Header-Information -->
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
          <!-- Body-Information -->
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```



Beispiel für eine SOAP Nachricht

```
POST /satservice HTTP/1.1
Content-Type: text/xml; charset=utf-8
Content-Length: 502

<SOAP-ENV:Envelope xmlns:SOAP-ENV=
                        "http://www.w3.org/2003/05/soap-envelope/">
   <SOAP-ENV:Body>
   <sp:SATProblem xmlns:sp=
       "http://www.informatik.uni-tuebingen.de/satproblem/schema">
       <sp:configuration literalSelector="SPC" learnParameter="13"/>
       <sp:problemInstance>
              <sp:clause>1,-2,-3</sp:clause>
              <sp:clause>-1,2</sp:clause>
              <sp:clause>4</sp:clause>
       </sp:problemInstance>
   </sp:SATProblem>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

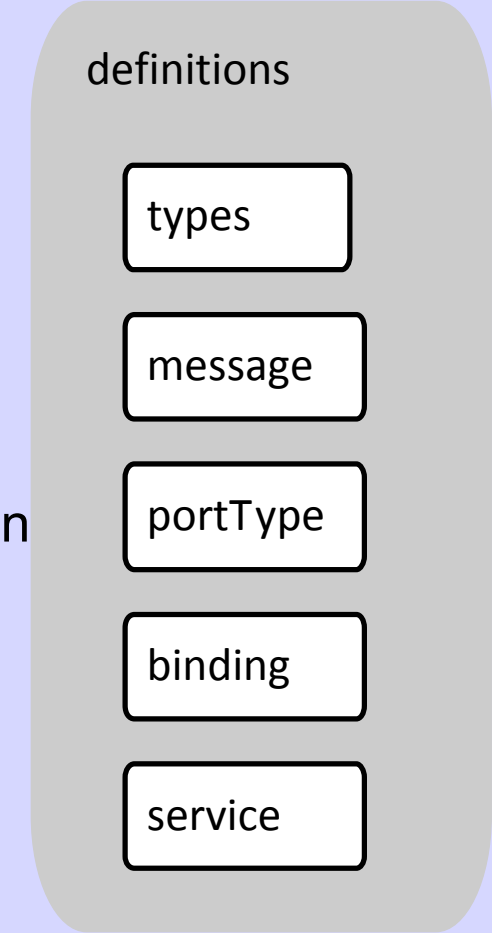


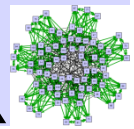
WSDL – Web Services Description Language

- Interface Definition Language for Web Services
- Basiert auf XML Schema
- Aufbau:
- Data Type Definitions
  - Beschreibung der Datentypen, die in Nachrichten vorkommen
- Abstract Operations
  - Die Operationen, die durch die Nachrichten ausgelöst werden
- Service Bindings
  - Abbildung der Nachrichten auf Transportprotokolle

WSDL – Web Services Description Language

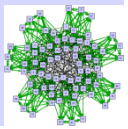
- Data Type Definitions
  - `types` – verwendete Datentypen als XML Schema
  - `message` – Definition der Nachrichten mit Parametern
- Abstract Operations
  - `operation` – Definition, welchem Dienst (Prozedur, Queue etc) die Nachricht zur Behandlung übergeben werden soll
  - `portType` – Abstrakter (Service-) Port als Menge von Operationen (= Interface eines abstrakten Objekts)
  - `binding` – Abbildung eines Port Type auf einen Transportmechanismus (Protokoll: SOAP, TCP, …)
- Service Bindings
  - `port` – Netzwerkadresse zu einem Binding
  - `service` – Menge von Port Types, die gesamthaft einen logischen Dienst darstellen.



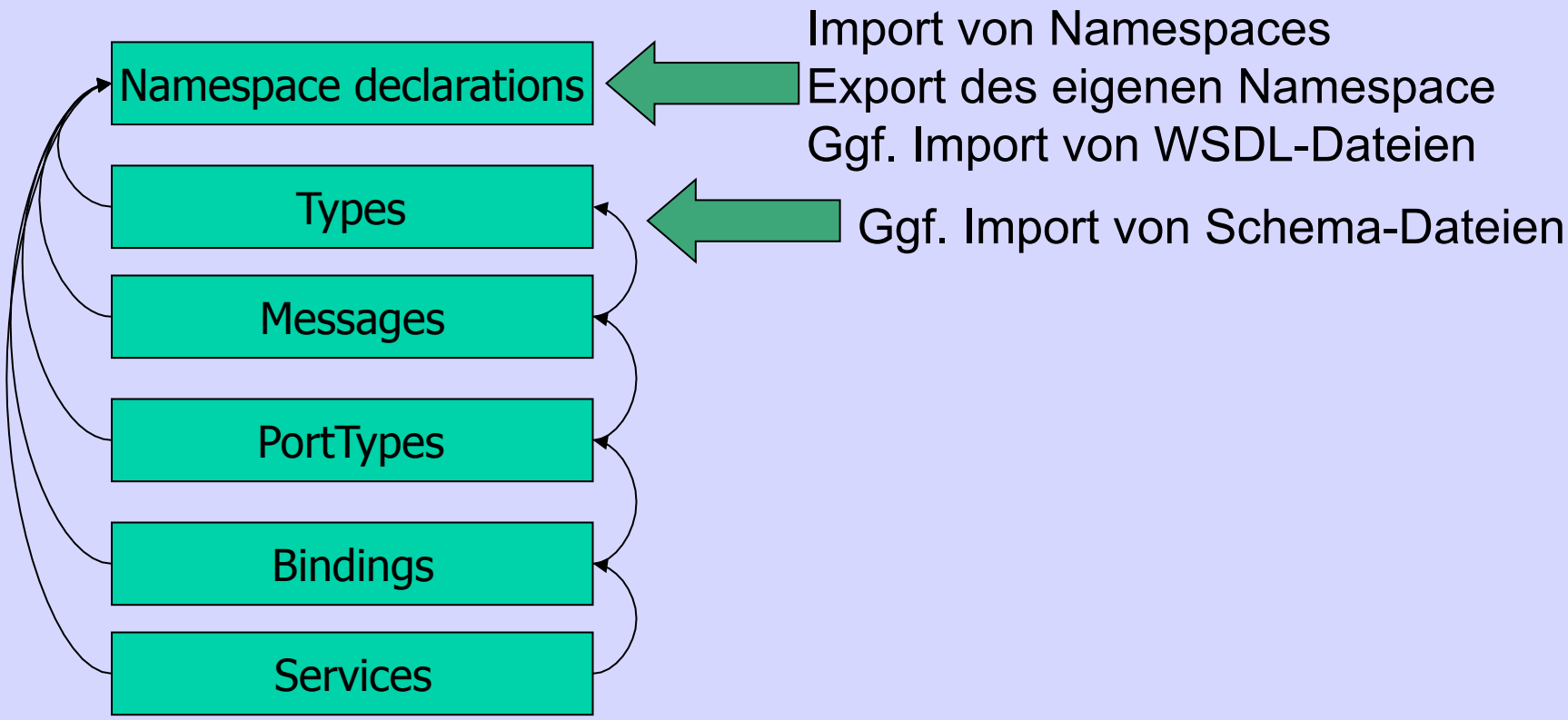


WSDL: Grundideen

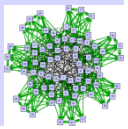
- *Types* und *Messages* definieren **Was** verschickt wird
  - Types = Datentypen (gegeben als XML-Schema)
  - Message = Paket von Datentypen
- *PortTypes* definieren **Welche Funktionalität** ein Dienst bietet.
- *Bindings* definieren **Wie** ein Dienst angesprochen wird
  - „Wie“ im Sinne von „Welches Protokoll?“
- *Services* definieren **Wo** Dienste sich befinden
  - „Wo“ im Sinne von „Wie lautet die Netzwerkadresse?“



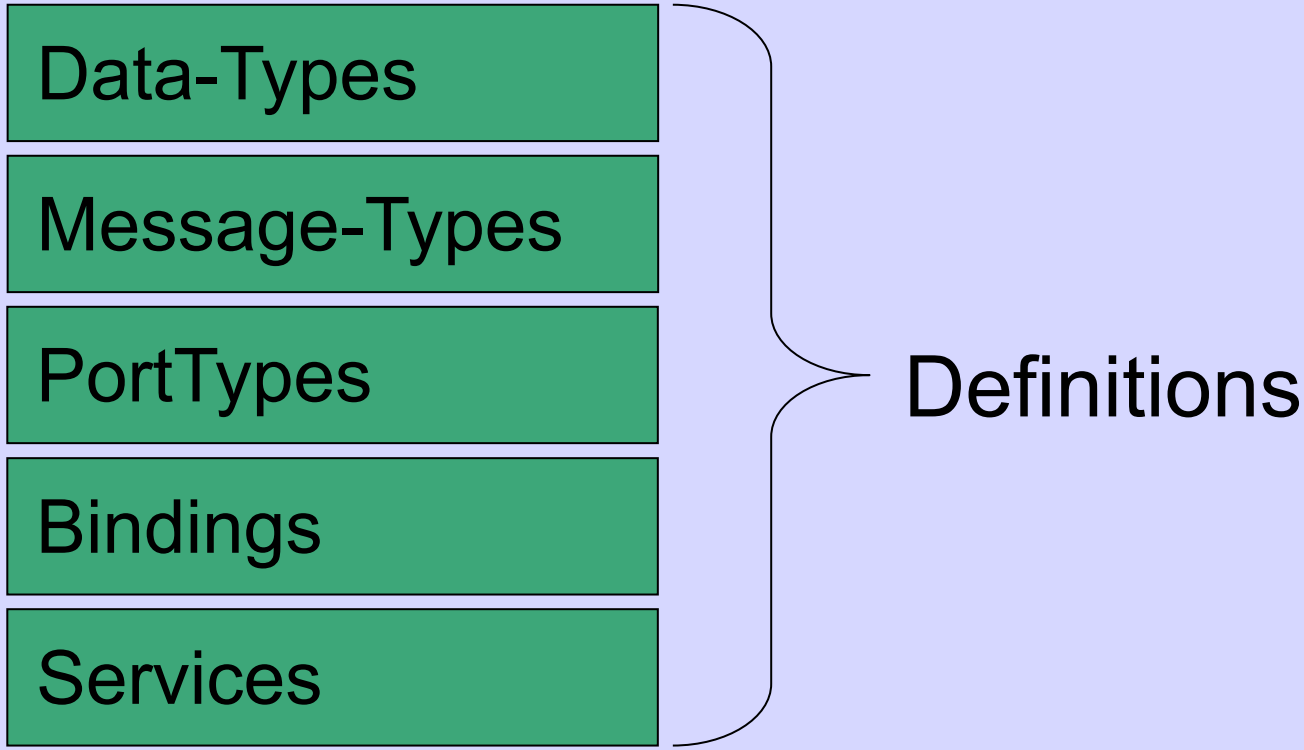
WSDL-Struktur

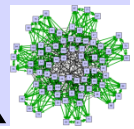


Durch den Import von Namespaces kann WSDL um neue Sprachelemente erweitert werden. Dies wird z.B. von der Web-Service Workflow-Sprache BPEL genutzt.



Schematischer Aufbau einer WSDL-Datei



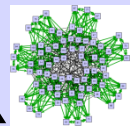


Datentypen im XML-Schema-Format

- Data-Types
- Message-Types
- PortTypes
- Bindings
- Services

Datentypen werden im Format *XML-Schema* definiert, ähnlich wie Records in *C* und Klassen in *Java*.
Es können auch *XML-Schema*-Dateien eingebettet werden.

In XML-Schema können Datentypen durch Einschränkung (z.B. 10 < i < 50) und durch Komposition erstellt werden (z.B. Punkt = (x:Double, y:Double))

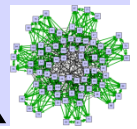


Message-Types: Nachrichtenformate

- Data-Types
- Message-Types
- PortTypes
- Bindings
- Services

Message-Types definieren Nachrichtenformate.
Jeder *Message-Type* besteht aus einer Menge von *Message-Parts*.
Jeder *Message-Part* besitzt einen Namen und einen Datentyp.

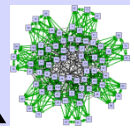
Anmerkung:
Streng genommen sind Message-Types selber wieder eine Art Datentyp.



PortTypes: Schnittstellendefinitionen

- Data-Types
- Message-Types
- PortTypes
- Bindings
- Services

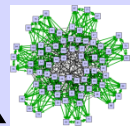
Port-Types definieren Interfaces (analog zu Java).
Jeder *Port-Type* enthält eine Menge von *Operations*.
Eine *Operation* entspricht einer Prozedur bzw. Methode und hat einen bestimmten Nachrichtentyp als Eingabe bzw. Ausgabe.



Bindings: Bindung von PortTypes an Protokolle

- Data-Types
- Message-Types
- PortTypes
- Bindings
- Services

Bindings spezifizieren die Zuordnung der Operationen eines Port-Types auf ein Transport-Protokoll. (Auf welche Art werden die zur Operation gehörenden Messages übertragen?) Standardmäßige Bindings gibt es für SOAP.



Services & Ports

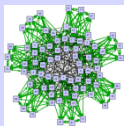
- Data-Types
- Message-Types
- PortTypes
- Bindings
- Services

Services enthalten so genannte *Ports*. Jeder Port implementiert einen *Port-Type* mit einem bestimmten *Binding*. Zusätzlich enthält ein Port eine Adresse, deren Format vom *Binding* abhängig ist (z.B. eine URL bei SOAP).

Jeder Service besteht aus einer Menge von Tripeln (PortType, Binding, Adresse)!

UDDI

- Anforderungen
  - Veröffentlichen von Web Services
  - Finden von Web Services
- Anbieter von UDDI-Repositories: Microsoft, SAP und IBM
- Benutzung
  - Web-Interface
  - API z.B. JAXR (Java API for XML Registries)
- Beschreibung der Web Services mittels XML-Datenstrukturen
  - Business Entity
    - Kontakt, Beschreibung, Beziehung zu anderen Geschäftseinheiten, …
  - Service
    - Web Service oder andere Dienstleistungen
  - Binding
    - Technische Beschreibung, Access point URL, Verweis auf Spezifikation
  - …
- am wenigsten genutzter Standard

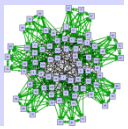


WSDL 1.1 und 2.0

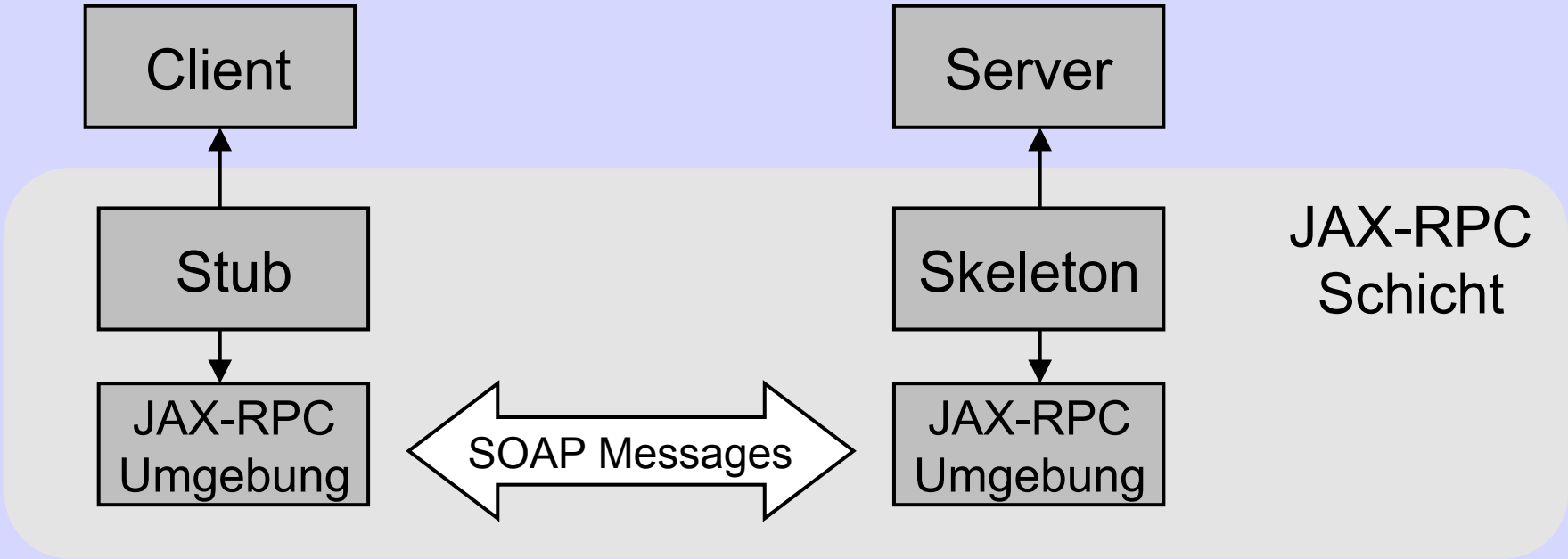
- Es wurde der WSDL-Standard 1.1 vorgestellt
  - Dieser besteht seit 2001 und wird heute allgemein unterstützt.
- Seit September 2007 gibt es WSDL 2.0
  - WSDL 2.0 wird bisher nur selten genutzt.
  - WS-BPEL 2.0 unterstützt z.B. nur WSDL 1.1
- Hauptunterschiede:
  - *PortTypes* wurden in *Interfaces* umbenannt
  - *Ports* wurden in *Endpoints* umbenannt.
  - Es gibt keine *Messages* mehr.
  - Unterstützung für sogenannte „RESTful Web Services“

Java Web Service Developer Pack

- Im JWSDP u.a. enthalten:
  - Java Architecture for XML Binding (JAXB)
    - Generierung von Java Klassen aus DTDs
  - Java API for XML Processing (JAXP)
    - SAX (Simple API for XML Parsing)
      - ereignisgesteuerter Parser
    - DOM (Document Object Model)
      - Objekt Repräsentation in Form eines Baums
  - Java API for XML-based RPC (JAX-RPC)
    - Methodenaufrufe und Rückgabewerte als SOAP Nachricht
    - Erzeugung von Stubs und Ties aus WSDL Beschreibung
  - SOAP with Attachments API for Java (SAAJ)
    - Erstellen und Verschicken von SOAP Nachrichten mit Anhängen
    - Asynchroner Nachrichtenaustausch

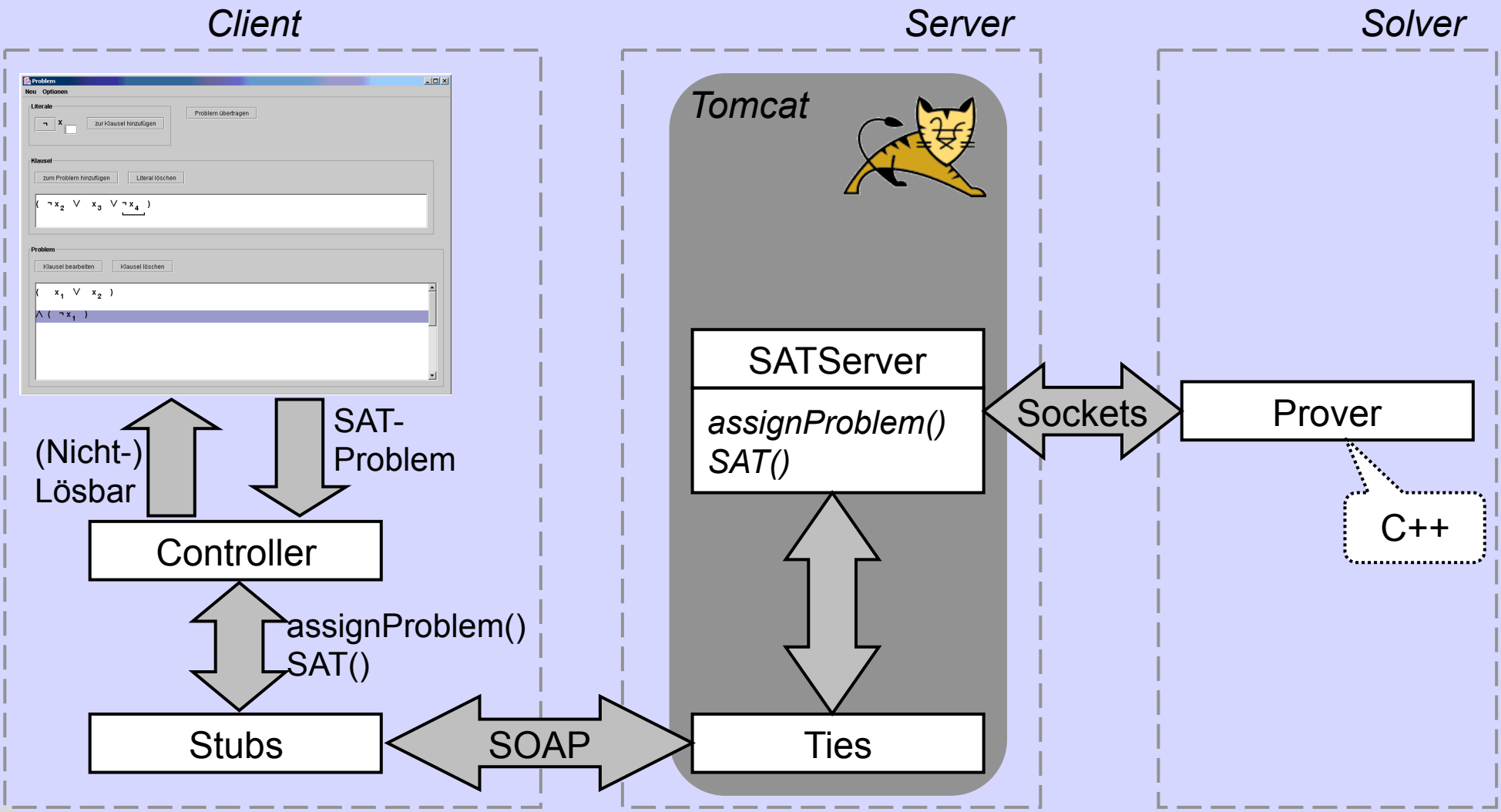


JAX-RPC



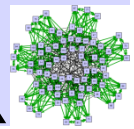
Stark: „J2EE“, Abb.12-2

Beispiel: SAT-Solver als Web Service

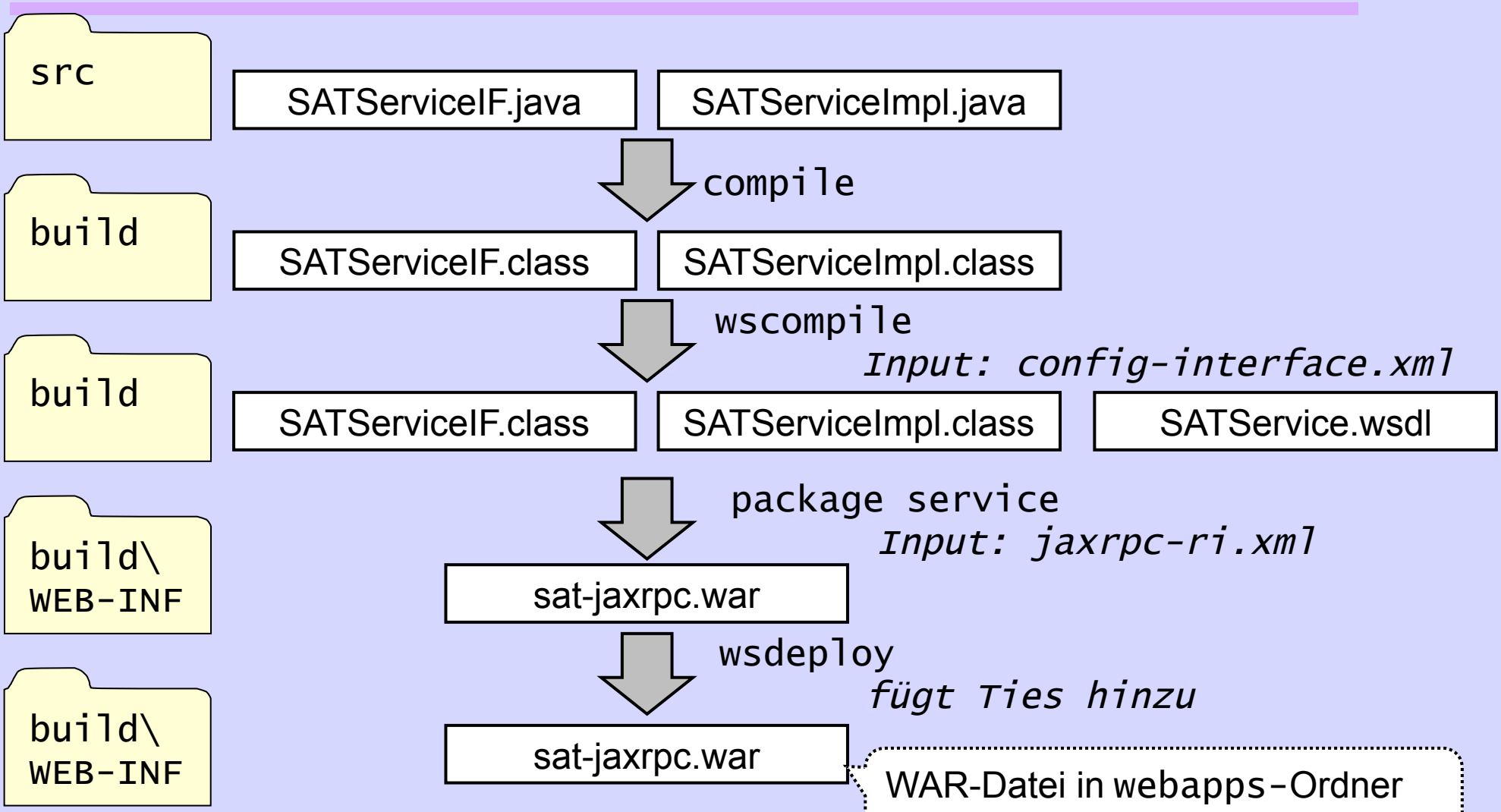


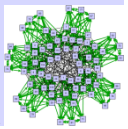
Beispiel: Schritte zu einem Web Service

1. Code the Service Endpoint Interface (SEI) and implementation class and interface configuration file
2. Compile the service classes
3. Use `wscompile` to generate the files required to deploy the service
4. Package the files into a WAR file
5. Deploy the WAR file
   - The tie classes (which are used to communicate with clients) are generated by the Application Server during deployment
6. Code the client class
7. Use `wscompile` to generate and compile the stub files
8. Compile the client class
9. Run the client



Beispiel: Service Deployment





Beispiel: Service Endpoint Interface

```
package SATService;

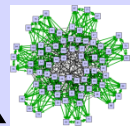
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.LinkedList;

public interface SATServiceIF extends Remote {

  public boolean SAT() throws RemoteException;
  public LinkedList assignProblem(LinkedList p)
                              throws RemoteException;

  // ...
}
```

leeres Interface



Beispiel: Service Implementierung

```
package SATService;

import java.rmi.RemoteException;
import java.util.LinkedList;

public class SATServiceImpl implements SATServiceIF {

   LinkedList problem = new LinkedList();

   public boolean SAT() {
        // calls Prover via Sockets
   }

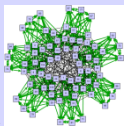
   public LinkedList assignProblem(LinkedList p) throws RemoteException {
        problem = p;
        return problem;
   }

   // ...
}
```

Beispiel: SATService.wsdl (1)

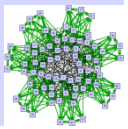
```
<?xml version="1.0" encoding="UTF-8"?>

<definitions ...>
  <types>
    <schema ...>
      <import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
      <complexType name="linkedList">
        <complexContent>
          <extension base="tns:list">
            <sequence/></extension></complexContent></complexType>
      <complexType name="list">
        <complexContent>
          <extension base="tns:collection">
            <sequence/></extension></complexContent></complexType>
      <complexType name="collection">
        <complexContent>
          <restriction base="soap11-enc:Array">
            <attribute ref="soap11-enc:arrayType" wsdl:arrayType="anyType[]"/>
          </restriction>
        </complexContent>
      </complexType></schema>
  </types>
```



Beispiel: SATService.wsdl (2)

```
<message name="SATServiceIF_SAT"/>
<message name="SATServiceIF_SATResponse">
  <part name="result" type="xsd:boolean"/></message>
<message name="SATServiceIF_assignProblem">
  <part name="LinkedList_1" type="ns2:linkedList"/></message>
<message name="SATServiceIF_assignProblemResponse">
  <part name="result" type="ns2:linkedList"/></message>
<portType name="SATServiceIF">
  <operation name="SAT" parameterOrder="">
    <input message="tns:SATServiceIF_SAT"/>
    <output message="tns:SATServiceIF_SATResponse"/>
  </operation>
  <operation name="assignProblem" parameterOrder="LinkedList_1">
    <input message="tns:SATServiceIF_assignProblem"/>
    <output message="tns:SATServiceIF_assignProblemResponse"/>
  </operation>
</portType>
```



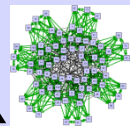
Beispiel: SATService.wsdl (3)

```
<binding name="SATServiceIFBinding" type="tns:SATServiceIF">
  <operation name="SAT">
    <input>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 use="encoded" namespace="urn:Foo"/></input>
    <output>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 use="encoded" namespace="urn:Foo"/></output>
    <soap:operation soapAction=""/></operation>
  <operation name="assignProblem">
    <input>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 use="encoded" namespace="urn:Foo"/></input>
    <output>
      <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
 use="encoded" namespace="urn:Foo"/></output>
    <soap:operation soapAction=""/></operation>

<soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
</binding>
```

Beispiel: SATService.wsdl (4)

```
  <service name="SATService">
    <port name="SATServiceIFPort"
              binding="tns:SATServiceIFBinding">
      <soap:address location= "REPLACE_WITH_ACTUAL_URL"/>
   </port>
  </service>
</definitions>
```



Beispiel: Client Controller.java (1)

```
public class Controller {

    static SATServiceIF serviceStub;
    Problem ph = new Problem();

    public Controller(){
          ProblemDialog p = new ProblemDialog(this,ph);
    }

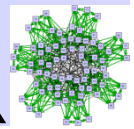
    public static void main(String[] args) {
          System.out.println("Endpoint address = " + args[0]);
          try {
                    Stub stub = createProxy();
                    stub._setProperty(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY,args[0]);
                    serviceStub = (SATServiceIF)stub;

          } catch (Exception ex) {
                              ex.printStackTrace();
          }
          Controller scm = new Controller();

    }

    private static Stub createProxy() {
          return (Stub)(new SATService_Impl().getSATServiceIFPort());
    }
```

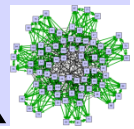
z.B.:
http://localhost:8080/sat-jaxrpc/SAT
Je nachdem wo Server läuft und welche URL in Konfigurationsdateien angegeben wurde.



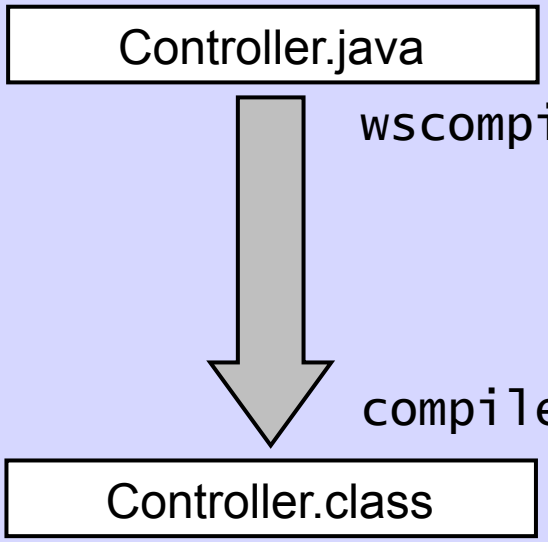
Beispiel: Client Controller.java (2)

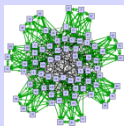
```
public void SAT(){
    System.out.println("Controller:
                        SAT() wird entfernt aufgerufen.");
    try{
          boolean b = serviceStub.SAT();
          ph.setSolvable(b);
    } catch (Exception ex) { /*... */ }
}

public void assignProblem(LinkedList l){
    System.out.println("Controller:
                        assignProblem() wird entfernt
                        aufgerufen.");
    try{
          ph.setProb(serviceStub.assignProblem(l));
    } catch (Exception ex) { /*... */ }
}
```



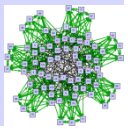
Beispiel: Service Deployment





RESTful Web Services

(Beitrag von Markus Held, 2009)

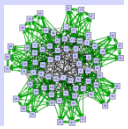
SR 

Von WSDL+SOAP-Web Services zu REST-WS

- Es gibt so viele verschiedene „WS-*“-Standards, dass viele Entwickler den Überblick verloren haben
- Die Beschreibung von Schnittstellen mit WSDL und XML Schema ist kompliziert und allgemein
  - Oftmals overkill
- Alternative: „REST“-Web Services
  - Request / response (RPC-)Protokoll mit messages genügt
  - Zustandslose Web-Services. Alle Information in RPC-messages
  - Als message Typen genügen HTTP POST und GET (Request parameter werden in URL übertragen)
  - Konsequenz: Pro Operation eine URL

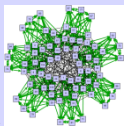
REST Web Services

- REST = Representational State Transfer
  - „REST“ stammt aus der Dissertation von Roy Fielding, Mitentwickler des HTTP-Standards und ehemaliger Leiter des Apache-Projekts
- Ursprünglich: Architektur*stil* des World Wide Web
  - URL-basierte Request / Response Kommunikation ohne (impliziten) Zustand im Server
  - Die WWW-Architektur ist eine konkrete Ausprägung des REST-Architekturstils.
- Heute: Architekturstil *auch* für Web Services
  - HTTP-basiert
  - Häufig einfacher als SOAP
  - Dienste werden als Ressourcen betrachtet, nicht als Objekt-Methoden. Darauf POST und GET anwendbar.



REST (Representational State Transfer)

- Dienst = Menge von Ressourcen und darauf anwendbare Operationen
  - Vgl. WSDL: Dienst = Menge von Operationen
- Ressourcen in der Praxis
  - Ressourcen sind z.B. Webseiten, Bilder, Servlets…
  - Über URLs adressierbar
  - Operationen create, read, update und delete
- Rückgabeformate für Operationen:
  - Text
  - XML
  - JSON (= „JavaScript Object Notation“)

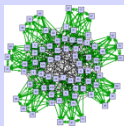


REST (2)

- Aufrufe sind zustandslos
  - D.h. unabhängig von vorhergehenden Aufrufen
  - Alle Informationen müssen über die URL mitgegeben werden
- Code on demand
  - Programmcode wird dynamisch beim Aufruf einer Funktion nachgeladen
- Viele Webservices arbeiten nach den REST-Prinzipien
  - Google Charts
  - Flickr
  - Amazon Webservices

Grundprinzipien von REST

- Ressource
  - Funktion, die für einen bestimmten Zeitpunkt eine Menge von Entitäten bzw. Werten liefert, die als äquivalent betrachtet werden.
  - Ressourcen können statisch sein, oder sich im Laufe der Zeit ändern
  - Bsp.: Webseite
- Resource Identifier
  - Referenz auf eine Ressource
  - Bsp.: URL
- Representations
  - Darstellung einer Ressource durch Byte-String + Metadaten
  - Bsp.: HTML-Datei, Flash-Datei
- Control Data
  - Definieren den Zweck einer Nachricht zwischen Client und Server.



REST-Verben

- PUT
  - Idempotent
  - Erzeuge Ressource
- GET
  - Idempotent
  - Abfragen
- POST
  - Nicht idempotent
  - Modifiziere Ressource
  - Seiteneffekte
- DELETE
  - Idempotent
  - Zerstöre Ressource

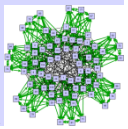
- CRUD
  - Create
  - Read
  - Update
  - Delete
- Vergleiche SQL
  - Create = Insert
  - Read = Select
  - Update = Update
  - Delete = Delete

Beispiel

http://api.flickr.com/services/rest/?method=flickr.**photos.search**&**tags=blume**

XML-Ausgabe:

```
<rsp stat=„ok“>
  <photos page=“1“ …..>
    <photo id=„2760283136“ …./>
    <photo id=„2760277760 …/>
    …
  </photos>
</rsp>
```



Vorteile von REST

- Verwendung von Standards
  - HTTP, XML, URI, MIME
- Skalierbarkeit
  - Unterstützung für leichtgewichtige Nachrichtenformate (z.B. JavaScript Object Notation)
  - Unterstützung von Caches, Proxies und Lastverteilung
- Kommunikation höchst heterogener Systeme
  - Client benötigt keine Details über die Implementierung

Nachteile von REST

- Es gibt keine standardisierte Schnittstellen-beschreibung
- HTTP-Operationen PUT und DELETE werden von einigen Firewalls geblockt
  - Lösung: PUT und DELETE über POST implementieren

- URIs besitzen maximale Größe (4kb)
  - Komplexe Anfragen können dieses Maximum evtl. überschreiten (Fehlermeldung, Absturz des Servers)

Vergleich: SOA, REST, CORBA, RMI, RPC

- RPC und RMI ermöglichen die Integration verteilter Module, die in der jeweils selben Programmiersprache implementiert wurden
- SOA und CORBA haben als Ziel die Integration von in unterschiedlichen Programmiersprachen implementierten Modulen über das Netz
- REST ist eine Alternative zu SOA
  - SOA baut zwar auf dem Protokollstack des WWW auf, nutzt aber nicht dessen Eigenheiten
  - Häufig scheinen REST Web Services einfacher zu sein, als ihre SOA-Gegenstücke

Vergleich: SOA, REST, CORBA, RMI, RPC

	SOA	REST	CORBA	RMI	RPC
Objektzustand	nein	nein	ja	ja	nein
Programmierspr.	beliebig	beliebig	Implementierungs-abhängig	Java	C
Interfacebeschr.	WSDL	Noch kein Standard (WSDL 2, WADL)	IDL	Java	IDL
Stub-Erzeugung	Compiler	manuell oder über ein Framework (z.B. Rails)	Compiler	Compiler	Compiler
Registry	UDDI	-	ORB	RMI-Registry	-
Kommunikations-protokoll	i.d.R. SOAP/ HTTP	HTTP	IIOP	RMI-Protokoll oder IIOP	
Datenversand	Text (XML)	Text (XML, JSON)	binär	binär	binär