Einführung in XML Web Services

Christoph Hartmann, Martin Sprengel, Michael Perscheid, Gregor Gabrysiak und Falko Menge

Seminar Konzepte und Methoden der Web-Programmierung WS 05/06
Hasso-Plattner-Institut für Softwaresystemtechnik

{christoph.hartmann, martin.sprengel, michael.perscheid,
gregor.gabrysiak, falko.menge}@hpi.uni-potsdam.de

Diese Arbeit beschreibt ausführlich was Web Services sind und wie man sie realisieren kann. Es werden die drei wichtigsten Standards für XML-basierte Web Services ausführlich erklärt: SOAP ist dabei das am häufigsten verwendete Kommunikationsprotokoll für den Zugriff auf entfernte Dienste. Mit der Web Services Description Language (WSDL) werden vertragsähnliche Beschreibungsdokumente erstellt, die vom Server für einen Client angeboten werden können. Universal Description, Discovery and Integration (UDDI) ist der Standard für Namensdienste, mit denen Web Services bekannt gemacht und gesucht werden können.

Als Hilfe für Designentscheidungen bei der Realisierung werden die besonderen Eigenschaften von Web Services erläutert. An praktischen Beispielen mit PHP5 wird sowohl das Anbieten und Nutzen von Web Services als auch der Zugriff auf UDDI-Verzeichnisse demonstriert. Am Ende folgen eine Auseinandersetzung mit einigen Schwierigkeiten der aktuellen Web Service Technologie und ein Ausblick auf künftige Entwicklungen.

Stichworte: Web Services, SOAP, WSDL, UDDI, PHP5

1. Einführung

Web Services sind Dienste, die von Webservern angeboten werden. Im Gegensatz zu Internetseiten für menschliche Leser werden diese Dienste durch andere Softwaresysteme genutzt. Eine angebotene Dienstleistung kann beispielsweise darin bestehen, Informationen bereitzustellen oder Berechnungen durchzuführen. Interessant ist diese Technologie deshalb, weil die Dienste selbst beschreibend und in sich geschlossen sind und sich über das Internet publizieren, lokalisieren und aufrufen lassen. Damit werden viele interessante Geschäftsmodelle möglich, beispielsweise Modelle, bei denen ein Nutzer für jede einzelne Nutzung einer Softwarekomponente und der bereitgestellten Ressourcen zahlt. Dabei ist ein Nutzer nicht an eine bestimmte Plattform oder Programmiersprache gebunden, denn Web Services sind weitgehend unabhängig von anderen Technologien. Durch diese Unabhängigkeit lassen sich Web Services sehr gut einsetzen um heterogene Systeme miteinander zu verbinden.

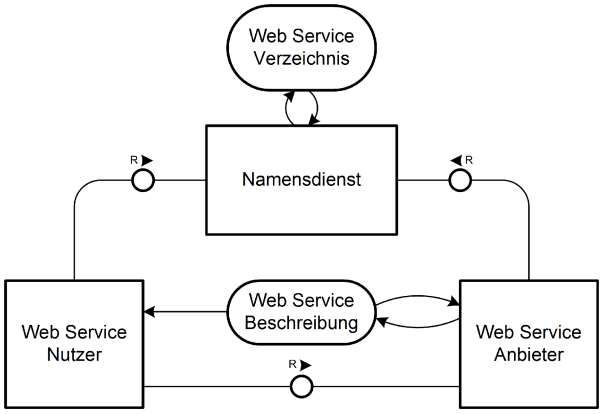

Abbildung 1: allgemeines Web Service Szenario

Ein typisches Szenario für die Nutzung eines Web Service besteht aus Anbietern, Nutzern und möglicherweise auch einem Namensdienst. Um einen Web Service zu veröffentlichen muss ein Anbieter zusätzlich zur Entwicklung der Softwarekomponenten für die eigentliche Diensterbringung eine Web Service Beschreibung erstellen. Sie enthält Informationen über die angebotenen Schnittstellen und das zu verwendende Kommunikationsprotokoll. Anschließend kann der Anbieter nun den Web Service bei einem Namensdienst registrieren. Potentielle Nutzer können dann den Namensdienst abfragen, um passende Dienste für ihre Anwendung zu finden. Ein Nutzer bekommt vom Namensdienst die genaue Adresse, unter welcher der Web Service erreichbar ist. Dort kann er vom Anbieter die Beschreibung anfordern, um zu erfahren, wie der Dienst zu verwenden ist. Mit diesen Informationen ist der Nutzer in der Lage, über

ein Kommunikationsprotokoll Anfragen an die Schnittstellen des Anbieters zu senden und die Dienstleistungen zu nutzen. Die Web Service Beschreibung wirkt wie ein Vertrag zwischen Anbieter und Nutzer: Hält einer der beiden Partner sich nicht an die Beschreibung, kommt keine sinnvolle Kommunikation zustande.

In den folgenden drei Abschnitten werden die wichtigsten Standards vorgestellt, mit denen dieses Szenario zwischen Softwaresystemen realisiert wird. Mit SOAP wird das wichtigste Kommunikationsprotokoll für Web Services erklärt. Als Standard für die Beschreibung von Web Services wird WSDL vorgestellt. Der Abschnitt 4 beschäftigt sich mit dem Namensdienst UDDI. Nachdem diese Standards behandelt wurden, soll Abschnitt 5 soll durch die Beschreibung der besonderen Eigenschaften eines Web Services bei Enwurfsentscheidungen helfen und zur praktischen Umsetzung hinführen. In Abschnitt 6 wird dann die Implementierung von Web Services an praktischen Beispielen in PHP5 erläutert. Am Ende werden Schwierigkeiten und Probleme der aktuellen Technologie aufgezeigt und kurz weitere Entwicklungen vorgestellt.

2. SOAP

2.1. SOAP Überblick

SOAP bildet den Eckpfeiler der plattformunabhängigen Kommunikation von Web Services. Es ist ein XML-basiertes Kommunikationsprotokoll, das zum Austausch von strukturierten und typisierten Daten zwischen Computern in einer dezentralisierten, verteilten Umgebung genutzt werden kann. Dabei findet SOAP hauptsächlich in den Bereichen Electronic Data Interchange (EDI), Remote Procedure Call (RPC) und Business-to-Business-Kommunikation (B2B) seine Anwendung. Zur Zeit liegt SOAP in der aktuellen Version SOAP 1.2 als Recommendation des W3C vor [SOAP].

2.2. Struktur einer SOAP Nachricht

2.2.1. Allgemeiner Aufbau

Eine SOAP Nachricht unterliegt zunächst einer vorgegebenen Syntax. Dabei wird als Basiselement ein Element des Typs `<Envelope>` aus dem Namensraum *http://www.w3.org/2003/05/soap-envelope* definiert, der die Hülle einer SOAP Nachricht repräsentiert. In dieses Element eingebettet sind das optionale `<Header>`- sowie das notwendige `<Body>`-Element. Des weiteren bietet SOAP die Möglichkeit, wie bei einer E-Mail einen oder mehrere Anhänge an die SOAP Nachricht anzuhängen. Dies wird gesondert in der W3C-Note „SOAP messages with attachments“ [ATTACH] behandelt.

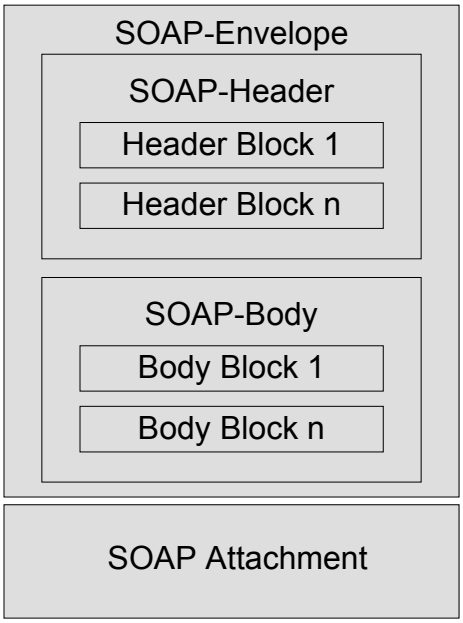

Abbildung 2: Blockdiagramm der Struktur einer SOAP-Nachricht

```
<env:Envelope xmlns:env=
   “http://www.w3.org/2003/06/soap-envelope”>
  <env:Header>
    …
  </env:Header>
  <env:Body>
    …
  </env:Body>
</env:Envelope>
```

Abbildung 3: Struktur einer SOAP-Nachricht

2.2.2. Der SOAP-Header

Das `<Header>`-Element, welches den Kopf einer SOAP Nachricht darstellt, kann weitere Elemente beinhalten, die als `<Header>`-Blöcke bezeichnet werden. Hierbei werden meist Informationen untergebracht, die für den Empfänger einer SOAP Nachricht bezüglich der weiteren Verarbeitung wichtig sind. Im Verlauf der Verarbeitung einer SOAP Nachricht entlang eines Nachrichtenpfades (Message Path) können `<Header>`-Blöcke verändert und entfernt werden. Um die Verarbeitung von SOAP

Nachrichten durch Zwischenknoten und Empfänger zu präzisieren, werden durch SOAP folgende Attribute für <Header>-Elemente vorgegeben.

- *role* gibt an, welche Knoten für die Verarbeitung des <Header>-Blocks zuständig sind.
- *encodingStyle* definiert Datentypen, die für diesen <Header>-Block verwendet werden können und gibt einen Satz von Regeln an, der zur Serialisierung bzw. Deserialisierung der SOAP Nachricht verwendet werden kann.
  Bsp.: *http://www.w3.org/2003/05/soap-encoding*
- *mustUnderstand ist* ein boolescher Wert, der angibt, ob der <Header>-Block vom Empfänger verarbeitet werden muss oder ignoriert werden kann
- *relay* ist wiederum boolescher Wert, der angibt, ob ein <Header>-Block von einem Knoten weitergegeben werden soll, falls dieser ihn nicht verarbeiten konnte.

```
<env:Header>
  <p:block1 xmlns:p="http://www.example.org"
     env:role=
"http://www.w3.org/2003/05/soap-envelope/role/next"
     env:mustUnderstand="false"
     env:relay="true">
    ...
  </p:block1>
  <q:block2 xmlns:q="http://www.example.org"
     env:role="http://www.w3.org/2003/05/soap-
     envelope/role/ultimateReceiver"
     env:mustUnderstand="true">
    ...
  </q:block2>
  <r:block3 xmlns:r="http://www.example.org"
     env:encodingStyle=
     "http://www.example.org/encoding">
    ...
  </r:block3>
</env:Header>
```

Abbildung 4: Ein beispielhafter SOAP-Header mit mehreren Header-Elementen.

2.2.3. Der SOAP-Body

Das <Body>-Element kann weitere <Body>-Blöcke beinhalten, welche die eigentlichen Anwendungsdaten enthalten. Diese Informationen sind für den Empfänger am Ende eines Nachrichtenpfades gedacht.

2.2.4. SOAP mit Anhängen

SOAP ist hervorragend geeignet für den Umgang mit strukturierten und typisierten Daten. Einige Anwendungen arbeiten allerdings nicht nur mit dieser Art von Daten, sondern auch mit Binärdaten, wie zum Beispiel Bildern. Bei SOAP mit Anhängen wird nun das SOAP Protokoll mit dem MIME Standard kombiniert.

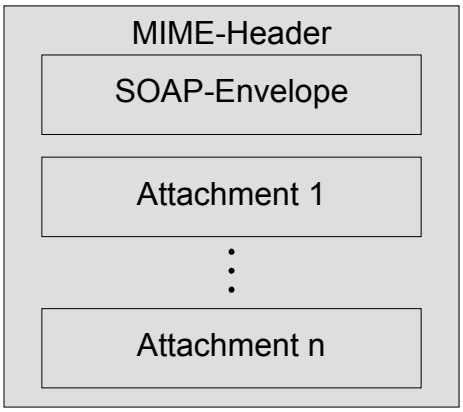

Abbildung 5: Struktur einer SOAP-Nachricht

Dabei wird ein MIME-Header erzeugt, der an das unterliegende Protokoll gebunden wird. Die SOAP Nachricht ist nun der erste Teil des MIME-Headers, alle anderen Anhänge reihen sich dahinter ein. Im MIME Header ist genau definiert, wo jeder Datenblock beginnt und endet.

2.3. Verarbeitung von SOAP Nachrichten

Am einfachsten Szenario der Verarbeitung von SOAP Nachrichten sind zunächst zwei Knoten beteiligt: der SOAP-Client und der SOAP-Server.

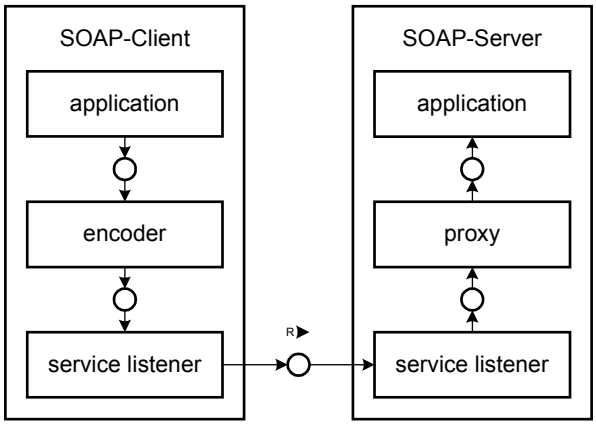

Abbildung 6: an der Verarbeitung einer SOAP-Nachricht beteiligte Komponenten

Auf der Seite des SOAP-Client stellt die Anwendung zunächst die strukturierten Daten zur Verfügung. Diese Daten werden mittels eines

Encoders in einer SOAP Nachricht eingebettet, durch einen Sender an ein spezielles Protokoll gebunden und anschließend an den SOAP-Server übermittelt. Der SOAP-Server seinerseits hat einen oder mehrere Listener, die dafür zuständig sind, die Anfrage entgegen zu nehmen und die darin enthaltene SOAP Nachricht zu extrahieren. Die SOAP Nachricht wird dann an einem Proxy weitergereicht, welcher die enthaltenen Daten einer Anwendung zur Verfügung stellt.

2.3.1. Message Path

Der SOAP-Standard spezifiziert ein eigenes Verarbeitungsmodell. An der Kommunikation ist ein SOAP-Sender, ein SOAP-Empfänger sowie optional ein oder mehrere SOAP-Zwischenknoten (*Intermediaries*) beteiligt. Diesen Weg, vom Sender zum eigentlichen Empfänger der Nachricht, nennt man Nachrichtenpfad. Dabei wird aber nicht festgelegt, wie ein Zwischenknoten erreicht werden kann, sondern nur welcher Teil des SOAP-Nachricht für ihn bestimmt ist und wie dieser verarbeitet werden soll. Im Allgemeinen ist der SOAP-Body für den sogenannten *ultimateReceiver* bestimmt, also den Empfänger am Ende der Kette. Der SOAP-Header soll dagegen von den Zwischenknoten und dem SOAP-Receiver verarbeitet werden. Physikalisch können dabei auch mehrere Zwischenknoten auf dem gleichen Rechner arbeiten und dort eine sogenannte *Handler Chain* bilden.

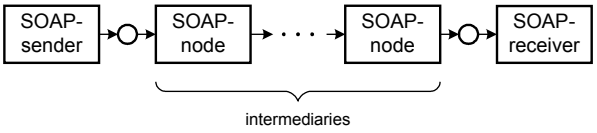

Abbildung 7: Ein Nachrichtenpfad mit mehreren Zwischenknoten

2.4. Austausch von SOAP Nachrichten

Die Ursprünge von SOAP liegen in synchronen Remote Procedure Calls über HTTP (XML-RPC), worauf man aus der aktuellen Spezifikation kaum noch schließen kann.

2.4.1. Message-based Document Exchange Model

Grundlegend werden SOAP Nachrichten einmalig asynchron von einem Sender zu einem Empfänger übertragen. Dies bezeichnet man auch als *message-based Document Exchange Model*.

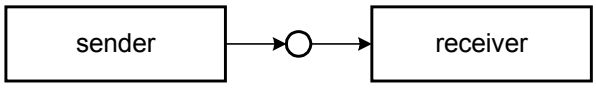

Abbildung 8: grundlegende Übertragungsart von SOAP-Nachrichten (one-way-transmissions)

2.4.2. Remote Procedure Calls (RPC)

Wie schon erwähnt, findet SOAP Anwendung im Bereich der RPCs. Ein RPC ist durch SOAP als ein Spezialfall des Message-based Document Exchange Models definiert, denn ein RPC stellt nichts anderes dar, als eine Aufeinanderfolge von mehreren asynchronen einmalig zugestellten Nachrichten. Bei SOAP wird dies auch als Frage/Antwort-Muster (request/response-pattern) bezeichnet. Allerdings wird hier nur der grobe Ablauf festgelegt und es ist den SOAP-Implementierungen überlassen die genaue Syntax und Semantik festzulegen.

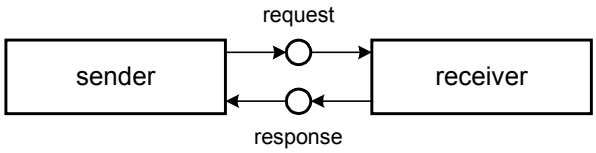

Abbildung 9: Übertragungsmuster (MEP, request/response-pattern)

Der eigentliche RPC wird nun im Rumpf der SOAP-Nachricht gekapselt. Dazu wird zunächst der Namensraum der entsprechenden Methode eingebunden. Sofern Parameter vorhanden sind, werden diese strukturiert und optional typisiert als Kindelemente des Methoden-Elementes angegeben. Die Nachricht wird dann an den SOAP-Server übermittelt.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV=
      "http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:ns1=
      "http://www.tele-task.de/LectureWebService.wsdl"
   xmlns:xsd=
      "http://www.w3.org/2001/XMLSchema"
   xmlns:xsi=
      "http://www.w3.org/2001/XMLSchema-instance"
   SOAP-ENV:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <ns1:getLectures>
     <ns1:count xsi:type="xsd:Integer”>3</ns1:count>
    </ns1:getLectures>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Abbildung 10: Beispiel für eine SOAP-Anfrage (RPC)

In der Antwort werden die Ergebnisse wiederum strukturiert und optional typisiert in einem Methoden-Response-Element angegeben.

Dabei wird zumeist das Wort „Response“ an den Methodennamen gehangen um zu kennzeichnen, dass es sich um die Antwort eines RPCs handelt. Es ist nicht zwingend notwendig sich an diese Konvention zu halten, wird jedoch empfohlen. Werden mehrere Ergebnisse zurückgegeben, so gibt es die Möglichkeit den Namensraum *http://www.w3.org/2003/05/soap-rpc* einzubinden und mit Hilfe des *rpc:result* Elementes den Haupt-Rückgabewert der Funktion zu kennzeichnen.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV=
    "http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:ns1=
    "http://www.tele-task.de/LectureWebService.wsdl"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi=
    "http://www.w3.org/2001/XMLSchema-instance"
   xmlns:SOAP-ENC=
    "http://schemas.xmlsoap.org/soap/encoding/"
   SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <ns1:getLecturesResponse
       xmlns:rpc=
       "http://www.w3.org/2003/05/soap-rpc">
      <rpc:result>ns1:return</rpc:result>
      <ns1:return
         SOAP-ENC:arrayType="xsd:string[5]"
         xsi:type="ns1:LectureListType">
        <item xsi:type="xsd:string">
          Verschluesselung (Prof. Meinel - 00:48:12)
        </item>
        <item xsi:type="xsd:string">
          Digitale Signaturen (Prof. Meinel -
          01:34:05)
        </item>
        <item xsi:type="xsd:string">
          Authentifikation (Prof. Meinel – 01:26:23)
        </item>
      </ns1:return>
    </ns1:getLecturesResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Abbildung 11: Beispiel für eine SOAP-Antwort (RPC)

2.5. SOAP Fehlerbehandlung

Bei der Verarbeitung einer SOAP Nachricht können natürlich auch Fehler auftreten. SOAP bietet hier das `<Fault>`-Element (Kindelement des `<Body>`-Elements), das zur Signalisierung eines aufgetretenen Fehlers genutzt werden kann. Das `<Fault>`-Element wiederum hat fünf Kindelemente.

- `<Code>` hat ein `<Value>`-Kindelement, das den Fehlercode angibt und ein `<Subcode>`-Kindelement mit entsprechendem `<Value>`-Element, das eine anwendungsspezifische Subkategorie des Fehlercodes angibt.
- `<Reason>` hat ein oder mehrere *text*-Kindelemente, die eine nähere Beschreibung des Grundes für den Fehler geben.
- `<Node>` identifiziert den Knoten, bei dem der Fehler aufgetreten ist.
- `<Role>` identifiziert die Rolle, in der sich der Knoten bei Auftreten des Fehlers befand.
- `<Detail>` beschreibt den Fehler im Detail.

SOAP definiert fünf Fehlercodes die bei der Verarbeitung einer SOAP Nachricht auftreten können.

- *VersionMismatch*: Der Namensraum des `<Envelope>`-Elements stimmt nicht mit dem des verarbeitenden Knotens überein.
- *MustUnderstand*: Ein Knoten kann einen Block nicht verarbeiten, den er jedoch in jedem Fall verarbeiten sollte.
- *DataEncodingUnknown*: Es wurde eine Kodierung angegeben, die der Client nicht unterstützt.
- *Sender*: Die Nachricht wurde falsch zusammengesetzt und reicht dem Receiver nicht zur Verarbeitung.
- *Receiver:* Der Inhalt kann vom Receiver nicht verarbeitet werden.

2.6. SOAP Protokoll-Bindung

Einer der großen Pluspunkte des SOAP Protokolls ist die Unabhängigkeit vom darunter liegenden Transportprotokoll. Das heißt die SOAP Nachricht wird einfach in eine Nachricht des darunter liegenden Protokolls eingekapselt.

Es gibt vor allem zwei Protokollbindungen, die hauptsächlich genutzt werden und die hier näher erläutert werden sollen. Es handelt sich dabei zum einen um die HTTP Protokoll-Bindung und um die SMTP Protokoll-Bindung.

2.6.1. HTTP Protokoll-Bindung

Während in SOAP 1.1 noch ein spezielles Headerfeld, der SOAPAction Header im HTTP Protokoll, notwendig war um den Inhalt der SOAP Nachricht nach außen zu tragen, ist in der aktuellen Version SOAP 1.2 ein neuer Medientyp *application/soap+xml* eingeführt worden, der als Content-Type angegeben wird.

```
GET /www.example.org/example HTTP/1.1
Host: www.example.org
Accept: text/html; application/soap+xml
```

Abbildung 12: HTTP-GET Anfrage

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env=
   “http://www.w3.org/2003/06/soap-envelope”>
  ...
</env:Envelope>
```

Abbildung 13: SOAP-Nachricht in einer HTTP-GET Antwort eingebettet

```
POST /example HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml
Content-Length: nnnn

<?xml version='1.0' ?>
<env:Envelope xmlns:env=
   “http://www.w3.org/2003/06/soap-envelope”>
  ...
</env:Envelope>
```

Abbildung 14: SOAP-Nachricht in einer HTTP-POST Anfrage eingebettet

2.6.2. SMTP Protokoll-Bindung

Eine weitere Möglichkeit ist die Bindung einer SOAP Nachricht an das SMTP Protokoll. Diese Art der Bindung ist jedoch nicht durch die SOAP 1.2 Spezifikation beschrieben, sondern durch eine W3C Note (SOAP Email Binding). Obwohl die Bindung ähnlich der Bindung an das HTTP Protokoll ist, hat sie einen entscheidenden Nachteil. Es kann nicht garantiert werden, das eine SOAP Nachricht beim Empfänger angekommen ist. Es gibt zwar die Möglichkeit per Delivery Status Notification (DSN) bzw. Message Disposition Notification (MDN) den Empfang zu quittieren, jedoch sind dies nur Benachrichtigungen auf der STMP-Ebene und sie werden losgelöst von der ursprünglichen Nachricht gesendet. Es ist also Vorsicht geboten, wenn man daraus auf SOAP-Ebene Aussagen über ausgelieferte Daten treffen möchte.

```
From: sender@example.org
To: receiver@example.org
Subject: example
Date: Thu, 29 Nov 2001 13:20:00 EST
Message-Id: <exampleID001@example.org>
Content-Type: application/soap+xml

<?xml version='1.0' ?>
<env:Envelope xmlns:env=
   “http://www.w3.org/2003/06/soap-envelope”>
  ...
</env:Envelope>
```

Abbildung 15: SOAP-Nachricht in eine E-Mail eingebettet

Die SOAP Nachricht wird bei dieser Bindung im Email-Body eingebettet und Informationen zum Inhalt der Email-Nachricht werden im Email-Header gesetzt, so zum Beispiel eine Transaktionsnummer.

3. WSDL

3.1. Was ist WSDL?

WSDL steht für Web Services Description Language und ist eine XML-basierte Beschreibungssprache. Grundsätzlich kann man WSDL als Schnittstellenbeschreibung von im Internet angebotenen Diensten ansehen. Diese von Plattformen, Programmiersprachen und Protokollen unabhängige Definition erlaubt das Definieren von abstrakten Operationen und deren Parametern. Diese werden in der Beschreibung an ein konkretes Protokoll gebunden. Dabei wird in WSDL zwischen dokumenten-orientierten und prozedur-orientierten Diensten unterschieden. Das sind also wie eingangs beschrieben Dienste, die Informationen bereitstellen oder Berechnungen durchführen.

Obwohl die Sprache erweiterbar konzipiert und eben grundsätzlich unabhängig vom verwendeten Kommunikationsprotokoll ist, wird WSDL heute meistens zusammen mit SOAP verwendet.

3.2. Motivation für WSDL

In den letzten Jahren sind viele XML-Protokolle zum Kommunizieren über das Internet entstanden. Die bekanntesten Ansätze dafür waren:

- *Web Distributed Data eXchange* (WDDX)
- *XML Remote Procedure Call* (XML-RPC)
- *Simple Object Access Protocol* (SOAP)

Nachdem sich diese Protokolle entwickelt hatten, wurde es immer wichtiger eine Schnittstellenbeschreibung in einem wohldefinierten Format nach außen anzubieten. Dabei ist WSDL nicht die erste Lösung. Vorläufer von WSDL waren:

- WebMethods *Web Interface Definition Language* (WIDL) gilt als Pionier. Sie wird meist mit dem XML-RPC System verwendet.
- Die *SOAP Contract Language* (SCL) wurde

von Microsoft entwickelt und für das eigene Kommunikationsprotokoll als Beschreibungssprache zur Verfügung gestellt.
- IBMs *Network Accessible Service Specification Language* (NASSL) wurde zur gleichen Zeit wie Microsofts SCL erarbeitet.

Bevor WSDL entwickelt wurde, arbeiteten 36 Unternehmen, darunter IBM, Ariba und Microsoft, an dem Universal Description, Discovery and Integration (UDDI) System. Die Entwicklung eines Verzeichnisdienstes war von Nöten, um die möglicherweise millionenfach angebotenen Web Services lokalisieren zu können. Die Umsetzung sollte eine klare API aufweisen und hauptsächlich für sogenannte Business Applications Verwendung finden.

Genau diese Unternehmen haben sich später zusammengetan, um der Vielfalt an Schnittstellen ein Ende zu setzen. Sie haben dabei ihre eigenen Ideen und Entwicklungen in der Web Services Description Language (WSDL) Version 1.0 vereint.

3.3. Einführung in WSDL

WSDL bietet folgende Elemente an um Web Services zu beschreiben:

- `<types>` sind Container für Datentypen. Diese können lokal definiert oder separat in eigenen XSD Dateien abgelegt werden.
- Eine `<message>` stellt eine abstrakte Typdefinition über die zu kommunizierenden Daten dar. Diese Nachrichtentypen sind zu vergleichen mit der Deklaration der Parameterliste oder des Rückgabewertes einer Funktion.
- Ein `<portType>` definiert eine Menge von Operationen, welche zusammenhängend angeboten werden.
- Jede `<operation>` legt Nachrichtentypen für Eingabe-, Ausgabe- und Fehlernachrichten fest.
- Das `<binding>`-Element spezifiziert zu den Operationen und Nachrichten eines `<portType>` das Kommunikationsprotokoll und das Format, in dem die Nachrichten kodiert werden sollen.
- `<port>` enthalten verschiedene Endpunkte, welche sich aus einer Bindung und einer Netzwerkadresse ergeben .
- Ein `<service>` ist eine Sammlung von mehreren Ports.

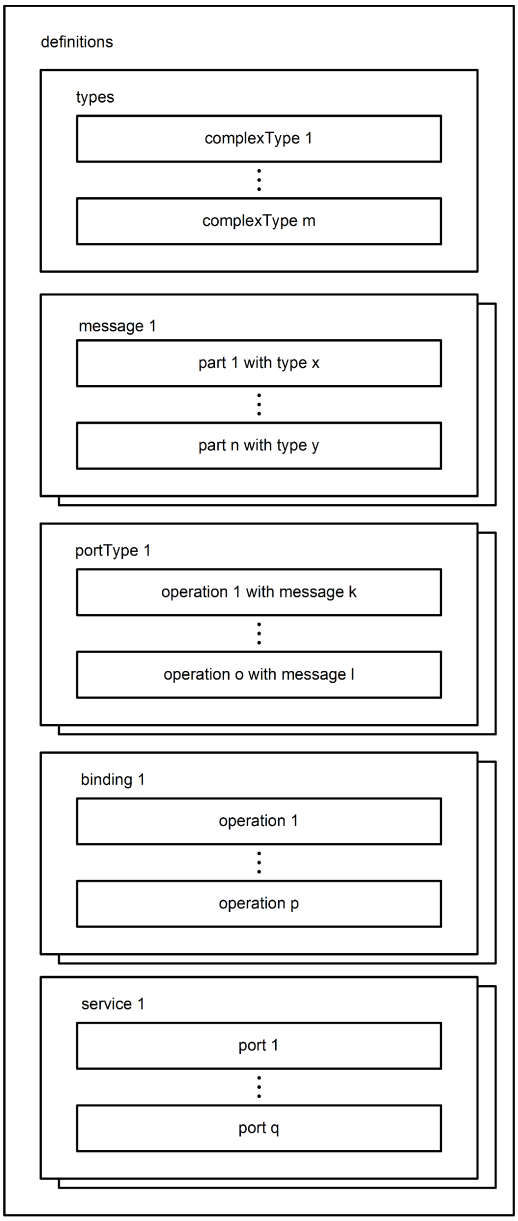

Abbildung 16: Typischer Aufbau einer WSDL Beschreibung

Das Diagramm in Abbildung 16 soll die Struktur verdeutlichen. Die Anzahl muss nicht bei allen Abschnitten gleich sein. Es kann daher zu n Nachrichten (Messages) auch m Typdefinitionen geben. Dies gilt für alle Daten, die mehrfach verwendet werden können.

3.4. WSDL Dokumentenstruktur

Zur Beschreibung des Dokuments wird nicht der klassische Top-Down-Ansatz verwendet. Bei WSDL empfiehlt es sich zum besseren Verständnis des Dokumentes einen Bottom-Up-

Ansatz zur Analyse durchzuführen. Dabei wird natürlich von der typischen Definitionsart eines WSDL-Dokumentes ausgegangen, in welcher erst die Typen, Messages, PortTypes & Bindings definiert werden und zum Schluss erst die Servicedefinition. Natürlich steht es dem Benutzer frei, die jeweiligen Tags in einer anderen Reihenfolge anzuwenden. Hier wird vom Standardfall ausgegangen, wie es auch vom W3C vorgeschlagen wird.

Im folgendem Beispiel geht es um einen Teleteaching-Anbieter, der Live-Übertragungen von Lehrveranstaltungen über das Internet anbietet. Um möglichst weite Verbreitung zu erreichen, möchte er Informationen über die angebotenen Veranstaltungen anderen Anbietern zugänglich machen. Ein zugreifender Anbieter kann dann beispielsweise sein Content Management System mit dem Web Service des Teleteaching-Anbieters verbinden und somit die Informationsquelle direkt in sein Angebot integrieren.

3.4.1. Dokumentenrumpf

Da es sich um ein XML-Dokument handelt muss als erstes die XML-Definition erscheinen. Direkt darauf folgt das Tag `<wsdl:definitions>`. Innerhalb dieses Tags wird die gesamte Schnittstelle definiert.

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions
  name="LectureWebService"
  targetNamespace=
    "http://www.tele-task.de/LectureWebService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns=
    "http://www.tele-task.de/LectureWebService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC=
    "http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:ns="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:ns1="http://schemas.xmlsoap.org/wsdl/mime/"
>

<!—weitere Definitionen erscheinen hier -->

</wsdl:definitions>
```

Abbildung 17: Rumpf der WSDL Beschreibung

Es soll hier noch kurz auf die Namensräume eingegangen werden. Der wohl wichtigste Namensraum ist der von WSDL selbst: *http://schemas.xmlsoap.org/wsdl/*. Nur dadurch ist dem späteren Interpreter dieser Beschreibung bekannt, wie das Dokument aufgebaut ist. Als `targetNamespace` ist das eigene Dokument einzubinden.

Präfix **:**	**wsdl**
Namensraum:	http://schemas.xmlsoap.org/wsdl/
Beschreibung:	WSDL Namensraum
Präfix**:**	**soap**
Namesraum:	http://schemas.xmlsoap.org/wsdl/soap/
Beschreibung:	WSDL Namensraum für SOAP Bindung
Präfix**:**	**http**
Namensraum:	http://schemas.xmlsoap.org/wsdl/http/
Beschreibung:	WSDL Namensraum für HTTP GET und POST Bindungen
Präfix**:**	**mime**
Namensraum:	http://schemas.xmlsoap.org/wsdl/mime/
Beschreibung:	WSDL Namensräume für MIME Bindungen
Präfix**:**	**soapenc**
Namensraum**:**	http://schemas.xmlsoap.org/soap/encoding/
Beschreibung:	Namensraum für die SOAP Verschlüsselung
Präfix**:**	**soapenv**
Namensraum:	http://schemas.xmlsoap.org/soap/envelope/
Beschreibung:	Namensraum für die SAOP Einhüllung
Präfix**:**	**xsi**
Namensraum:	http://www.w3.org/2000/10/XMLSchema-instance
Beschreibung:	Exemplarischer Namensraum für XSD
Präfix**:**	**xsd**
Namensraum:	http://www.w3.org/2000/10/XMLSchema
Beschreibung:	Namensraum der für XSD definiert ist
Präfix:	**tns**
Namensraum:	Verschieden
Beschreibung:	Für die Referenz auf das eigene Dokument benutzt
Präfix**:**	**verschieden**
Namensraum**:**	verschieden
Beschreibung:	Weitere benötigte Namensräume

Abbildung 18: Typische Präfixe von Namensräumen

Alle weiteren Namensräume müssen nicht in jedem WSDL-Dokument vollständig angegeben werden. Es werden dabei jeweils immer nur die benötigten eingebunden. Typischerweise wird der eigene Namensraum explizit eingebunden, um die definierten Typen genau adressieren zu können. Als Präfix für die Einbindung von Namensräumen sollte die Empfehlung des W3C verwendet werden (Abbildung 18). Die Anwendung ermöglicht eine konsistente Beschreibung über mehrere Dokumente hinweg. Die konsequente Verwendung erleichtert die Orientierung in der großen Vielfalt der Namensräume, die bei WSDL zum Einsatz kommen.

Das hier angewandte Element `<wsdl:definitions>` kann auch durch `<definition>` ersetzt werden, da der Namensraum durch

```
xmlns=http://schemas.xmlsoap.org/wsdl/
```

als Hauptnamensraum eingebunden ist. Hier im Beispiel ist er noch einmal separat durch

```
xmlns:wsdl=http://schemas.xmlsoap.org/wsdl/
```

eingebunden.

Um Entwicklern, die den Web Service nutzen wollen, den Zugriff zu erleichtern, sollten die Datentypen, Operationen und Services mit dem Tag `<wsdl:documentation ... />` genau beschrieben und dokumentiert werden.

In dem folgenden Abschnitten wurden einige spezielle SOAP-Bindungen verwendet. Diese werden aber nicht näher erklärt, da hier die reine WSDL – Beschreibung im Vordergrund steht. Der Leser sei verwiesen auf [WSDL-SOAP].

3.4.2. Service

Ein Service definiert die angebotenen Funktionen nach Außen, die dann mittels der definierten Ports bereitgestellt werden. Dabei können in einer WSDL Beschreibung auch mehrere Services angelegt werden. Zu beachten ist, dass dabei kein Port mit einem anderen kommunizieren kann, wodurch der Output eines Ports nie gleichzeitig Input eines anderen Ports ist. Der jeweilige Port definiert eine Verknüpfung zur angelegten Bindung. Im Beispiel wird der Port an getLectureBinding gebunden.

```
  <service name="LectureWebService">
    <port name="getLecturesPort"
       binding="tns:getLecturesBinding">
      <soap:address
         location=
"http://www.tele-task.de/LectureWebService.php"/>
    </port>
  </service>
```

Abbildung 19: Service-Definition aus dem Projektbeispiel

3.4.3. Binding

Im Abschnitt `<binding>` wird das Nachrichtenformat an ein Protokoll gebunden. Somit wird das logische Model, welches mit WSDL beschrieben wird, an ein reales physisches Model angebunden. Dabei wird eine Bindung immer für einen abstrakten Port Type vorgenommen. Es ist dabei nicht verboten, dass es mehrere Bindungen an ein und demselben Port Type gibt. Beispielsweise kann es daher gleichzeitig eine SOAP-over-HTTP-Bindung sowie eine SOAP-over-SMTP-Bindung geben.

```
<binding name="getLecturesBinding"
   type="tns:getLecturesPortType">
  <soap:binding style="rpc" transport=
"http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getLectures">
    <soap:operation soapAction="getLectures"/>
    <input>
      <soap:body
         namespace=
"http://www.tele-task.de/LectureWebService.wsdl"
         use="encoded"
         encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body
         namespace=
"http://www.tele-task.de/LectureWebService.wsdl"
         use="encoded"
         encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
</binding>
```

Abbildung 20: Bindung an SOAP-over-HTTP

3.4.4. Port Type

Bei einem Port Type handelt es sich um eine Ansammlung von abstrakten Arbeitsschritten. Hierbei wird grundsätzlich zwischen zwei verschiedenen Typen von Kommunikationsarten unterschieden. Das sind einmal die unidirektionale und andererseits die bidirektionale Kommunikation. Hier sollen kurz beide Arten mit entsprechenden Beispielen gezeigt werden

Unidirektional

- *One-way*

```
<wsdl:operation name="Search">
  <wsdl:input name="inputToken"
     message="SearchString"/>
</wsdl:operation>
```

- *Notification*

```
<wsdl:operation name="readOutAllBooks">
  <wsdl:output name="outputToken"?
     message="returnAllBooks"/>
</wsdl:operation>
```

Bidirektional

- *Request-response*

```
<wsdl:operation name="getBookPrice"
      parameterOrder="nmtokens">
  <wsdl:input name="inputToken"
      message="Book"/>
  <wsdl:output name="outputToken"
      message="Price"/>
  <wsdl:fault name="faultToken"
      message="FaultMessage"/>
</wsdl:operation>
```

- *Solicit-response*

```
<wsdl:operation name="checkUser"
      parameterOrder="nmtokens">
  <wsdl:output name="outputToken"
      message="Username"/>
  <wsdl:input name="inputToken"
      message="Password"/>
  <wsdl:fault name="faultToken"
      message="FaultMessage"/>
</wsdl:operation>
```

Die Abfolge kann mittels der vier Arten beschrieben werden. Zu beachten ist dabei, dass Fehlermeldungen nur in bidirektionalen Verbindung möglich sind. Um an die bereits vorangegangenen Beispiele anzuknüpfen sei zusätzlich das folgende Beispiel gegeben:

```
<portType name="getLecturesPortType">
  <operation name="getLectures">
    <input message="tns:getLectures"/>
    <output message="tns:getLecturesResponse"/>
  </operation>
</portType>
```

Abbildung 21: Beschreibung eines Port-Typs

3.4.5. Messages

Messages sind vergleichbar mit der Parameterliste einer Methode, wobei in einer Nachricht mehrere Parameter angegeben werden können. Diese werden in WSDL mittels `part` eingeleitet. Dabei ist nicht festgelegt, wie viele `part` Elemente eine `message` maximal enthalten darf.
Ein `part` Element kann außerdem mittels `element` auf ein XSD Element zeigen:

```
<part name="return"
      element="tns:LectureListType"/>
```

oder eben über `type` auf `simpleType` oder `complexType` zeigen:

```
<part name="return"
      type="xsd:f"/>
```

3.4.6. Types

Hier werden Datentypen festgelegt, die zum Austausch verwendet werden. Dabei sind diese nicht systemabhängig, sondern über XSD abstrakt definiert. Es interessiert zur Festlegung der Typen nicht die Kodierung des Typen im Protokoll. Das ist nur in dem schon erwähnten Fall von mehreren Bindungen sinnvoll.

Die in `types` verwendeten Datentypen müssen nicht direkt in der WSDL Datei definiert werden, sondern können separat in einem eigenen XML Schema erstellt werden. Dazu muss bei den Messages nur der korrekte Namensraum verwendet werden.

```
<wsdl:types>
  <schema
     xmlns="http://www.w3.org/2001/XMLSchema"
     targetNamespace=
"http://www.tele-task.de/LectureWebService.wsdl">
    <xsd:import namespace=
"http://schemas.xmlsoap.org/soap/encoding/"/>
    <xsd:import namespace=
       "http://schemas.xmlsoap.org/wsdl/"/>
    <complexType name="LectureListType">
      <complexContent>
        <restriction base="SOAP-ENC:Array">
          <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="string[]"/>
        </restriction>
      </complexContent>
    </complexType>
  </schema>
</wsdl:types>

<message name="getLectures"/>
<message name="getLecturesResponse"
  <part name="return" type="tns:LectureListType"/>
</message>
```

Abbildung 22: Beschreibung von Nachrichten und Datentypen mit WSDL

4. UDDI

4.1. Wofür steht UDDI?

Universal Description, Discovery, and Integration – kurz UDDI – wurde als Standardformat zur Beschreibung von Web Services im Jahre 2000 von Microsoft, Ariba und IBM eingeführt und hat sich seitdem durch die Akzeptanz in der Industrie zu einem der integralen Bestandteile von Web Services etabliert.
Die Intention von UDDI ist es, angebotene Web Services in so genannten Service-Registern mit allen relevanten Informationen zu erfassen. Potentielle Klienten können diese dann durch Suchanfragen an ein Register finden, das heißt sie werden dadurch (global) verfügbar. Dabei bestehen die in UDDI verfassten Beschreibungen aus den Weißen, den Gelben und den Grünen Seiten, auf die noch weiter eingegangen wird.

4.2 Der Aufbau von UDDI – Einträgen

4.2.1 Der Grundaufbau

Es handelt sich bei UDDI um eine Anwendung von Web Services mit einem XML-basierten Abfrage-Protokoll und einem standardisierten relationalen Datenmodell. Die eigentlichen Daten müssen wieder in eine SOAP-Nachricht verpackt werden.

Im Allgemeinen werden die in UDDI verfügbaren Informationen in die folgenden drei Kategorien eingeteilt:

Die „Weißen Seiten" (White Pages)

Hier findet man alle Angaben über die registrierten Anbieter von Web Services, wie zum Beispiel den Firmennamen, die Adressdaten, Telefon- und Faxnummern, E-Mail- und Internetadressen.
Während für einen Menschen ein Firmenname eindeutig ist, kann ein Computer einen solchen nicht immer korrekt auswerten. Aus diesem Grund findet man in allen UDDI-Einträgen ebenfalls die D-U-N-S-Nummer des Anbieters. Mit Hilfe dieser lässt sich das Unternehmen mittels eines von Dun & Bradstreet entwickelten Verfahrens identifizieren .

Die „Gelben Seiten" (Yellow Pages)

Der Vergleich zu dem bekannten Branchenverzeichnis ist passend, da sich hier alle Informationen zu Diensten befinden. Passende Dienste können anhand dieser Beschreibungen dann einfach heraus gesucht werden.

Die „Grünen Seiten" (Green Pages)

Letztendlich sind hier noch die technischen Daten zu den Servicetypen, Schnittstellen und Protokollen, die bei den E-Businesstransaktionen verwendet werden, veröffentlicht.

Die verschiedenen Abschnitte eines UDDI-Eintrags können nun anhand dieser Einteilung klassifiziert werden:

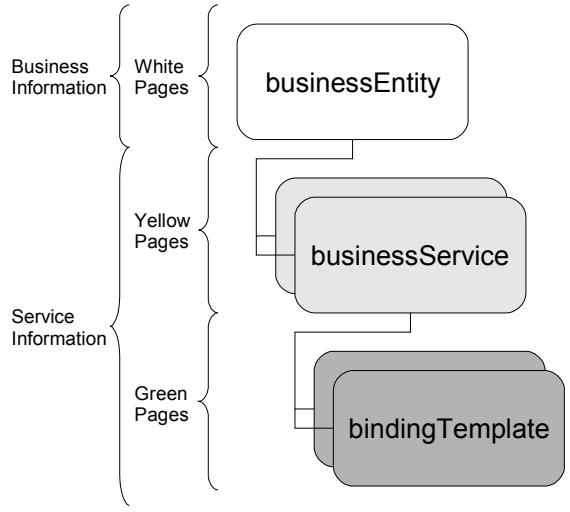

Abbildung 23: Übersicht der Datenstrukturen

Wichtig zu bemerken ist, dass die BusinessEntity immer angegeben werden muss. Doch wie in den Abbildungen 23 und 24 zu erkennen ist, können mehrere Web Services zu einem Anbieter gehören. Als Beispiel seien hier einige von Microsoft angebotene Services aufgelistet:
Online Shopping, Microsoft Developer Network, Electronic Business Integration Services, Volume Licensing Select Program, ...
(Quelle: https://uddi.ibm.com/ubr/find)

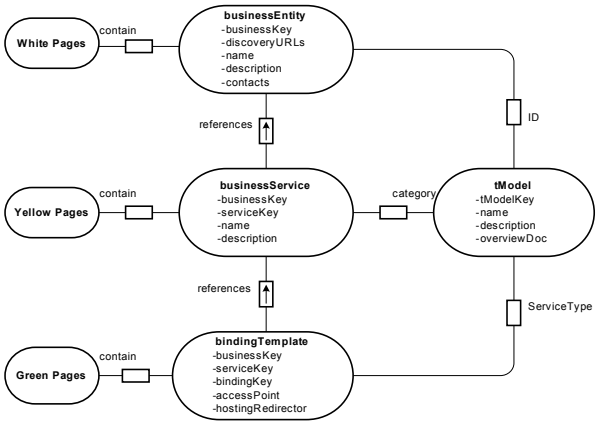

Abbildung 24: Entity Relationship Diagramm für ein UDDI-Verzeichnis (Quelle: [NICOLAI2005])

Wie auf den ersten Blick zu sehen ist, enthalten Entities der oberen Schicht Elemente der unteren Schichten – so kann eine BusinessEntity mehrere BusinessServiceEntities enthalten, diese BusinessServiceEntities wiederum können mehrere BindingTemplateEntities enthalten, die ihrerseits mehreren tModels entsprechen können – dazu jedoch später mehr.

Innerhalb eines obligatorischen SOAP-Envelopes kommen dann folgende Elemente zum Einsatz, die UDDI ausmachen:

```
<businessEntity
  authorizedName="testuser"
  businessKey="ab058fd2-47f6-058f-e5e8-f3b5bf62f171"
  operator="Sun Microsystems Inc.">
```

Abbildung 25: Beispiel für eine BusinessEntity

Eine *BusinessEntity* enthält Informationen über den Anbieter eines Web Services. Diese umfassen den Namen, eine für Menschen lesbare Beschreibung, eine Liste mit URLs unter welchen der Anbieter zu finden ist, sowie Kontaktinformationen (z.B. Telefonnummer). Das wichtigste ist jedoch der *BusinessKey*, da erst dieser einen Anbieter weltweit eindeutig durch einen zentral vergebenen Schlüssel identifiziert.

Die BusinessEntity umfasst auch eine Liste

von *BusinessServiceEntities*, in denen dann die einzelnen Web Services im Detail beschrieben werden. So findet man hier zu jedem Web Service in der entsprechenden Entity den dazugehörigen *ServiceKey*, der ebenfalls wieder weltweit eindeutig diesen Dienst identifiziert. Unter anderem findet man hier auch den passenden Namen und die Beschreibung zu dem Web Service. Um bei einer Suche nach nutzbaren Services auch den Anbieter ausfindig machen zu können, haben die BusinessServiceEntities noch den BusinessKey der sie umfassenden BusinessEntity hinterlegt.

Jeder Web Service kann seine Service Access Points veröffentlichen. Dies wird durch eine Liste von *BindingTemplateEntities* erreicht, die innerhalb jeder BusinessServiceEntity zu finden ist. In diesen Entities wird wieder ein identifizierender Schlüssel, der *BindingKey* benutzt. Jede BindingTemplateEntity enthält außerdem wieder einen Namen und eine Beschreibung.

```
<bindingTemplate
  serviceKey="ef25102d-2171-454c-ade9-3dd7a4a914ee"
  bindingKey="f46fced9-2b8a-4817-b957-f8d8aca0a2f9">
  <accessPoint URLType="http">
    http://localhost/SalesReportUSA/SalesReport.asmx
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo
      tModelKey="uuid:b28fe40a-ea62-
                          4657-88d5-752d8a6cdf77" />
  </tModelInstanceDetails>
</bindingTemplate>
```

Abbildung 26: Beispiel für ein BindingTemplate

Wesentlich ist auch das Konzept der *tModelEntities*. Wenn maschinell nach Web Services gesucht wird, wenn also ein Web Service selbständig nach kompatiblen Web Services sucht, so kann dies anhand der *tModelKeys* geschehen. Jeder Web Service entspricht nämlich durch sein Interface, seinen Service Access Points sowie seiner Datenrepräsentation immer mindestens einem tModel. Sollte ein anderer Web Service den gleichen tModelKey haben, so sind diese beiden potentiell kompatibel zueinander (wobei natürlich noch nicht gesagt ist, ob die Services auch inhaltlich zueinander passen).

Eine Liste von Web Services mit dem gleichen tModelKey ist als eine Art Kompatibilitätsliste zu sehen, da man hier keine externen Wrapper einsetzen muss um die Kompatibilität von diesen Web Services zu gewährleisten.

```
<tModelInstanceDetails>
  <tModelInstanceInfo tModelKey=
      "uuid:4CD7E4BC-648B-426D-9936-443EAAC8AE23">
    <description xml:lang="en">
      UDDI SOAP Inquiry Interface
    </description>
  </tModelInstanceInfo>
</tModelInstanceDetails>
```

Abbildung 27: Dieses Beispiel enthält den tModelKey des Interfaces das genutzt wird, wenn man eine UDDI-Anfrage per SOAP absendet.

4.2.2 Zusammenfassung dieses Abschnittes

Es gibt bei UDDI mehrere Entities, die in einer hierarchischen Beziehung zueinander stehen:

- Eine **BusinessEntity** wird durch den *BusinessKey* identifiziert und enthält Informationen über den Anbieter (weiße Seiten).
- In der **BusinessServiceEntity**, die durch den *ServiceKey* identifiziert wird, findet man Angaben über den angebotenen Service (gelbe Seiten).
- Eine **BindingTemplateEntity** wird durch einen *BindingKey* identifiziert und gibt die Service Access Points an (grüne Seiten).
- Die **tModelEntity** wird durch den *tModelKey* identifiziert und gibt an, ob ein Web Service zu einem anderen kompatibel ist (grüne Seiten)

4.3 Die Nutzung von UDDI

4.3.1. Öffentliche und firmeninterne Service Register

Jeder Entwickler kann per UDDI eine Anfrage starten und sich so den für ihn am besten geeigneten Web Service suchen. Doch gerade in öffentlichen Registern gibt es viele UDDI-Einträge, die nicht mehr aktuell sind (z.B. „Hello-World"-Web Services) oder nicht das erfüllen, was sie versprechen. Gerade beim öffentlichen UDDI-Register von IBM kann sich jeder registrieren, um eigene Services zu veröffentlichen. Dabei wird die Funktionalität dieser jedoch nicht kontrolliert.

Zum Testen mögen solche Register demzufolge ihren Zweck erfüllen, jedoch für die Industrie reichen sie nicht aus. Daher haben alle großen Unternehmen, die mit Web Services arbeiten, firmeninterne Service Register, in denen nur selbstentwickelte oder von anderen

Anbietern eingekaufte Web Services veröffentlicht werden.

4.3.2. Die Inquiry – API

4.3.2.1. Eine Beispielanfrage

```
<?xml version="1.0" encoding="utf-8"?>
<Envelope xmlns=
  "http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <find_service  maxRows="50"
       xmlns="urn:uddi-org:api"
       generic="1.0">
      <findQualifiers>
        <findQualifier>
          sortByNameAsc
        </findQualifier>
        <findQualifier>
          sortByDateAsc
        </findQualifier>
      </findQualifiers>
      <name>%image%</name>
    </find_service>
  </Body>
</Envelope>
```

Abbildung 28: Suche nach Web Services über einen UDDI Web Service

Bei diesem Beispiel handelt es sich um eine Suchanfrage nach einem Service (`find_service`), dessen Name die Zeichenkette **`%image%`** enhalten soll (die Prozentzeichen stehen bei Suchanfragen für Wildcards). Die Suchparameter (`findQualifier`) sind als Liste mit Priorität zu lesen:

1. Absteigend nach Namen sortiert
2. Absteigend nach Datum sortiert

Wie schon die UDDI-Einträge müssen auch die Suchanfragen in einem SOAP-Envelope eingebettet sein.

4.3.2.2. Allgemeine Anfragen

Die UDDI-Inquiry API stellt einem Entwickler Funktionen zur Abfrage anhand bestimmter Parameter zur Verfügung. Die so erstellten Anfragen werden dann in einem SOAP-Envelope eingebettet an ein Service Register geschickt und dort bearbeitet. Es findet dabei keine Identifizierung statt. Das heißt alle erhalten die gleichen Ergebnisse, da bei öffentlichen Registern die Einträge frei verfügbar sind.

Die bereitgestellten Funktionen erlauben es, direkt nach den in UDDI vorhandenen Entities zu suchen. So lassen sich Anbieter, angebotene Web Services, deren Bindings und tModels durch die entsprechenden **find-Funktionen** (`find_business`, `find_service`, `find_binding` und `find_tmodel`) finden. Anbieter lassen sich auch indirekt per `find_relatedBusinesses` finden. Dabei wird ein BusinessKey übergeben um zu prüfen, ob es irgendwelche Verbindungen (`publisherAssertions`, dazu mehr im Abschnitt 4.3.3) zu dem entsprechenden Anbieter gibt. Wenn ja, so werden die entsprechenden Anbieter zurückgeliefert.

Der primäre Suchparameter, der in jedem Fall übergeben werden kann, ist natürlich der Key der entsprechenden Entity, zumal diese dadurch sofort problemlos ausfindig gemacht werden kann.

Sollte dieser aber nicht bekannt sein, so sind auch Suchen nach den einzelnen Attributen, die jede Entity hat, möglich – z.B. nach dem Namen. Parameter der Suche können dabei direkt als `findQualifier` übergeben werden. Diese `findQualifiers` sind eine optionale Liste spezieller Sucheigenschaften. So sorgt z.B. das Element:

```
<findQualifier>
  caseSensitiveMatch
</findQualifier>
```

dafür, dass als Resultat nur Entities zurückgeliefert werden, die auch in Groß- und Kleinschreibung mit dem Suchbegriff übereinstimmen. Weitere wichtige Qualifier sind „`sortByNameAsc`“, „`sortByNameDesc`“, „`sortByDateAsc`“, „`sortByDateDesc`“ (für die jeweilige Sortierung der Ergebnisse) und „`exactMatch`“ (um Ergebnisse zu erzwingen, die nur exakt mit dem Suchbegriff übereinstimmen).

Der Suchbegriff selbst wird als `<name>Suchbegriff</name>` angegeben und kann mit den bereits erwähnten Wildcards verallgemeinert werden.

Die Syntax der Suche nach einer BusinessEntity sieht damit wie folgt aus:

```
<find_business maxRows="10"
   generic="2.0"
   xmlns="urn:uddi-org:api_v2">
  <findQualifiers>
    <findQualifier>
      caseSensitiveMatch
    </findQualifier>
  </findQualifiers>
  <name>%IBM%</name>
  <name>%Microsoft%</name>
</find_business>
```

Diese Anfrage würde das Service Register nach Anbietern mit den Zeichenketten *IBM* oder *Microsoft* durchsuchen.

Um nicht von den Suchergebnissen überschwemmt zu werden, wird eine Begrenzung der Suchergebnisse durch `maxRows="10"` erreicht – nur maximal 10 Ergebnisse werden zurückgegeben.

Anfragen nach anderen Entities sehen im Grunde ähnlich aus und unterscheiden sich nur durch die Parameter, nach denen sich suchen lässt, sowie nach der Art des Ergebnises.

4.3.2.3 Die Antwort

Als Ergebnis erhält man eine Liste mit Entities, welche den Parametern entspricht – `find_service` würde eine ServiceList zurückliefern, `find_business` also eine BusinessList.

```
<?xml version="1.0" encoding="UTF-8" ?>
<SOAP:Envelope xmlns:SOAP=
   "http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <serviceList generic="1.0"
       xmlns="urn:uddi-org:api"
       operator="www.ibm.com/services/uddi"
       truncated="false">
      <serviceInfos>

<serviceInfo
 serviceKey="34A76230-132A-11DA-B2FE-000629DC0A53"
 businessKey="8DEAC090-1329-11DA-B2FE-000629DC0A53">
  <name>Grid Image Compression Service</name>
</serviceInfo>

<serviceInfo
 serviceKey="C7553B40-B37B-11D7-8BE8-000629DC0A53"
 businessKey="A5641D30-B37B-11D7-8BE8-000629DC0A53">
  <name>Image Conversion</name>
</serviceInfo>

<serviceInfo
 serviceKey="FF7A74F0-AFF6-11D7-903D-000629DC0A53"
 businessKey="AA46A0D0-AFF6-11D7-903D-000629DC0A53">
  <name>Image Storage and Retrieval</name>
</serviceInfo>

...

      </serviceInfos>
    </serviceList>
  </SOAP:Body>
</SOAP:Envelope>
```

Abbildung 29: eine gekürzte Antwort zur Beispielanfrage aus Abbildung 28

Zu bemerken ist dabei, dass alle in der Liste aufgeführten Web Services nur mit ihrem Namen, dem BusinessKey und dem ServiceKey angegeben sind.

Eine solche Ergebnisliste umfasst nämlich immer nur die Namen der Entities und die in ihnen vorkommenden Keys. Um mehr Informationen zu einer bestimmten Entity zu erhalten, müssen dann neue Anfragen gestartet werden.

Solche Anfragen werden durch die Funktionen `get_businessDetail`, `get_serviceDetail`, `get_bindingDetail`, `get_tModelDetail` und `get_businessDetialExt` ermöglicht. Wenn man nur die Liste der Entities hat und man will über einen Service, den man sich heraus gesucht hat, mehr wissen als den ServiceKey und seinen Namen, so kann man z.B. mit der Funktion `get_serviceDetail` und dem Servicekey, die gesamte BusinessServiceEntity auslesen. Diese kann dann beispielsweise die menschenlesbare Beschreibung der Funktionalität des Web Services enthalten.

4.3.3. Publishing API

Die Funktionen der Publishing-API dienen zum publizieren und verändern der Einträge einer Registry. So ist es möglich, UDDI-Strukturen durch einen entsprechenden delete-Befehl (`delete_business`, `delete_service`, `delete_binding` und `delete_tModel`) zu entfernen, wobei eine Entity aus allen sie enthaltenden Entities entfernt wird (z.B. wird ein Service aus seiner BusinessEntity entfernt). Ein Sonderfall liegt beim Löschen von tModels vor, da diese ja von mehreren Web Services genutzt werden. Wenn ein tModel gelöscht werden soll, so wird der tModelKey-Wert nur unsichtbar gemacht. Wenn nun jemand nach genau diesem Schlüsselwert sucht, so erhält er eine leere Liste zurück. Trotzdem ist es durch `get_serviceDetails` noch möglich, auf diesen tModelKey zuzugreifen.

Neben dem Löschen ist selbstverständlich auch das Erstellen von UDDI-Einträgen möglich. Erzeugt werden dazu Entities, die durch Keys miteinander verbunden sind. So erstellt man z.B. eine BindingEntity mit der Funktion `save_binding`, indem man dieser eine Entity übergibt. Diese übergebene Entity muss allerdings einen ServiceKey enthalten, da das Service Register nur dann in der Lage ist, diese BindingEntity dem richtigen Service zuzuordnen. Die Funktionen `save_business`, `save_service` und `save_tModel` arbeiten wieder nahezu identisch mit angepassten Parametern.

Bei der Änderung von UDDI-Einträgen kommt nun zum ersten Mal eine Identifizierung ins Spiel, da ein Datensatz nur vom jeweiligen „Besitzer“ verändert werden darf. Dazu besitzt

jede Entity das Attribut `authorizedName`. Den Wert dieses Attributs kann man für jede Entity, falls vorhanden, durch `get_authToken` erhalten.

Die schon angesprochenen Relationen zwischen Anbietern lassen sich ebenfalls durch die Publishing-API verwirklichen. Mit der Funktion `add_publisherAssertions` lassen sich BusinessEntities miteinander verbinden.

In diesem Beispiel wird die Aussage gemacht, dass BusinessEntity BE1 eine Holding ist, zu der die BusinessEntity BE2 gehört:

```
<add_publisherAssertions
   xmlns="urn:uddi-org:api_v3" >
  <authInfo>FFFFF</authInfo>
  <publisherAssertion>
    <fromKey>BE1</fromKey>
    <toKey>BE2</toKey>
    <keyedReference
       tModelKey="uddi:uddi.org:relationships"
       keyName="Holding Company"
       keyValue="parent-child" />
  </publisherAssertion>
</add_publisherAssertions>
```

Die so erstellte Relation ist jedoch erst durch `find_relatedBusiness` zu entdecken, wenn der autorisierte Benutzer der BusinessEntity BE2 diese Relation ebenfalls für seine Entity bestätigt, indem er die identische Relation eingibt.

Gelöscht werden diese dann durch die Funktion `delete_publisherAssertions`, indem man die zu löschende Struktur noch einmal komplett übergibt, um sie identifizieren zu können.

5. Programmiermodelle für Web Service Anwendungen

Aus dem Zusammenspiel der in den vorangegangenen Kapiteln vorgestellten Standards ergibt sich nun eine typische Architektur für XML-basierte Web Services.

Wie Abbildung 30 illustriert, wird seitens des Servers eine WSDL-Datei erstellt und der Web Service bei einem UDDI-Namensdienst publiziert. Der Client lokalisiert über UDDI einen für seine Aufgabe passenden Web Service und erfährt aus der WSDL Beschreibung vom Server, welche Ports zur Verfügung stehen und dass er den Web Service in diesem Fall via SOAP aufrufen kann.

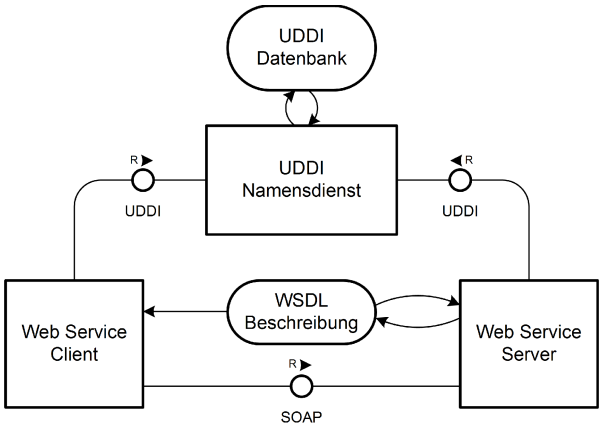

Abbildung 30: typische Architektur für XML-basierte Web Services

Web Services kapseln die auf dem Server verwendete Implementierungsplattform, ihre Infrastrukturanforderungen und Abhängigkeiten von anderen Web Services. Sie sind in sich geschlossen, selbst beschreibend und viel eigenständiger als typische Komponenten oder Applikationen. Aus dieser großen Unabhängigkeit ergibt sich der Nachteil, dass Web Services nicht in unterschiedlichen Kontexten ausgeführt werden können, denn die Flexibilität der Komponentenmodelle lässt sich nicht vollständig auf eine lose gekoppelte Welt übertragen. Web Services können nur so wie sie sind genutzt werden, aber explizite Kontextabhängigkeiten oder Rekonfiguration würden ohnehin weit über die aktuellen Denkweisen und Anforderungen hinausgehen.

Auf abstrakter Ebene und aus Sicht eines Nutzers können Web Services als Softwarekomponenten angesehen werden, die zur Laufzeit einer Client-Anwendung lokalisiert und eingebunden werden können. Genauer betrachtet lassen sich die zu Services gruppierten Ports, die in WSDL beschrieben sind, mit den Methoden eines Interfaces vergleichen. Diese Analogie würde dazu führen, die Aufrufe als synchrone entfernte Funktionsaufrufe zu modellieren. Um sich für ein geeignetes Programmiermodell zu entscheiden muss man die besonderen Eigenschaften von Web Services mit einbeziehen. Diese ergeben sich aus der Verwendung des Internets als Übertragungsmedium. Als Nutzer muss man mit hohen, unvorhersagbaren Antwortzeiten und jederzeit möglichen Fehlern rechnen. Es kann auch passieren, dass eine Nachricht in einer Warteschlange gespeichert wird bis Ressourcen zur Bearbeitung zur Verarbeitung verfügbar sind (Message Queuing). Da bei der Auswahl des

Servers, der eine Anfrage beantworten wird, eine Lastverteilung (Load-Balancing) möglich sein muss, gibt es für eine Client-Anwendung keine Garantie, dass aufeinander folgende Anfragen vom gleichen Server bearbeitet werden oder dass ein bestimmtes Objekt mit einem bestimmten Zustand auf Serverseite existiert. Ebenso ist es für einen Web Service Anbieter nicht klar, ob der anfragende Client auch selbst die Antwort bearbeitet oder sie weiterleitet (Client Side Redirection).

Die Implementierung eines WSDL-Ports als Methode an einer synchronen Klasse ist durchaus geeignet, wenn keine Abhängigkeiten zu weiteren Web Services oder anderen langsamen Ressourcen bestehen. Beim Aufruf von Web Services kann ein synchroner Ansatz zu ernsthaften Performanzproblemen führen. Man benötigt viele Threads für die blockierenden synchronen Anfragen und kann in der Zwischenzeit nichts nützliches rechnen. Hier sollte man sich also für ein asynchrones Programmiermodell entscheiden.

6. Realisierung mit PHP5

Nun soll an Beispielen die praktische Umsetzung von Web Services demonstriert werden. Mit PHP5 lassen sich Web Services auf einfache Art realisieren, denn die PHP Implementierungen der Standards haben nicht so viel Plattform-Ballast wie Lösungen in .Net oder Java und lassen auch Einblicke in die unteren Ebenen der Kommunikation zu. Zunächst soll der Zugriff auf UDDI-Verzeichnisse über die Inquiry-API gezeigt werden. Danach wird das Anbieten und Nutzen von Web Services mit PHP5 ausführlich erklärt.

6.1. UDDI-Abfragen

Wie in Kapitel 4 besprochen handelt es sich bei UDDI um einen Verzeichnisdienst für Web Services. Dabei werden große öffentliche UDDI Verzeichnisse von einigen Unternehmen bereitgestellt. Einerseits bieten sie diesen Dienst selbst über ihre Webseiten an und andererseits gibt es Möglichkeiten die Informationen direkt über eine Web Services abzufragen.

Eine elegante Lösung wird im PHP Extension and Application Repository [PEAR] bereitgestellt und ist auch mit PHP 4 kompatibel. Das Paket UDDI bietet eine Klasse UDDI an, welche die vollständige UDDI 2.0 Inquiry API implementiert. Das folgende UML Diagramm zeigt die wichtigsten Methoden der Klasse.

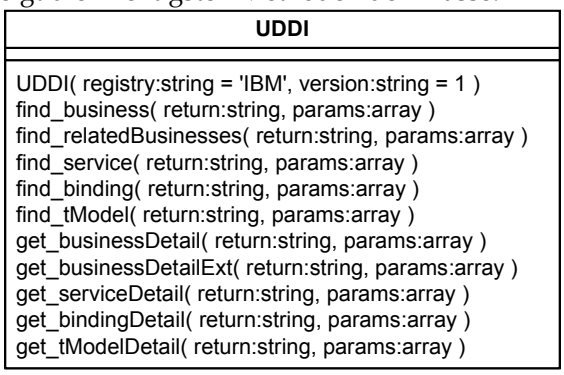

Abbildung 31: Klassendiagramm der UDDI-Klasse aus dem PEAR-Paket

Mit dem UDDI-Paket lässt sich komfortabel ein UDDI Namensdienst abfragen. Es wird hier nur auf einige Methoden näher eingegangen. Für eine weiterreichende Beschreibung sei die Dokumentation des Moduls empfohlen.

```
require_once 'UDDI/UDDI.php';

$my_uddi = new UDDI('IBM', 1);
$result = $my_uddi->find_service(
  array(
    'name' => '%image%',
    'maxRows' => 50,
    'findQualifiers' =>
      'sortByNameAsc,sortByDateAsc'
  )
);
```

Abbildung 32: Web Service Suche mit UDDI

Über den Konstruktor selbst legt man eine Datenbank und die UDDI-Version fest. In diesem Beispiel erfolgt die Anfrage bei IBM mit der Version 1. Die Zugangsdaten für die UDDI Verzeichnisse von IBM und Microsoft sind in der der Klasse als Konstanten vorhanden. Um auf andere Ressourcen wie beispielsweise einen eigenen UDDI Namensdienst zuzugreifen muss ein Registry-Array mit Daten zur URL und Portnummer zum Veröffentlichen und Anfragen von Web Services gefüllt werden.

Mit dem Aufrauf von `find_service()` findet nun eine Suchanfrage nach Web Services mit dem Namen `%image%` statt, wie sie in Abbildung 28 bereits vorgestellt wurde. Dabei übergibt man der Methode ein Array mit den entsprechenden Parametern und erhält, wenn alles funktioniert, eine Zeichenkette im XML Format zurück (siehe Abbildung 29). Diese enthält die geordneten Werte für die Anfrage nach den ersten 50 Web Services, welche das

Wort „image“ im Namen beinhalten. Alle weiteren find- und get-Methoden dienen dem weiteren Navigieren in den verschiedenen Seiten des UDDI-Models.

6.2. Eine Web Service Implementierung mit Kommunikation via SOAP

Es empfiehlt sich die Version 5 von PHP zu verwenden, denn SOAP-Server und -Client wurden als Erweiterungsmodul in PHP 5 integriert. Der compilierte C-Code dieser SOAP-Implementierung ist wesentlich schneller als Implementierungen in PHP, die ebenfalls zur Verfügung stehen. Um die SOAP Erweiterung nutzen zu können wird die GNOME XML Bibliothek benötigt. SOAP 1.1, SOAP 1.2 und WSDL 1.1 werden damit von PHP erkannt und direkt unterstützt.

Sollte aufgrund von technischen Hindernissen eine Umstellung auf das SOAP Modul nicht möglich sein, kann hier auf Abschnitt 6.3 verwiesen werden.

Bei der Entwicklung von Web Services mit WSDL-Dateien ist es hilfreich, den WSDL-Cache der PHP 5 Erweiterung auszustellen, da sonst auch fehlerhafte WSDL Dateien gespeichert werden und so zu unangenehmen Fehlern führen.

6.2.1. Das Projektbeispiel

Für die weitere Implementierung von Web Services ist es erforderlich, das etwas umfangreichere Beispiel zu erläutern. Es handelt sich dabei um eine typische Internet-Anwendung aus dem Teleteaching-Szenario, das in Kapitel 3.4 für WSDL vorgestellt wurde. Der Benutzer soll aus einem Angebot von Veranstaltungen eine auswählen können. Die Informationen über die angebotenen Veranstaltungen werden intern in einer Datenbank verwaltet. Abbildung 33 verdeutlicht den lokalen Aufbau ohne Web Service.

Die Klassen `ADOdb` (adodb.sourceforge.net) und `Smarty` (smarty.php.net) sind vorgefertigte Komponenten, welche dabei helfen die Datenbankabfragen unabhängig vom eingesetzten Datenbanksystem zu entwickeln und mittels Templates die Präsentationslogik von der Geschäftslogik zu trennen. `SecureSmarty` ist eine zusätzlich eingeführte Klasse, die in ihrem Konstruktor die Sicherheitseinstellungen für die Template-Engine verstärkt.

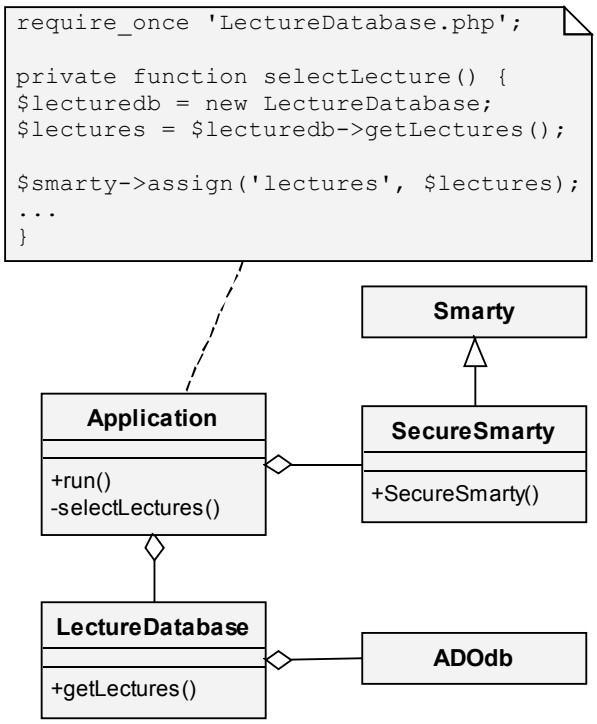

Abbildung 33: Projektbeispiel ohne Web Service

Die Hauptklasse `Application` beinhaltet die Logik zum Aufbau der Website. Sie wird beim Aufruf der Website instanziert und mittels `run()` gestartet. Diese Dispatcher-Methode ruft dann die private Methode `selectLectures` auf. In ihr werden die Templatevariablen von Smarty gesetzt. Ein besonderes Augenmerk gilt dabei der Variablen `$lectures`. Es handelt sich dabei um ein String Array mit Daten aus der Datenbank. Diese werden in der Version ohne Web Service von einer Instanz der Klasse `LectureDatabase` angefordert. (siehe Quelltext-Auschnitt in Abbildung 33).

6.2.2. Web Service Server

PHP 5 stellt sowohl ein Klasse für den Server als auch für den Client bereit. Der hier behandelte Server bietet Funktionen, welche den Umgang mit Web Services recht komfortabel machen.
Der Konstruktor bietet zwei verschiedene Modi an. Beim *Non-WSDL-Modus* handelt es sich um eine Variante in der das gegenüberliegende Objekt nicht bekannt ist, d.h. Methodenaufrufe, Ort und andere Eigenschaften müssen manuell im Parameter `options` angegeben oder in einer externen Dokumentation herausgefunden

werden. Andererseits bietet er den *WSDL-Modus* an. Dieser unterstützt WSDL-Dateien (siehe Kapitel 3) und erleichtert den Umgang mit Web Services, da man nun die Schnittstellenbeschreibung und andere wichtige Informationen in der WSDL-Beschreibung finden kann. Der options-Parameter erlaubt es zudem weitere Eigenschaften wie zum Beispiel SOAP Version oder URI anzugeben.

Die Methoden `addFunction()` oder `setClass()` registrieren nun die Funktionen oder Klassen, welche der Server nach außen als Ports anbieten soll. Dabei sei gesagt das alle public Methoden offen gelegt werden. Um bestimmte Funktionen nicht freizugeben, bieten sich demnach zwei Möglichkeiten an. Man kann in der WSDL-Beschreibung nur die gewünschten Methoden angeben oder eine Wrapperklasse um die eigentliche Klasse legen, um nur bestimmte Methoden zu offenbaren.

Mit der Methode `handle()` wird dem Server der Befehl gegeben auf eingehende Anfragen zu antworten.

Weiterhin existieren Methoden zur Fehlerbehandlung (`fault()`), zum Abruf aller verfügbaren Funktionen (`getFunctions()`) und um Persistenz zu realisieren (`setPersistence()`). Letztere bietet sich vor allem dann an, wenn man den eigentlich zustandslosen Web Service mittels Session in einen zustandsbasierten umwandeln möchte. Bei Web Services mit Zustand ist jedoch aufgrund der in Kapitel 5 erläuterten Eigenschaften Vorsicht geboten. Die gesamte Schnittstelle für einen SOAP-Server sieht man in der folgenden Abbildung.

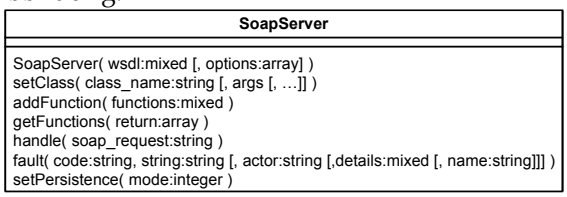

Abbildung 34: Die Klasse SoapServer aus der PHP5-Erweiterung

Das Projektbeispiel soll nun so verändert werden, dass die Datenbank auf einem anderen Rechner liegt und die Datenübertragung mittels eines Web Service realisiert wird. Dabei wird eine neue Datei mit dem Namen *LectureWebService.php* erstellt, welche den Web Service Server darstellen wird. Dieser Server stellt mittels WSDL-Datei die Klasse `LectureDatabase` nach außen bereit und somit auch die Methode `getLectures()`. Das folgende Diagramm zeigt nun den veränderten Teilaufbau.

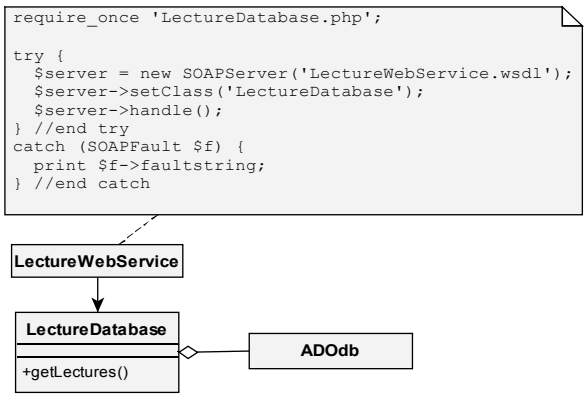

Abbildung 35: Erster Teil des Projektbeispiels verändert zu einem Server für Web Services

Es sei an dieser Stelle darauf hingewiesen, dass es sich bei `LectureWebService` um keine Klasse handelt sondern lediglich um ein kleines PHP-Skript, das sich nun auch recht einfach erklären lässt. `$server` wird als `SoapServer` im WSDL-Modus initialisiert. Die verwendete WSDL-Datei *LectureWebService.wsdl* ist die aus Kapitel 3 bekannte Beschreibung.

```
<?xml version='1.0' encoding='UTF-8'?>
<wsdl:definitions
  name="LectureWebService"
  targetNamespace=
    "http://www.tele-task.de/LectureWebService.wsdl"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:tns=
    "http://www.tele-task.de/LectureWebService.wsdl"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENC=
    "http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:ns="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:ns1="http://schemas.xmlsoap.org/wsdl/mime/"
>

  <wsdl:types>
    <schema
       xmlns="http://www.w3.org/2001/XMLSchema"
       targetNamespace=
"http://www.tele-task.de/LectureWebService.wsdl">
      <xsd:import namespace=
"http://schemas.xmlsoap.org/soap/encoding/"/>
      <xsd:import namespace=
         "http://schemas.xmlsoap.org/wsdl/"/>
      <complexType name="LectureListType">
        <complexContent>
          <restriction base="SOAP-ENC:Array">
            <attribute ref="SOAP-ENC:arrayType"
               wsdl:arrayType="string[]"/>
          </restriction>
        </complexContent>
      </complexType>
    </schema>
  </wsdl:types>
```

```
  <message name="getLectures"/>
  <message name="getLecturesResponse"
    <part name="return" type="tns:LectureListType"/>
  </message>

  <portType name="getLecturesPortType">
    <operation name="getLectures">
      <input message="tns:getLectures"/>
      <output message="tns:getLecturesResponse"/>
    </operation>
  </portType>

  <binding name="getLecturesBinding"
     type="tns:getLecturesPortType">
    <soap:binding style="rpc" transport=
"http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getLectures">
      <soap:operation soapAction="getLectures"/>
      <input>
        <soap:body
           namespace=
"http://www.tele-task.de/LectureWebService.wsdl"
           use="encoded"
           encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
      </input>
      <output>
        <soap:body
           namespace=
"http://www.tele-task.de/LectureWebService.wsdl"
           use="encoded"
           encodingStyle=
"http://schemas.xmlsoap.org/soap/encoding/"/>
      </output>
    </operation>
  </binding>

  <service name="LectureWebService">
    <port name="getLecturesPort"
       binding="tns:getLecturesBinding">
      <soap:address
         location=
"http://www.tele-task.de/LectureWebService.php"/>
    </port>
  </service>

</wsdl:definitions>
```

Abbildung 37: Die WSDL-Beschreibung LectureWebService.wsdl (siehe Kapitel 3)

Danach wird die Datenbank-Klasse beim Server registriert und zum Schluss die Methode `handle()` aufgerufen. Das Ganze wird von einem `try-catch`-Block umrundet um eventuelle Ausnahmen abzufangen.

6.2.3. Web Service Client

Nun geht es darum den Server zu erreichen und die Daten abzufragen. Dabei wird analog zum Server vorgegangen und als erstes die Klasse `SoapClient` behandelt.

Der Konstruktor legt ein neues Client-Objekt an. Dabei gelten ähnliche Richtlinien wie beim Server mit einem Non-WSDL und einem WSDL Modus. Bei einfachen Aufrufen genügt es diese lokal am Client-Objekt auszuführen. Dies ermöglicht dem Programmierer mittels eines Stubs viele Funktionen vorher zu testen ohne den eigentlichen Web Service zu benutzen. Intern werden dann die entsprechenden Methoden benutzt, wie zum Beispiel `__soapCall()`, welche eine direkte Web Service Anfrage stellt.

Weitere Methoden ermöglichen unter anderem das Senden von Cookies oder das Auslesen des SOAP Headers. Die gesamte Spezifikation sieht man in Abbildung 37.

Für das Projektbeispiel heißt es nun die Methode `selectLectures()` der Klasse `Application` so zu verändern, dass sie die Werte über den Web Service erfährt.

```
private function selectLecture() {
// remote procedure call via SOAP
$lecturedb = new SoapClient(
   'http://www.tele-task.de/LectureWebService.wsdl');
$lectures = $lecturedb ->getLectures();
$smarty->assign('lectures', $lectures);
...
}
```

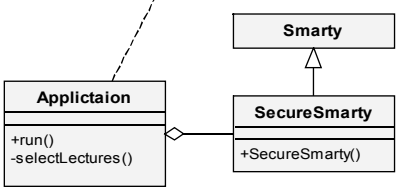

Abbildung 38: Zweiter Teil des Projektbeispiels verändert zu einem passenden Client

Dazu legt sie anstatt einer Instanz der `LectureDatabase` eine Instanz des

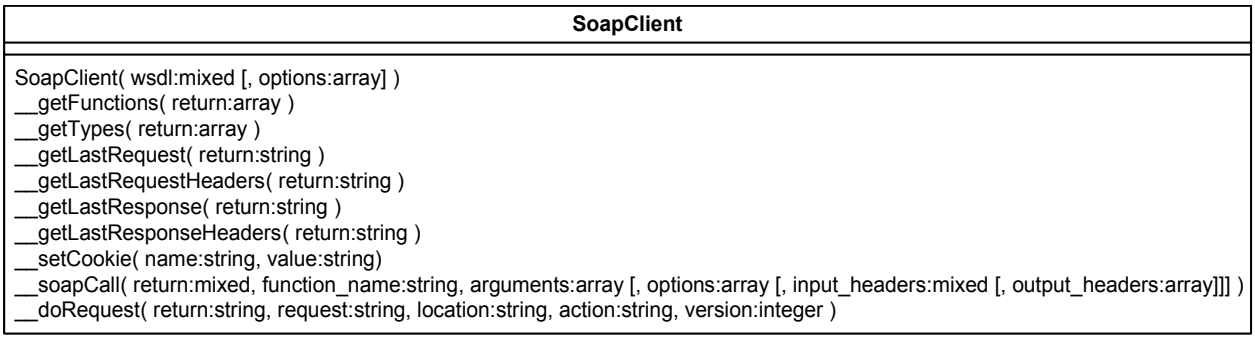

Abbildung 36: UML 2.0 Klassendiagramm vom SOAP-Client der PHP5 Erweiterung

`SoapClient` mit der URL der WSDL Datei an. Hiermit weiß der Client nun, was der Web Service anbietet und kann dies ohne große Umstände auch benutzen. Für den Anwender ändert sich das sichtbare Ergebnis nicht. Maximal kann sich die Laufzeit durch den Aufruf des Web Service verschlechtert haben. Zusätzlich gibt es aber nun auch die Möglichkeit die Angaben zu den Veranstaltungen durch andere Anwendungen abzufragen und unter Umständen weitere Clients in anderen Programmiersprachen und für andere Plattformen zu erschaffen.

6.3. Weitere SOAP-Klassen

Die PHP5-Erweiterung stellt noch weitere Klassen bereit, die in den Beispielen nicht benötigt wurden.

Mit der Klasse `SoapFault` ist es möglich verschiedene SOAP-Fehler zu erzeugen und als Antwort zu versenden. Die Manipulation des SOAP-Headers ist mit der Klasse `SoapHeader` möglich, wenn man direkt mit `SoapClient->__soapCall()` eigene SOAP-Nachrichten versendet. `SoapParam` und `SoapVar` erlauben das Bezeichnen und Kodieren von Parametern und das Zurückgeben von Werten im non-WSDL-Modus.

6.4. Web Services in PHP 4

Alternativen zu den Konzepten aus PHP5 bieten zwei nützliche Bibliotheken, welche hier kurz vorgestellt werden sollen. Zum einen wäre dies die SOAP-Implementierung im [PEAR]. Diese im Beta Stadium doch recht stabile Version zeichnet sich vor allem durch eine hohe Komplexität aus. Ebenso spricht es für die Implementierung, Teil des [PEAR] zu sein und somit einen gewissen Qualitätsanspruch zu besitzen. Neben dieser gibt es mit *NuSoap* eine recht ähnliche Bibliothek, welche zu den NuSphere Bibliotheken gehört, aber unabhängig ist und damit keine Abhängigkeiten, wie die PEAR-Implementierung aufweist. Für alle Anwender ohne PHP5 sind diese Bibliotheken sehr zu empfehlen. Durch die Standardisierung von SOAP sind sie kompatibel zu allen Web Services.

6.5. Probleme bei der Lösung mit PHP

Leider ist die Lösung mit PHP nicht so effizient, wie beispielsweise eine Web Service Implementierung mit Java. Es ist nämlich nicht möglich Aufrufe asynchron umzusetzen. Außerdem müssen sich Server und Client bei jeder Anfrage neu initialisieren und die WSDL-Beschreibung neu parsen. Dies führt zu einer höheren Latenzzeit und einer schlechteren Skalierbarkeit bei der Umsetzung mit PHP.

Die automatische Generierung von WSDL-Beschreibungen aus PHP-Code ist aufgrund der schwachen Typisierung der Sprache etwas schwierig. Es gibt jedoch in diesem Bereich erste Entwicklungen die mittels JavaDoc und Reflection gute Ergebnisse erzielen.

7. Herausforderungen

In der Literatur findet man häufig das Bild vom sogenannten *Web Service Protocol Stack*.

Discovery	UDDI (Itself a Web Service)
Description	WSDL, WSFL/XLANG, others to come
Access	SOAP, SOAP with attachments, XML-RPC, REST
Transfer	HTTP, SMTP, FTP, others
Transport	TCP/IP, UDP, others

Abbildung 39: Web Service Protocol Stack

Dieses Schichtenmodell ist zwar nicht so streng zu interpretieren wie die Schichtung bei Netzwerkprotokollen, aber es verdeutlicht die Abhängigkeiten zwischen den beteiligten Standards. Es zeigt aber beispielsweise nicht die enge Beziehung der oberen Schichten zu XML. Gerade die Verwendung von XML ist jedoch die Quelle eines Problems. Das XML Format kodiert nämlich die enthaltenden Informationen nicht besonders effizient und das führt zu erhöhtem Bandbreitenbedarf und vor allem zu höheren Latenzzeiten. Die folgende Gegenüberstellung soll ein Gefühl für die Zeitunterschiede vermitteln.

Im Vergleich zu einem lokalen Funktionsaufruf ist ein Aufruf zwischen verschiedenen Prozessen (IPC) zehn bis tausend mal langsamer. Ein Aufruf über Rechnergrenzen hinweg mit RPC-Mechanismen ist nochmals zehn bis tausend mal langsamer als ein Interprozessaufruf. Bei Web Service Aufrufen

über SOAP erhöht sich der Zeitbedarf zusätzlich um den Faktor zehn gegenüber traditionellen RPC-Technologien. (Quelle: [SZYPERSKI2002])

Gerade diese letzte Steigerung durch die Nutzung von SOAP bringt den Zeitbedarf eines Aufrufes in einen Bereich außerhalb des menschlichen Echtzeitempfindens. Das macht es nahezu unmöglich, in Echtzeit eine Kette von Web Services nacheinander aufzurufen, wenn ein menschlicher Nutzer eine sofortige Reaktion erwartet.

Ein möglicher Ansatz, die Performanz zu verbessern, wäre die Nutzung einer *Representational State Transfer* (REST) Architektur. Um einfacher und schneller auf Web Services zugreifen zu können nutz REST die Grundoperationen des HTTP Protokolls: GET, POST, PUT und DELETE. Das spart den Overhead von SOAP Nachrichten und soll zusätzlich die Entwicklung einfacher machen.

Probleme, die bereits bei traditionellen Ansätzen für entfernte Funktionsaufrufe, verteilte Objekte und Nachrichtenübertragung auftraten, zum Beispiel Authentifizierung, Verschlüsselung, Berechtigungen, Reflection, Routing, Nachrichtenweiterleitung und Sicherheit, müssen auch für Web Services gelöst werden. Dazu gibt es, wie in Abbildung 39 bereits erkennbar ist, gerade im Bereich der Web Service Beschreibung weitergehende Entwicklungen. Viele ergänzende Web Service Standards wie beispielsweise WS-Inspection, WS-License, WS-Referal, WS-Routing oder WS-Security adressieren diese zusätzlichen Aspekte.

8. Zusammenfassung

Ein Web Service ist eine Dienstleistung, die über das Internet von anderen Software-Systemen durch den Austausch standardisierter Nachrichten genutzt werden kann. Zusätzlich sind Mechanismen nötig, die es ermöglichen einen Web Service automatisiert aufzufinden und sich über dessen Schnittstellen zu informieren. Solche Anwendungen lassen sich mit den vorgestellten Technologien interoperabel realisieren. Die Interoperabilität entsteht durch die Nutzung offener Standards.

Im Falle von XML Web Services wird das SOAP-Protokoll für einen XML-basierten Nachrichtenaustausch verwendet. Mit der ebenfalls XML-basierten Web Services Description Language (WSDL) wird für Anbieter und Nutzer verbindlich die Schnittstelle eines Web Service beschrieben. Eine solche Beschreibung umfasst eine Liste der angebotenen Operationen, die Datentypen für Anfrage- und Antwortnachrichten, Informationen über das zu verwendende Kommunikationsprotokoll und die Adresse, unter welcher der Web Service erreichbar ist. Für das Publizieren und Lokalisieren von Web Services findet Universal Description, Discovery and Integration (UDDI) breite Anwendung. UDDI definiert eine Datenstruktur für Namensdienste, die via SOAP abgefragt werden können und damit selbst Web Services darstellen. Die sogenannten „White Pages“ enthalten Informationen über die registrierten Anbieter. In den „Yellow Pages“ finden sich die Beschreibungen eigentlichen Dienste. Um einen Web Service zu nutzen erhält man aus den „Green Pages“ die nötigen technischen Daten wie zum Beispiel die Adresse.

Mit PHP5 lassen sich XML Web Services einfach aber nicht sehr effizient implementieren. Die SOAP-Erweiterung von PHP5 enthält Klassen für Server und Client, die eine komfortable WSDL Unterstützung realisieren. Zum Abfragen von UDDI-Namensdiensten ist ein PEAR-Paket mit allen wichtigen Funktionen verfügbar.

Referenzen

[SZYPERSKI2002] Clemens Szyperski, Component Software - Beyond Object-Oriented Programming, Second Edition, 2002

[JAVAWS] David A. Chappell und Tyler Jewell, Java Web Services

[SOAP] SOAP Version 1.2, W3C Recommendation, 24. Juni 2003,
Part 0: Primer, *http://www.w3.org/TR/2003/REC-soap12-part0-20030624/*
Part 1: Messaging Framework, *http://www.w3.org/TR/2003/REC-soap12-part1-20030624/*
Part 2: Adjuncts, *http://www.w3.org/TR/2003/REC-soap12-part2-20030624/*

[ATTACH] SOAP messages with attachments, W3C Note, 11. Dezember 2000, *http://www.w3.org/TR/2000/NOTE-SOAP-attachments-20001211*

[WSDL-SOAP] Web Services Description Language (WSDL) 1.1, W3C Note, 15. März 2001, *http://www.w3.org/TR/2001/NOTE-wsdl-20010315*

[IBM] Uche Ogbuji, Using WSDL in SOAP applications, 1. November 2000, *http://www-128.ibm.com/developerworks/webservices/library/ws-soap/*

[UDDI] Offizielle Website des UDDI Projekts, *http://www.uddi.org*

[COOKBOOK] David Sklar und Adam Trachtenberg, PHP Cookbook, 2003

[PHP] PHP Dokumentation, *http://www.php.net/docs.php*

[PEAR] PHP Extension and Application Repository, *http://pear.php.net/*

[NICOLAI2005] Johannes Nicolai, Web Services and the SAP Web Application Server, 2005

[FIELDING2000] Roy Thomas Fielding, Architectural Styles and the Design of Network-based Software Architectures, 2000