

# *Web-Dienste mit XML und SOAP*

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken



# *Funktionalität und Präsentation*

In der Softwaretechnik gilt es als wichtiges Entwurfsprinzip, die *Funktionalität* (Semantik) eines Systems von der *Präsentation* (Syntax) zu trennen:

- Dienste können durch *verschiedene Präsentationen* angesprochen werden
- Bessere *Automatisierung* der Dienste
- Besseres *Zusammenspiel* mehrerer Teile
- Separation der Interessen

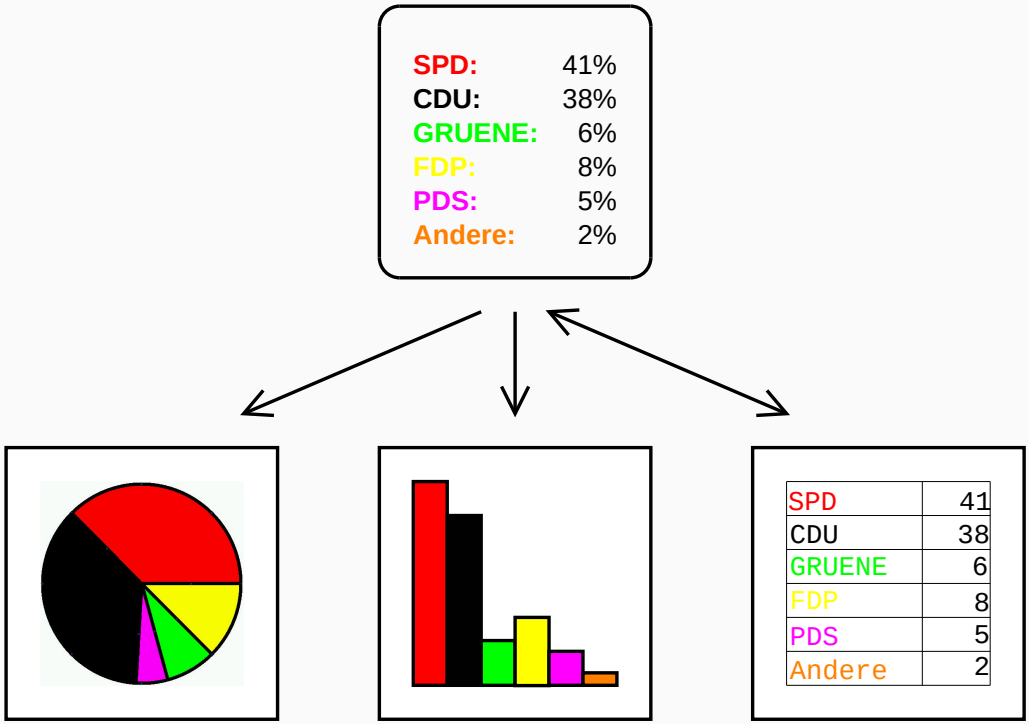






# *Model-View-Controller*

realisiert verschiedene *Präsentationen* eines Modells

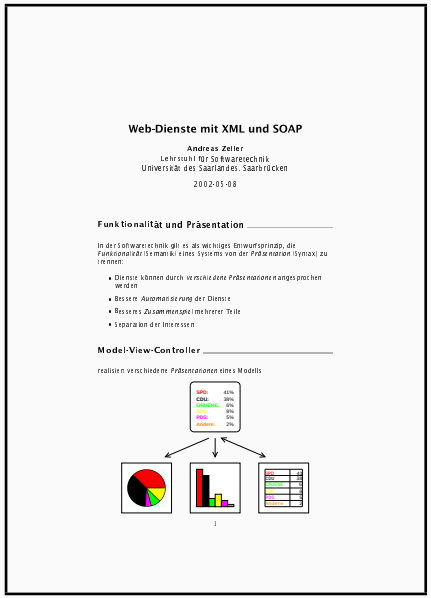


# Dokumente



In LaTeX kann derselbe Inhalt auf verschiedene Weisen präsentiert werden:

z.B. als Dokument. . .



Web-Dienste mit XML und SOAP

Andreas Zeller
Lehrstuhl für Softwaretechnik
Universität des Saarlandes, Saarbrücken

2002-05-08

**Funktionalität und Präsentation**

In der Softwaretechnik gilt es als wichtiges Entwurfsprinzip, die *Funktionalität* (Semantik eines Systems von der *Präsentation* (Syntax) zu trennen:

- Dienste können durch *verschiedene Präsentationen* angesprochen werden
- Bessere *Automatisierung* der Dienste
- Besseres *Zusammenspiel* mehrerer Teile
- Separation der Interessen

**Model-View-Controller**

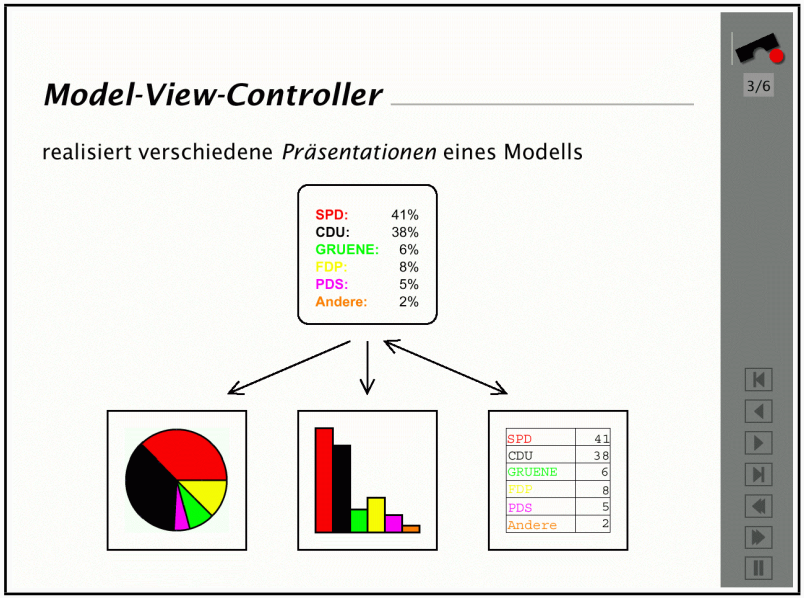
realisiert verschiedene *Präsentationen* eines Modells

| | |
|---|---|
| SPD: | 41% |
| CDU: | 38% |
| GRUENE: | 6% |
| FDP: | 8% |
| PDS: | 5% |
| Andere: | 2% |

| | |
|---|---|
| SPD | 41 |
| CDU | 38 |
| GRUENE | 6 |
| FDP | 8 |
| PDS | 5 |
| Andere | 2 |

1

. . . und als Folie



**Model-View-Controller**

3/6

realisiert verschiedene *Präsentationen* eines Modells

| | |
|---|---|
| SPD: | 41% |
| CDU: | 38% |
| GRUENE: | 6% |
| FDP: | 8% |
| PDS: | 5% |
| Andere: | 2% |

| | |
|---|---|
| SPD | 41 |
| CDU | 38 |
| GRUENE | 6 |
| FDP | 8 |
| PDS | 5 |
| Andere | 2 |

# *Skriptsprachen*



In Skriptsprachen werden Anwendungen ebenfalls in *Funktionalität* und *Präsentation* aufgeteilt:

- Funktionalität wird herkömmlich realisiert und einzelnen *Kommandos* (in Perl, Python, Tcl...) zugeordnet







# *Skriptsprachen*



In Skriptsprachen werden Anwendungen ebenfalls in *Funktionalität* und *Präsentation* aufgeteilt:

- Funktionalität wird herkömmlich realisiert und einzelnen *Kommandos* (in Perl, Python, Tcl...) zugeordnet
- Kommandos werden aufgerufen
  - von *graphischer Benutzeroberfläche*
  - innerhalb von *automatischen Skripten*
- Kommandos + Dienste sind als Bibliothek überall einsetzbar







# *Im Netz*

ist die Trennung von *Funktionalität* und *Präsentation* nicht so weit fortgeschritten:


und ähnliche Ärgernisse – festgelegte Fontgrößen, Farben, Browser. . .

Suche in Google nach „best viewed with“ liefert ungefähr 1.410.000 Dokumente.

# *Im Netz (2)*



Hintergrund: Das ursprünglich *präsentationsneutrale* HTML wurde mit immer weiteren *präsentationsspezifischen* Erweiterungen angereichert:

`<FONT>` · `<TABLE WIDTH=...>` · `<BLINK>` · `<SCRIPT>` ···

Folge:

- Aufgezwungene Präsentationsform
- Automatisierung nur schlecht möglich
- Zusammenspiel der Teile nur schlecht möglich









# *Übersicht*

Wir betrachten die folgenden Techniken zur Realisierung *automatisierter*, *verteilter*, *präsentationsunabhängiger* Dienste über das Internet:

**HTTP** (*HyperText Transfer Protocol*) übermittelt Dienste und Ergebnisse

**XML** (*Extended Markup Language*) trennt den *Inhalt* von seiner *Darstellung*

**SOAP** (*Simple Object Access Protocol*) ermöglicht Zugriff auf Dienste über XML und HTTP

**WSDL** (*Web Services Description Language*) ermöglicht *selbstbeschreibende Dienste*



# *Kommunikationsprotokolle*

Es gibt dezidierte Protokolle zur Kommunikation von Komponenten:

**DCOM** (*Distributed COM*) von Microsoft (.NET)

**IIOP** (*Internet Inter-ORB Protocol*) von CORBA

**RMI** (*Remote Method Invocation*) aus der Java-Welt (SunONE)

Diese Protokolle realisieren effizient und effektiv *verteilte, synchrone* Dienstanforderungen.







# *Kommunikationsprotokolle (2)*



Probleme mit dezidierten Protokollen:

- Aufwendige Infrastruktur mit zahlreichen Diensten vonnöten ⇒ muß gewöhnlich eingekauft werden







# *Kommunikationsprotokolle (2)*



Probleme mit dezidierten Protokollen:

- Aufwendige Infrastruktur mit zahlreichen Diensten vonnöten ⇒ muß gewöhnlich eingekauft werden
- Abhängigkeit vom Hersteller der Kommunikations-Infrastruktur (Microsoft, CORBA-Anbieter, Java-Anbieter)







# *Kommunikationsprotokolle (2)*



Probleme mit dezidierten Protokollen:

- Aufwendige Infrastruktur mit zahlreichen Diensten vonnöten ⇒ muß gewöhnlich eingekauft werden
- Abhängigkeit vom Hersteller der Kommunikations-Infrastruktur (Microsoft, CORBA-Anbieter, Java-Anbieter)
- Schlecht für offene, skalierbare Anwendungen geeignet (jeder Client braucht seinen ORB-Zugang)







# *Kommunikationsprotokolle (2)*



Probleme mit dezidierten Protokollen:

- Aufwendige Infrastruktur mit zahlreichen Diensten vonnöten ⇒ muß gewöhnlich eingekauft werden
- Abhängigkeit vom Hersteller der Kommunikations-Infrastruktur (Microsoft, CORBA-Anbieter, Java-Anbieter)
- Schlecht für offene, skalierbare Anwendungen geeignet (jeder Client braucht seinen ORB-Zugang)

Kein Problem in einer geschlossenen Umgebung (z.B. einer Server-Farm), aber in *offenen* Umgebungen wie dem Internet!







# *HTTP als Kommunikationsprotokoll*



Das weitverbreitete HTTP-Protokoll („Internet = HTTP + Mail“) wird ebenfalls für Dienstanforderungen benutzt.

Wir betrachten Anfragen an *Praktomat*, der eingereichte Praktikumslösungen prüft:

```
POST /praktomat/check HTTP/1.1      - Aufgerufener Dienst
Host: www.praktomat.org             - Zielrechner
Content-Type: text/plain            - Datenformat
Content-Length: 1264                - Länge der Daten

/* Lösung Aufgabe 1 */              - Daten
int main(int argc, char *argv[])
...
```







# *HTTP als Kommunikationsprotokoll (2)*



Der Praktomat-Server gibt das Ergebnis der Prüfung (= des Dienstaufrufs) zurück:

```
200 OK                                  - Status
Content-Type: text/plain                - Datenformat
Content-Length: 34                      - Länge der Daten

Ihre Aufgabe ist bestanden. Danke!      - Daten
```







# *HTTP als Kommunikationsprotokoll (2)*



Der Praktomat-Server gibt das Ergebnis der Prüfung (= des Dienstaufrufs) zurück:

```
200 OK                                  - Status
Content-Type: text/plain                - Datenformat
Content-Length: 34                      - Länge der Daten

Ihre Aufgabe ist bestanden. Danke!      - Daten
```

Einfach zu parsen und zu verarbeiten

Einfach zu realisieren (z.B. als CGI-Skript)

Wird in Millionen von Web-Anwendungen benutzt







# XML



Problem: Anfragen und Ergebnisse müssen *maschinenlesbar* sein ⇒ Trennung von *Inhalt* und dessen *Darstellung*.

Ansatz: *Extended Markup Language* (XML)

- ermöglicht es, beliebige Daten in strukturierter, serialisierter Form zu beschreiben
- ist Text-basiert und einfach zu verarbeiten
- ist ein extrem flexibles und erweiterbares Format







# *Eine Anfrage in XML*



Die folgende Anfrage unterteilt das eingereichte *Programm* (prog1) und den Kommentar (comment):

```
<praktomat_check xmlns="urn:praktomat-org:CheckProcs">
  <prog name="foo.cc">
    int main(int argc, char *argv[]) ...
  </prog>
  <comment
   xmlns='http://www.praktomat.org/documentation'>
    Dies ist ein Kommentar!
  </comment>
</praktomat_check>
```

Das Attribut xmlns legt den Namensraum (*namespace*) fest.







# *Eine Anfrage in XML (2)*



Etwas bequemer geht es mit *Kürzeln* für die Namensräume:

```
<cp:praktomat_check
    xmlns:cp="urn:praktomat-org:CheckProcs"
    xmlns:doc="http://www.praktomat.org/documentation">
  <cp:prog name="foo.cc">
    int main(int argc, char *argv[]) ...
  </cp:prog>
  <doc:comment>
    Dies ist ein Kommentar!
  </doc:comment>
</cp:praktomat_check>
```

# *Schemas*



Schemas legen *Typen* für XML-Datentypen fest:

```
<schema xmlns="http://www.w3.org/1999/XMLSchema'
 targetNamespace="urn:schemas-praktomat-org:CheckProcs'>
  <element name="praktomat_check">
    <type>
      <element name="prog1" type="string" />
      <any minOccurs="0" maxOccurs="*" />
    </type>
  </element>
</schema>
```

Grundtypen: Strings, Bytes, Integers, Zeit/Datum...

Zusammengesetzte Typen: Listen, Unions...

Außerdem *Muster* (reguläre Ausdrücke)







# XML-Ausgaben parsen



XML-Ausgaben können wegen der einheitlichen Syntax sehr einfach weiterverarbeitet werden.

Beispiel – Eine XML-Ausgabe des Praktomat-Servers:

```
<message_list>
    <message>
        <location>
            <file>main.cc</file>
            <line>45</line>
        </location>
        <text>undefined reference</text>
    </message>
</message_list>
```

XML-Parser gibt es für alle gängigen Programmiersprachen







# *Stylesheets*



Um XML-Daten im Browser für Menschen lesbar darzustellen, benötigt man *Stylesheets*, die die Präsentation definieren.

```
location { font-family: "sans-serif",
           font-weight: bold,
           color: #ff0000
         }
text     { font-family: "sans-serif",
            color: #000000
         }
```

stellt Text dar als

**main.cc 45** undefined reference









# *HTML aus XML erzeugen*

Die *XSL Transformation language (XSLT)* ermöglicht es, XML-Elemente in andere XML-Elemente zu wandeln – insbesondere in HTML-Elemente.

Die Wandlung per XSLT kann geschehen

**im Server** – ermöglicht Ausgabe je nach Fähigkeiten des Browsers (bis hin zu WML)

**im Web-Browser** – ermöglicht Austauschbarkeit von XSLT-Spezifikationen und somit individuelle Anpassungen

# *Eine XSLT-Spezifikation*



```
<tt><ul>
<xsl:for-each select="message">
    <li>
    <strong>
    <xsl:value-of select=".//location/file">:
    <xsl:value-of select=".//location/line">:
    </strong>
    <xsl:value-of select=".//text">
    </li>
</xsl:for-each>
</ul></tt>
```

Ergebnis:

- **main.cc:45:**undefined reference







# *Zwischenbilanz*



- Zentrales Prinzip: Trennung von Funktionalität und Präsentation
- Im Web bislang unterentwickelt
- Das HTTP-Protokoll erlaubt den Aufruf von Diensten
- Mit XML läßt sich Inhalt von Darstellung trennen
- Stylesheets und XSLT sorgen für die Darstellung von XML-Inhalten







# *SOAP*



Kommen wir nun zur *maschinellen* Weiterverarbeitung der Daten.

SOAP (*Simple Object Access Protocol*) realisiert ein einfaches Protokoll auf Basis von XML und (typischerweise) HTTP, mit dem externe Dienste aufgerufen werden können.

Im Wesentlichen besteht SOAP aus einer Methode, um Datentypen nach XML und zurück umzuwandeln.



# *SOAP*

Kommen wir nun zur *maschinellen* Weiterverarbeitung der Daten.

SOAP (*Simple Object Access Protocol*) realisiert ein einfaches Protokoll auf Basis von XML und (typischerweise) HTTP, mit dem externe Dienste aufgerufen werden können.

Im Wesentlichen besteht SOAP aus einer Methode, um Datentypen nach XML und zurück umzuwandeln.

SOAP ist eine einfache Möglichkeit, Protokolle wie DCOM, IIOP, RMI zu realisieren

⇒ CORBA, SunONE, .NET usw. können SOAP als Protokoll einsetzen!

# *SOAP-Anfrage*



Der *Header* einer SOAP-Anfrage enthält ein zusätzliches Feld, das den aufgerufenen Dienst spezifiziert.

```
POST /praktomat/check HTTP/1.1      - Aufgerufenes Objekt
Host: www.praktomat.org             - Zielrechner
Content-Type: text/plain            - Datenformat
Content-Length: 1264                - Länge der Daten
SOAPMethodName: urn:praktomat-org:CheckProcs#check
                                    - Aufgerufener Dienst
```

Die Abbildung des SOAP-Methodennamens auf tatsächliche Methoden ist Sache des Servers!

# *SOAP-Anfrage (2)*



Der *Body* der Anfrage enthält die tatsächlichen Daten – in textueller Form.

```
<Envelope>                                  - SOAP-Anfrage
  <Body>
    <cp:check xmlns:cp="urn:praktomat-org:CheckProcs">
      <theProgram>
/* Lösung Aufgabe 1 */                      - Daten
int main(int argc, char *argv[])
...
      </theProgram>
    </cp:check>
  </Body>
</Envelope>
```

# SOAP-Rückmeldung

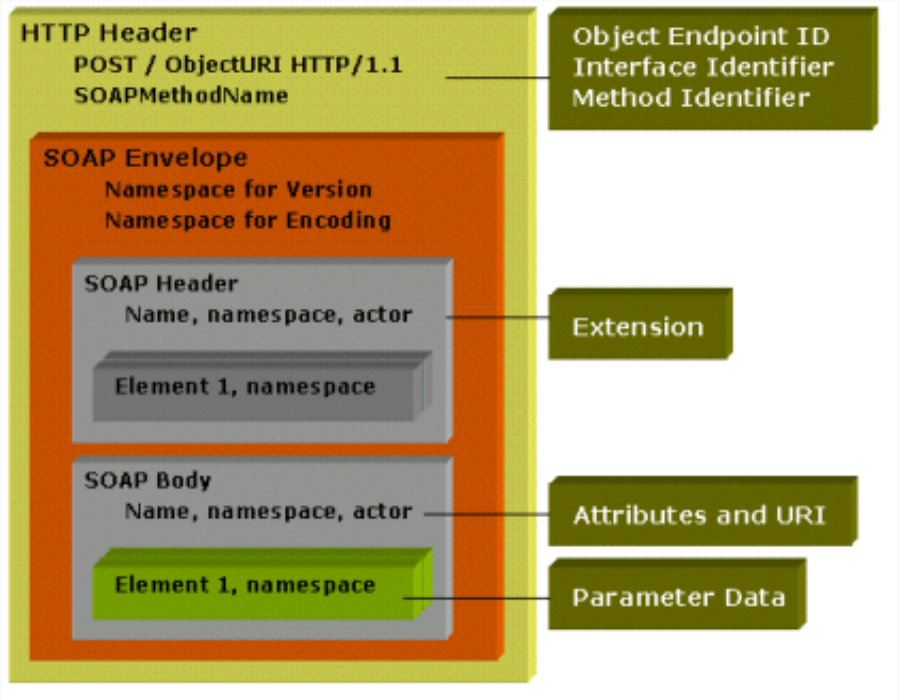


```
200 OK                                  - Status
Content-Type: text/plain                - Datenformat
Content-Length: 234                     - Länge der Daten

<Envelope>
  <Body>
    <cp:checkResponse xmlns:cp="urn:praktomat-org:CheckProcs">
      <message_list>
        <message>
          <location>
            <file>main.cc</file>
            <line>45</line>
          </location>
          <text>undefined reference</text>
        </message>
      </message_list>
    </cp:checkResponse>
  </Body>
</Envelope>
```

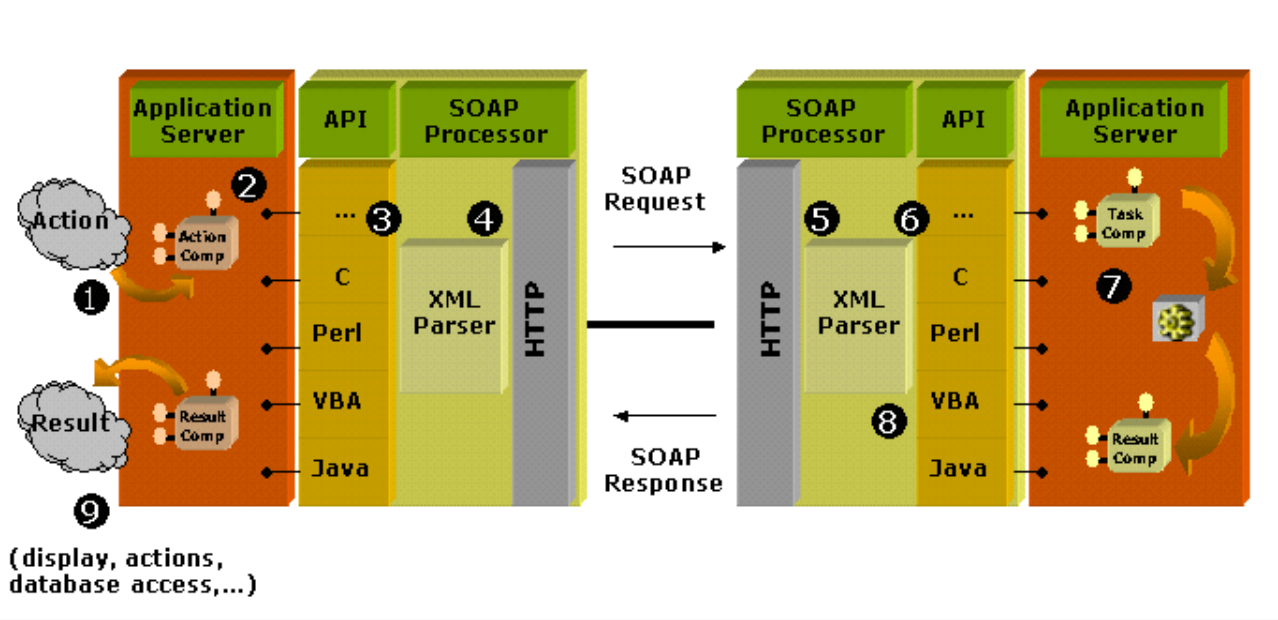
# SOAP-Nachricht: Allgemeine Struktur






Quelle: http://www.intranetjournal.com/articles/200012/pid_12_13_00a.html

# Allgemeines Aufruf-Muster




Quelle: http://www.intranetjournal.com/articles/200012/pid_12_13_00a.html



# *Allgemeines Aufruf-Muster (2)*

1. Ein Dienst des lokalen Application Servers wird aufgerufen
2. Der Application Server nutzt einen verteilten Dienst über dessen API
3. Die API nutzt SOAP, um die Aufrufdaten in XML zu wandeln. . .
4. . . . und an den Web-Server zu senden
5. Dessen SOAP-Komponente entpackt die Aufrufdaten. . .
6. und stellt sicher, daß alle relevanten Teile vorhanden sind.
7. Anschließend wird der Dienst aufgerufen und der Rückgabewert bestimmt
8. Der Rückgabewert wird genauso zurückgesandt.
9. Bei Dienstende geht die Kontrolle an den ursprünglichen Aufrufer zurück.









# *Eine Bankanwendung mit SOAP*

Wir betrachten eine Schnittstelle für eine Bankanwendung, wie sie z.B. in CORBA IDL spezifiziert sein könnte:

```
struct withdrawData {          - Daten zum Abheben
    long account;
    float amount;
};
struct withdrawResponse {      - Rückmeldung
    float newBalance;
    float amount;
    boolean overdrawn;
};
interface IBank {
    withdrawResponse withdraw(withdrawData data);
};
```







# *Bankanwendung: Aufruf des Dienstes*



(hier mit allen Namespaces)

```
<soap:Envelope
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'>
  <soap:Body>
    <IBank:withdrawData xmlns:IBank=
     'urn:uuid:DEADF00D-BEAD-BEAD-BEAD-BAABAABAABAA'>
      <account>3512</account>
      <amount>100</amount>
    </IBank:withdrawData>
  </soap:Body>
</soap:Envelope>
```

Methodenname und URI werden separat spezifiziert!

# *Bankanwendung: Rückmeldung*



Die Rückmeldung erfolgt ebenfalls im SOAP-Format

```
<soap:Envelope
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'>
  <soap:Body>
    <IBank:withdrawResponse xmlns:IBank=
     'urn:uuid:DEADFOOD-BEAD-BEAD-BEAD-BAABAABAABAA'>
      <newBalance>0</newBalance>
      <amount>5</amount>
      <overdrawn>true</overdrawn>
    </IBank:withdrawResponse>
  </soap:Body>
</soap:Envelope>
```







# *Serialisierung – Allgemeine Form*



Die allgemeine Form für serialisierte Daten sieht so aus:

```
<t:typname xmlns:t='namespaceuri'>
<feldname1>feld1wert</feldname1>
<feldname2>feld2wert</feldname2>
...
</t:typname>
```

Diese Form wird auch *Element-Normal-Form* (ENF) genannt.









# *WSDL*

WSDL (*Web Services Description Language*) beschreibt die Schnittstelle für SOAP-Dienste

Einfacher Standard

Ermöglicht die *Selbstbeschreibung* von Web-Diensten:

- Welche Dienste stelle ich zur Verfügung?
- Wie sieht die Schnittstelle aus?
- Welches Protokoll kann ich benutzen?, usw.







# *Schnittstelle der Bankanwendung*



Wir betrachten zunächst die Schnittstelle der Nachrichten:

```
<element name="withdrawData">
  <complexType>
    <sequence>
      <element name="account" type="long"/>
      <element name="amount"  type="float"/>
    </sequence>
  </complexType>
</element>
```







# *Schnittstelle der Bankanwendung (2)*



Der Rückgabewert wird analog spezifiziert:

```
<element name="withdrawResponse">
  <complexType>
    <sequence>
      <element name="newBalance" type="float"/>
      <element name="amount"     type="float"/>
      <element name="overdrawn"  type="boolean"/>
    </sequence>
  </complexType>
</element>
```







# *Schnittstelle der Bankanwendung (2)*



Die Methodenbeschreibungen kommen in einen *Rahmen*.

Der Rahmen beschreibt, mit welchen Nachrichten über welche Protokolle die Dienste aufgerufen werden können.

```
<definitions name="IBank">
  <types>
    <schema xmlns="http://www.w3.org/1999/XMLSchema">
       - (Nachrichten-Schnittstellen) -
    </schema>
  </types>
  ...          - Messages, Ports, Services
</definitions>
```

In der Praxis werden solche Beschreibungen aus einer IDL generiert









# *WSDL in der Praxis*

Will ein Programmierer einen Web-Dienst nutzen, so . . .

1. Startet er eine WSDL-Anfrage beim Web-Dienst
2. läßt er die WSDL-Beschreibung in eine IDL umwandeln
3. läßt er die IDL in einen Stub umwandeln
4. kann er den Web-Dienst über den Stub so nutzen, als wäre er lokal verfügbar.

All diese Schritte lassen sich automatisieren
⇒ Zugang zu Web-Diensten ist nicht schwieriger als die Nutzung einer Bibliothek









# *Fazit*

Web-Dienste realisieren verteilte Dienste mit *Bordmitteln:*

Sie kombinieren

- bekannte Techniken (Serialisierung, entfernter Dienstaufruf)
- mit bekannten Protokollen (HTTP),

um verteilte Dienste leicht realisierbar zu machen.

Vereinheitlichende Grundlage ist XML









# *Fazit*

Web-Dienste realisieren verteilte Dienste mit *Bordmitteln:*

Sie kombinieren

- bekannte Techniken (Serialisierung, entfernter Dienstaufruf)
- mit bekannten Protokollen (HTTP),

um verteilte Dienste leicht realisierbar zu machen.

Vereinheitlichende Grundlage ist XML

Weder elegant noch tiefsinnig, aber einfach (wie z.B. HTML)

Ziel ist Nutzen durch breite Anwendung und Standardisierung









# *Zusammenfassung*

- *Funktionalität* und *Präsentation* gehören getrennt
- In Dokumenten wird der Inhalt durch *XML* ausgedrückt
- Die Präsentation erfolgt über *Annotation* (Stylesheets) und *Transformation* (XSLT)
- *Web Services* wie SOAP machen Funktionalität über Standard-Protokolle (HTTP) verfügbar
- SOAP dient als Basis für beliebige verteilte Anwendungen
- Mit WSDL können Anbieter ihre Dienste beschreiben









# *Literatur*

**World Wide Web Consortium** (http://www.w3.org) – alle hier beschriebenen Standards und Tutorials

**A Young Person's Guide to The Simple Object Access Protocol** (http://msdn.microsoft.com/msdnmag/issues/0300/soap/soap.asp) – Einführung in SOAP