3 Handwerkszeug: XML für Web Services

XML ist die Kerntechnologie für die Entwicklung der Web-Service-Technologie. Ein XML-Dokument macht jedoch noch keinen Web Service. Neben den reinen Erstellungs-, Einlesungs-, Speicherungs- und Darstellungstechniken wird für eine Web-Service-Applikation mindestens noch eine Technologie benötigt, um Daten auszutauschen. Hinzu kommt noch der Anspruch, dass die übertragenen Daten in standardisierter Form übertragen werden. Ein XML-Dokument sorgt noch nicht dafür, dass Daten, die von einem Rechner gesendet werden, auch von einem anderen korrekt interpretiert werden, so dass hierfür eine Möglichkeit der Standardisierung oder mindestens der Beschreibbarkeit gegeben sein muss. Web Services sind zudem recht unnütz, wenn sie auf einem Rechner installiert ihr Dasein fristen und nicht benutzt werden, da niemanden ihre Existenz bekannt ist. Für die Publikation solcher Services wird ein zentraler Dienst benötigt, der die Möglichkeit bietet, sich einen Überblick über die verfügbaren Services innerhalb einer Organisation zu machen.

In diesem Kapitel beschäftigen wir uns also mit den noch fehlenden Bausteinen, die man benötigt, wenn man eine Web-Service-basierte Architektur erstellen möchte.

3.1 XML-Remote Procedure Calls (XML-RPC)

XML-RPC ist eines der ersten XML-basierten Web-Service-Protokolle, zu der eine ansehnliche Anzahl an Implementierungen auf verschiedenen Betriebssystemen und Sprachen verfügbar ist. Mit XML-RPC können Funktionen auf einem anderen Rechner ausgeführt werden (RPC steht für Remote Procedure Call). Die Parameter, die die Funktion für ihre Arbeit benötigt, sowie der Rückgabewert werden als XML-Baum übertragen. Das zugrunde liegende Protokoll zur Übertragung ist HTTP. Im Vergleich zu späteren Ansätzen von Web Services ist das Protokoll noch recht einfach gehalten: Es gibt noch keine Meta-Beschreibung der übertragenen Daten.

Als Grundlage für die heutigen Web Services ist es sinnvoll, dass wir uns hier kurz dieses Protokoll ansehen, so dass wir die Erweiterungen in SOAP gegenüber dem Ur-Protokoll besser würdigen können. Tatsächlich war der Autor von XML-RPC auch bei der ersten Spezifikation von SOAP dabei; dabei hieß XML-RPC sogar vor der Veröffentlichung intern schon SOAP (siehe auch [Winer99b])!

Das XML-RPC-Protokoll dient ausschließlich dem Aufruf von Methoden und definiert die Form, in der Aufrufparameter eingepackt werden und wie das Resultat zurückgegeben wird. Die Spezifikation findet sich unter [Winer99a].

3.1.1 Aufrufprotokoll

Das Aufrufprotokoll für XML-RPC ist HTTP. Dabei wird in einem Request der Methodenaufruf geschickt, als Antwort erhält man das Ergebnis dieses Aufrufes. Der Aufruf erfolgt dabei über einen HTTP-POST Request (im Anhang A werden noch Details zum

HTTP-Protokoll beschrieben). Dabei wird das XML-Dokument einfach als Datenstrom übergeben, der den HTTP-Headern folgt. Nach der Spezifikation von XML-RPC müssen vier HTTP-Header richtig gesetzt sein: der Content-Type muss auf `text/xml` gesetzt sein, die Content-Length muss die Länge des XML-Requests enthalten (die Anzahl der Bytes). Weiterhin muss ein Host-Header angegeben werden (durch HTTP 1.1 gefordert) sowie der User-Agent. Ein Aufruf über HTTP kann also so beginnen:

```
POST /xmlrpc-app HTTP/1.1
Host: euklid.toxine.lan
User-Agent: Java-Client
Content-Type: text/xml
Content-length: 243

<?xml ...
```

3.1.2 Methodenaufruf

Ein Methodenaufruf wird in XML codiert per HTTP an den Server geschickt. Das Encoding des Dokumentes ist auf US-ASCII festgelegt und implizit angenommen, wenn es im XML-Deklarationsheader nicht angegeben ist. Das Format ist leicht verständlich; das folgende XML-Dokument ruft eine Methode `sayHello` auf der Serverseite auf und übergibt ihr einen einzelnen Parameter, den String "Julia":

```
<?xml version="1.0"?>
<methodCall>
   <methodName>sayHello</methodName>
   <params>
      <param>
         <value><string>Julia</string></value>
      </param>
   </params>
</methodCall>
```

Das XML-Dokument hat einen Hauptknoten `<methodCall>`, in dem der Unterknoten `<methodName>` den Namen der Methode angibt. Die nachfolgenden Parameter in dem `<params>`-Knoten enthalten eine Liste von Parametern `<param>`, die nun die einzelnen Werte enthalten können.

Wertetypen

Die Typen, die in einem `<param>`-Knoten enthalten sein können, erlauben die Abbildung üblicher skalare Datentypen sowie von Records und Arrays. Die Werte werden mit `<value>` eingeleitet, gefolgt von dem Datentyp als Tag codiert; in dem Beispiel oben ist das der Datentyp `String`. Weitere Datentypen sind in Tabelle 3.1 aufgeführt.

Bei diesen Datentypen gibt es einige Besonderheiten zu beachten, die eine Interoperabilität erschweren. So dürfen Strings nur als ASCII codiert sein und erlauben damit keine

Tag	Typ	Beispiel
<boolean>	boolescher Wert als Zahl 0 oder 1	0
<i4> oder <int>	vorzeichenbehafteter 32-Bit Integer	−22983
<double>	Fließkommazahl	−93.32821
<string>	Ein String in ASCII (nur 7-Bit Zeichen erlaubt).	Web Service
<dateTime.iso8601>	Ein Datum/Uhrzeit im ISO-8601 Format	20020324T12:22:16
<base64>	Ein binäre Bytefolge, kodiert in base64	Nh0fDKh9LeRo=
<array>	Eine Sequenz von <values>	
<struct>	Ein Record-Datentyp mit Name/Value mapping	

Tabelle 3.1: Datentypen für <value>s in XML-RPC

Sonderzeichen. Strings mit Sonderzeichen muss man grundsätzlich als binären String in base64-codierter Form verschicken. Beim Uhrzeitformat ist nicht spezifiziert, auf welche Zeitzone sie sich bezieht, so dass es Interpretationsmöglichkeiten in Richtung lokaler Zeit oder GMT gibt.

Komplexere Datentypen lassen sich über Structs und Arrays realisieren. Ein Struct hat dabei eine Menge von Membern, die jeweils einen Namen und einen Wert enthalten können:

```
<struct>
   <member><name>strasse</name>
           <value>
               <string>Wilhelmstrasse</string>
           </value>
   </member>
   <member><name>PLZ</name>
           <value><int>74072</int></value>
   </member>
   <member><name>ort</name>
           <value><string>Heilbronn</string></value>
   </member>
</struct>
```

Arrays schließlich können eine Sequence von `<values>` aufnehmen. Diese Werte können wieder von allen anderen Datentypen sein, d. h., es können darin auch wieder Arrays und Structs enthalten sein.

3.1.3 Rückgabewerte

Die Rückgabe eines solchen XML-RPC-Funktionsaufrufes ist wieder ein XML-Dokument. Dabei gibt es zwei Typen von Antworten: Rückgabe-Werte und Fehler-Rückgaben.

Rückgabewerte

Der Rückgabewert ist ähnlich wie beim Aufruf in einem `<params>`-Tag untergebracht. Dabei darf — analog des einzigen Rückgabewertes einer Funktion — nur genau ein einziger `<params>`-Knoten mit einem einzigen `<param>`-Element enthalten sein:

```
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
        <value><string>Hallo Julia</string></value>
      </param>
   </params>
</methodResponse>
```

Neben einem Rückgabewert, der wie hier einen Wert zurückgeben kann, kann der `<params>`-Knoten auch leer sein; dies entspricht einem `void`-Rückgabewert in Java. Der Wert kann einen beliebigen Typ haben, d. h., es können auch structs und Arrays zurückgegeben werden.

Fehler-Rückgabe

Schlägt die Methode fehl, dann kann alternativ zu dem `<params>`-Tag auch ein `<fault>`-Tag erscheinen, in dem ein Struct mit einer Fehlerbeschreibung zurückkommt. Dieser kann dazu benutzt werden, eine Exception beim Aufruf der Methode mitzuteilen:

```
<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value>
         <struct>
            <member>
               <name>faultCode</name>
               <value><int>4711</int></value>
            </member>
            <member>
               <name>faultString</name>
               <value>
                 <string>java.lang.IOException</string>
               </value>
```

```
            </member>
          </struct>
        </value>
      </fault>
    </methodResponse>
```

3.1.4 Implementierung

Die HTTP-Serverimplementierung hat nur begrenzte Anforderungen, denn es muss mit `POST` nur ein einziger HTTP-Request-Typ gehandhabt werden. Die garantierte Übergabe der Länge des Requests macht das Erkennen des Endes des Datenstroms einfach.

Aus diesen Gründen ist eine Implementierung mit Bordmitteln von Programmierumgebungen, die mindestens das Öffnen eines Server-Sockets erlauben, relativ einfach möglich. Unter Java kann man damit in wenigen Zeilen einen entsprechenden Server implementieren. Steht ein Web-Application-Server wie der Tomcat-Server zur Verfügung, kann man auch ein Servlet implementieren, das in der `doPost()`-Methode über den `ServletInputStream` die Anfrage entgegennimmt, bearbeitet und die Antwort auf den `ServletOutputStream` schickt.

Das Parsen und Erzeugen der XML-Strukturen ist so einfach, dass man auch ohne speziellen XML-Parser eine Implementierung wagen kann.

Die Implementierung eines Clients ist fast noch einfacher. Dieser muss nur in der Lage sein, einen Socket zu dem Server zu öffnen; das Protokoll zu implementieren ist aufgrund der wenigen Besonderheiten recht einfach.

Als Folge dieser Einfachheit gab es sehr schnell viele Implementierungen von Servern und Clients in verschiedenen Sprachen und führte zu dem, was die eigentliche Intention des Protokolls gewesen ist: eine einfache plattformunabhängige Integration von Systemen.

3.1.5 Diskussion

Die XML-RPC-Spezifikation erlaubt eine einfache Abbildung von Remote Procedure Calls. Dabei ist das Protokoll so einfach gehalten, dass es schnell implementiert werden kann, aber trotzdem die wesentlichen Anforderungen abdeckt, die bei RPCs auftreten. Die Beschränkung auf ein einziges Übertragungsprotokoll (HTTP) und einer Betriebsart (RPC) vereinfacht die Implementierung, weil keine Sonderfälle zu beachten sind. Die Wahl des Protokolls HTTP passt sehr gut auf den Anwendungszweck der Remote Procedure Calls, denn das Wesen von HTTP als Request/Response-Protokoll findet sich in den Anforderungen an ein RPC-Protokoll genau wieder. Die Antworttypen mit Rückgabewert und Fehlerrückgabe entsprechen den Anforderungen Exception-unterstützender Programmiersprachen. Da HTTP als wichtiges Protokoll des WWW sehr gut durch Firewalls oder Proxies zu tunneln ist, gibt es wenige Probleme auf der unteren Kommunikationsschicht, selbst wenn ein Client hinter solchen Schutzvorrichtungen auf Dienste im Internet zugreifen möchte.

Die Beschränkung auf das HTTP-Protokoll und die RPC-Betriebsart lassen möglicherweise noch Wünsche offen, wenn Anwendungsbereiche abgedeckt werden sollen, die nicht als Remote Procedure Call abgewickelt werden sollen. Sessionbehaftete Aufrufe, bei der ein Client immer Methoden auf derselben Instanz eines Objektes auf der Serverseite aufruft, sind nicht spezifiziert, können aber leicht durch übliche Techniken wie Cookies erreicht werden.

Das Protokoll in dieser Spezifikation kommt recht schnell an seine Grenzen, wenn es in größerem Umfeld bei der Interaktion komplexer Systeme eingesetzt werden soll. Die Gründe dafür lassen sich sowohl in den Datentypen finden als auch in der fehlenden Meta-Beschreibung. Die Probleme der Datentypen sind:

- Die Codierung der Datentypen lassen Interpretation zu oder sind zu ungenau.
  Das Datumformat legt sich nicht auf eine Zeitzone fest. Die Stringcodierung lässt nur das ASCII-Format zu. Strings mit Umlauten oder anderen Sonderzeichen müssen als Binärstring codiert werden, was unnötig aufwändig ist. Auf der anderen Seite vereinfacht diese Vorgehensweise natürlich die Implementierung der Server und Clients, wenn sie sich nicht um verschiedene Encodings kümmern müssen.
- Zyklische Datenstrukturen lassen sich nicht abbilden. Die Datenstruktur, die als XML-Dokument übertragen wird, ist ein Baum und kann daher keine zyklischen Beziehungen abbilden. Eine Referenzierung über ID und IDREF-Attribute ist nicht vorgesehen (s. auch »Eindeutige Bezeichner« auf Seite 60).
- Binäre Daten lassen sich nur über ein base64-Encoding übertragen. Beim base64-Encoding werden die binären Daten in Text überführt, so dass sie sauber als ASCII übertragen werden können. Das ist allerdings aufwändig, denn diese muss auf der einen Seite eingepackt und auf der anderen Seite wieder ausgepackt werden; zudem bläht es die Datenmenge um rund 30 Prozent auf.

Ein besonders negativer Punkt an diesem Protokoll ist, dass es keine Meta-Beschreibung der Aufrufe gibt. Alle RPC-Protokolle bieten eine abstrakte Beschreibung der Parameter und Aufrufkonventionen, meist in einer Form, die dem Java Interface ähnelt: eine Sammlung der verfügbaren Methodennamen, ihrer Parameter und Rückgabewerte. Die abstrakte Beschreibung wird bei den RPC-Protokollen verwendet, um für die Zielplattform entsprechenden Sourcecode zu generieren.

Stellt man einen XML-RPC-Dienst zur Verfügung, muss man zwangsläufig diesen mit natürlichsprachlichen Beschreibungen und vielleicht einem Beispielaufruf der veröffentlichten Methoden versehen. Dies bedingt manuelles Eingreifen bei der Implementierung, geringere Flexibilität bei Änderungen und behindert einen automatischen Aufruf oder Browsing des entsprechenden Dienstes. Auch ist ein Namensdienst für XML-RPC-Dienste damit nicht sehr sinnvoll, denn ein gefundener Dienst lässt sich nur dann richtig ansprechen, wenn eine entsprechende maschinenlesbare Beschreibung vorliegt.

Von XML-RPC zu SOAP

Aufgrund des richtigen Ansatzes, den XML-RPC schon zeigt, wurden Bemühungen unternommen, die noch verbliebenen Nachteile in einer erweiterten Spezifikation zu adressieren. Dabei landete die Spezifikation beim W3C, und es wurde eine Vielzahl von möglichen Low-Level-Protokollen, Datentypen und Betriebsarten hinzugefügt — das SOAP-Protokoll. Dies löst alle oben genannten Probleme und noch einige mehr[1].

3.2 Simple Object Access Protocol (SOAP)

3.2.1 Einleitung

Das SOAP-Protokoll ist eine Erweiterung der Ideen des XML-RPC-Protokolls. Die Nachteile dieses Protokolls wurden dabei angegangen, und die fehlenden Bestandteile hinzufügt. Eine Spezifizierung des SOAP-Protokolls der aktuellen Version 1.1 findet sich unter [W3C00h].

3.2.2 Zielsetzung

Die Zielsetzung von SOAP ist die Verallgemeinerung des Austausches von Daten über XML. Während XML-RPC nur für den Aufruf von Methoden gedacht ist, und damit ein Anfrage- und Antwort-Dokument impliziert, soll SOAP auch die Möglichkeit bieten, dass ein Dokument nur verschickt wird, ohne dass eine Antwort erwartet wird.

Die Beschränkung auf spezielle Dokumente zum Methodenaufruf soll einer allgemeinen Möglichkeit weichen, beliebige Dokumente zu verschicken — die den Fall des Methodenaufrufes einschließt.

Die Erweiterungen sollen zudem ermöglichen, dass die Beschränkung auf das HTTP-Protokoll aufgehoben wird und prinzipiell beliebige Transportwege offen stehen.

3.2.3 SOAP-Envelope

SOAP erweitert die Idee des XML-RPC um weitere Informationen, die bei der Übertragung der Message mitgegeben werden sollen. In anderen Protokollen wie z. B. HTTP (siehe A) werden Zusatzinformationen in Headern mitgegeben, und diese Idee wurde auch in SOAP aufgenommen. SOAP ist ein XML-Protokoll, daher müssen die Headerinformationen zu den Nutzdaten in XML verpackt werden. Für diesen Zweck wurde ein *Umschlag* (engl. envelope) entworfen, der beide Informationen aufnimmt. Ein SOAP-Umschlag sieht folgendermaßen aus:

1 Böse Zungen behaupten, dass sogar *zu viele* Probleme gelöst wurden und damit die Spezifikation unnötig aufgebläht wurde.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope
     xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
   <soap-env:Header>
          Header-Information
   </soap-env:Header>
   <soap-env:Body>
          Nutzdaten
  </soap-env:Body>
</soap-env:Envelope>
```

Der Umschlag, der im XML-Namespace *http://schemas.xmlsoap.org/soap/envelope/* `Envelope` heißt, trägt zwei Elemente, `Header` und `Body`, ebenfalls in diesem Namespace. Die Header enthalten Informationen, die von den Kommunikationspartnern ausgetauscht werden, aber nicht direkt etwas mit den Nutzdaten zu tun haben. Ähnlich wie bei HTTP betrifft das Informationen, die Aspekte der Nutzdaten beschreiben.

Die Aufnahme der Header-Informationen in das SOAP-Protokoll selbst ermöglicht, dass SOAP damit unabhängig von Übertragungsrpotokollen wird, die sonst entsprechende Meta-Informationen tragen helfen würden — die Beschränkung auf das HTTP-Protokoll ist damit aufgehoben und es können beliebige Protokolle eingesetzt werden, die XML-Dokumente als Nutzdaten akzeptieren.

Der Body enthält die eigentlichen Nutzdaten. Diese Nutzdaten können, ähnlich wie bei XML-RPC, einen Methodenaufruf codieren, oder aber komplette XML-Dokumente enthalten. Dies bedeutet, dass die zu übertragenen Nutzdaten in XML vorliegen müssen, was für viele verschachtelte Strukturen eine gute Repräsentation ist. Große binäre Datenströme eignen sich dafür nicht, denn ein Verpacken in XML erfordert in diesem Fall, dass es als base64[2]-Datentyp übertragen werden müsste.

3.2.4 Attachments

Attachments bieten die Möglichkeit, große binäre Daten zu übertragen. Der Hintergrund ist, dass eine Übertragung innerhalb des XML-Dokumentes als base64-codierter Strings sehr aufwändig bei der Verarbeitung ist: die Daten müssen verpackt und auf der Gegenseite von dem XML-Parser zerlegt und dann wieder entpackt werden. Es kommt hinzu, dass das Verpacken der Daten mittels base64 die Datenmenge um etwa 30 % erhöht. Diese binären Daten kann man auch als Attachments verpacken ([W3C00i]). Für diesen Ansatz wurde kein neues Protokoll entwickelt, sondern einfach bestehende Standards verwendet. Der hier verwendete Standard ist der MIME-Standard, eine Erweiterung zum E-Mail-Standard, der ermöglicht, eine Nachricht aus verschiedenen Teilen zusammenzusetzen.

2 Eine Codierung der 8 Bits unter Verwendung druckbarer Zeichen. Diese ist ein Set aus 64 Zeichen. Da damit ein übertragenes Zeichen nur 6 Bit Information überträgt, ist ein Datenstrom in base64-Encoding um etwa 30 % aufgebläht.

MIME

MIME steht für *Multipurpose Internet Mail Extensions* und war zunächst als eine Erweiterung für das Format, in dem Mails verschickt werden, entworfen [RFC2045]. Die ursprüngliche Mail sieht nämlich nur einfachen ASCII-Text für den Inhalt von Mails vor — die Benutzung verschiedene Encodings, die es auch erlauben, Umlaute zu übertragen, oder der Versand von Attachments wurden erst mit diesen Erweiterungen möglich. Einer der Erweiterungen von MIME sieht zusätzliche Header vor, die im Mail-Kopf stehen. Ein wichtiger MIME-Header ist der Content-Type (Inhaltstyp), der den Inhalt der Nachricht beschreibt. Ein Content-Type für einfachen Text ist:

```
Content-Type: text/plain
```

Die Ausprägung eines Content-Type wird *MIME-Type* genannt und ist immer aus zwei Teilen zusammengesetzt, die mit einem Schrägstrich (›/‹) getrennt sind. Der erste Teil bezeichnet den Oberbegriff, der zweite eine Spezialisierung. Ein HTML-Dokument hat den MIME-Type »`text/html`«, ein XML-Dokument »`text/xml`«. Neben dem Oberbegriff »`text`« gibt es noch weitere, wie »`image`« oder »`application`«.

Spezielle Parameter zu dem MIME-Type können, mit Semikolon getrennt, in der Form `<name>=<wert>` übergeben werden. Ein häufiger Einsatz dieser Möglichkeit bei Texten ist hierbei die Angabe der verwendeten Zeichentabelle:

```
Content-Type: text/html; encoding="ISO-8859-1"
```

Wichtige Basis-MIME-Typen sind in [RFC2046] definiert, aber weitere können für neue Anwendungen jederzeit dazu kommen; dafür ist ein entsprechendes Vorgehen bei der Registrierung vorgesehen ([RFC2048]).

Die MIME-Header sind eine Erweiterung der normalen Mail-Header ([RFC822]), für die festgelegt ist, dass nach einer Liste von Headern (Header-Part) der eigentliche Inhalt der Nachricht (Body-Part) nach einer Leerzeile folgt.

Multipart-MIME

Die Erweiterung, die sich unter anderem mit Attachments für Mails beschäftigt [RFC2046], erlaubt, dass eine Nachricht aus mehreren Teilen zusammengesetzt sein kann, die wieder automatisch getrennt werden können. Bei diesen so genannten Multipart-Messages werden die einzelnen Teile durch eine eindeutige Trennzeile auseinander gehalten. Die Trennzeile wird am Anfang der Message durch einen speziellen Content-Type vereinbart.

Der Inhaltstyp einer Mail, die Attachments verschickt, ist `multipart/mixed`. Die Zeile, die den Content-Type setzt, definiert dabei die Trennzeile:

```
Content-Type: multipart/mixed; boundary=eine-Trennlinie
```

Die einzelnen Teile der Message beginnen dabei immer mit dieser Trennzeile; dabei müssen die Trennzeilen mit zwei Bindestrichen beginnen, gefolgt von der im Content-Type-Header vereinbarten Boundary. Jeder Teil beginnt dann zunächst wieder mit Headern, die spezifisch für den Teil sind (z. B. der Content-Type), gefolgt von einer leeren Zeile. Die letzte Trennzeile endet mit zwei Bindestrichen, um das Ende zu kennzeichnen:

```
--eine-Trennlinie
Content-Type: text/plain

Eine Mail
--eine-Trennlinie
Content-Type: image/jpeg

...
--eine-Trennlinie--
```

Das Programm, das eine Multipart-Message erzeugt, muss natürlich darauf achten, dass die Trennlinie nicht auch zufällig in dem Text enthalten ist; daher besteht sie normalerweise aus Zufallszeichen und nicht aus einem einfachen Text wie in diesem Beispiel.

Diese mehrteiligen Nachrichten lassen noch keine eindeutige Zuordnung erkennen, welches die Haupt-Nachricht ist, und welche sich darauf beziehen. Dies ist bei Attachments in Mails auch nicht so notwendig, aber für die Übertragung von HTML-Mails, in der z. B. Bilder enthalten sind, ist es notwendig, eine klare Definition des Hauptdokumentes und der abhängigen Dokumente zu haben; weiterhin muss klar sein, wie man in der Haupt-Nachricht sich auf die anderen bezieht.

Dieses Problem wurde in [RFC2387] adressiert, einer Erweiterung von Multipart-Messages, indem es einen neuen Content-Type einführt, `mulitpart/related`. Dieser sieht vor, dass ein Start-Teil definiert wird, der durch eine eindeutige ID gekennzeichnet ist. Alle Teile der Nachricht müssen dabei eine eindeutige ID haben, damit die Hauptnachricht sich darauf beziehen kann. Die ID eines MIME-Parts wird dabei durch einen neuen Header, `Content-ID`, definiert. Neben einer `Content-ID` kann ein MIME-Part noch einen weiteren Header, die `Content-Location`, enthalten. Die ID des Start-Teils wird in dem Content-Type Header, in dem auch die Trennlinie vereinbart ist, als weiteres Attribut angegeben. Ist sie nicht angegeben, gilt automatisch der erste Teil als Startteil:

```
Content-Type: multipart/related; type="text/xml";
    boundary=eine-Trennlinie; start=id4711

--eine-Trennlinie
Content-Type: text/xml
Content-ID: id4711

<?xml version='1.0' ....
--eine-Trennlinie
Content-Type: image/jpeg
```

```
Content-ID: myImage
Content-Location: einbild.jpeg

...
--eine-Trennlinie--
```

Nun ist klar, mit welchem Haupt-Dokument die Nachricht beginnt, und welches die abhängigen Teile sind.

Für HTML-Mails wird dieses System genutzt, um alle notwendigen Elemente einer Mail (Text, Bilder) in einer Nachricht unterzubringen. Dabei wird in Verweisen innerhalb des HTML-Dokumentes (wie `<a href="..">` oder `<img src="...">`) entweder der Name der Content-Location angegeben oder der Name der Content-ID, der ein »cid:« vorangestellt ist ([RFC 2557]). Im obigen Beispiel kann von dem Hauptdokument auf das angehängte Bild daher über ein `<img src="cid:myImage">` oder `<img src="einbild.jpeg">` verwiesen werden.

Diese Konvention des Verweises auf andere Teile einer multipart/related-Nachricht hat man in [W3C00i] für SOAP-Messages mit Attachments übernommen. Dabei ist das Haupt-Dokument der SOAP-Part, der sich auf andere Teile beziehen kann. Damit ist es möglich, dass man in dem XML-Teil einen Verweis auf ein später folgendes Attachment unterbringen kann. Dies ist ein schönes Beispiel, wie schon vorhandene Spezifikationen sinnvoll in einem neuen Kontext verwendet werden können.

Multiparts mit DIME

Es bleibt anzumerken, dass es zurzeit gerade ein IETF-Working Paper gibt, eingereicht von Microsoft und IBM, bei dem an einer anderen Spezifikation für Dokumente aus mehreren Elementen gearbeitet wird: DIME. Die Direct Internet Multipart Encapsulation sieht eine Trennung der Teile über binäre Header vor, die die Länge des nachfolgenden Teils enthalten. Daher lässt sich ein eingehender Datenstrom schneller parsen, denn es muss nicht mehr wie im Fall der MIME-Multipart-Teile nach den Trennlinien gesucht werden, denn es genügt einfach, die angegebene Anzahl von Bytes einzulesen, was weniger Aufwand bei der Verarbeitung einer solchen Message bedeutet. Der Vorteil kann sein, dass die Verarbeitung etwas schneller geht. Der Nachteil einer solchen Codierung liegt auf der Hand: die binären Header wird man nicht ohne Werkzeuge lesen können, was einen wichtigen Punkt der Web Services — Wartbarkeit — erschwert. Auch eine Modifikation eines Teils, z. B. eine Neuformatierung eines XML-Dokuments zur besseren Lesbarkeit bei manuellen Versuchen, ist damit erschwert: die Länge des Teils stimmt dann natürlich nicht mehr mit dem Header überein.

3.3 Ein einfacher Web Service

Nach all der Theorie von XML, DOM, SOAP usw. ist es nun an der Zeit, den ersten Web Service zu implementieren. Unser erstes praktisches Beispiel besteht aus einem einfachen HTTP-Client, der eine SOAP-Anfrage an einen Applikationsserver stellt und eine

Antwort zurück erwartet. Der entsprechende Web Service soll lediglich die Anfrage des Clients ausgeben, um zu zeigen, dass die Kommunikation zwischen beiden Komponenten funktioniert. Als Server verwenden wir den *Apache Jakarta Tomcat*-Servlet-Container, der dazu dient, HTTP-Servlets zu verwalten und ihnen über HTTP eine Schnittstelle nach außen bietet. Der Client ist eine einfache Java-Applikation.

Abbildung 3.1 veranschaulicht die Architektur.

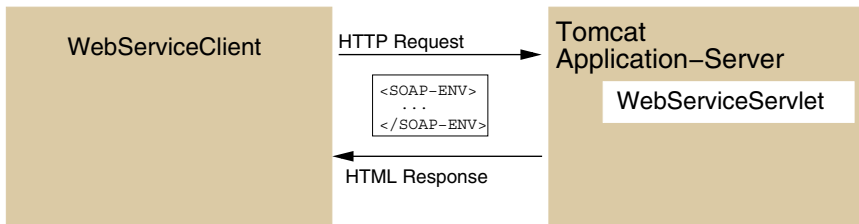

Abbildung 3.1: Architektur einer einfachen Web-Service-Applikation

Implementierung des Server-Servlets

Wir betrachten zunächst die Server-Komponente. Sie besteht aus einem Java-Servlet, das einen HTTP-basierten Dienst nach außen beschreibt. Die Programmierschnittstelle hierzu ist die Java Servlet-API, die von vielen Web- und Applikationsservern implementiert wird.

Servlets besitzen eine `doGet()`- und `doPost()`-Methode, die dazu dienen, eine Post- oder Get-Anfrage von außen entgegenzunehmen und eine entsprechende Antwort zurückzusenden. Die Anfrage wird durch die Klasse `javax.servlet.http.HttpServletRequest` gekapselt, die Antwort durch `javax.servlet.http.HttpServletResponse`. Als Basisklasse für selbst entwickelte Servlets dient `HttpServlet`.

```
import java.io.IOException;
import java.io.BufferedReader;
import javax.servlet.*;
import javax.servlet.http.*;

public class WebServiceServlet extends HttpServlet {
  public void doGet(HttpServletRequest request,
                HttpServletResponse response)
    throws IOException,ServletException
  {
    doPost(request,response);
  }

  public void doPost(HttpServletRequest request,
                 HttpServletResponse response)
    throws IOException,ServletException
```

```
  {
    System.out.println("Receive Request");
     if (request.getContentLength() > 0) {
      try {
        BufferedReader reader=request.getReader();
        String line=null;
        do {
          line=reader.readLine();
          if (line != null) {
            System.out.println(line);
          }
        }
        while(line!=null);
      }
      catch (Exception e) {
        e.printStackTrace();
      }
    }
    response.setContentType("text/xml");
  }
}
```

Listing 3.1: Ein einfacher Empfänger als Servlet

In unserem Beispiel erweitert das `WebServiceServlet` das `HttpServlet` und überschreibt die Methoden `doGet()` und `doPost()`. Der eigentliche Code befindet sich in der `doPost()`-Methode, wo zunächst die Anfrage des Clients über den Eingabestrom des Requests gelesen und dann ausgegeben wird. Per Definition muss jedes Servlet eine Antwort zurückgeben, so dass wir nach erfolgreichem Lesen der Anfrage eine leere Antwort mit dem Content-Type `text/xml` zurückliefern.

Implementierung eines HttpClients

Der Web Service Client dient dazu, eine SOAP-Anfrage zu generieren und an den Web Service zu senden. Unser Beispiel besteht im Wesentlichen aus zwei Methoden, die das Senden und Empfangen von SOAP-Dokumenten über HTTP ermöglichen. Beide verwenden die Konstante `TARGET_URL`, um den Web Service zu adressieren.

Die Methode `sendMessage()` konstruiert einen SOAP-Umschlag und einen leeren SOAP-Body, der in den Umschlag integriert wird. Nachfolgend wird ein `Message`-Objekt zur Übertragung instanziert, die den gesamten SOAP-Envelope an die Ziel-URL verschickt. Das Protokoll und die Zieladresse werden hierbei aus der Ziel-URL entnommen, so dass unterschiedliche Protokolle verwendet werden können. In unserem Fall wird HTTP verwendet.

Die `receiveMessage()`-Methode funktioniert im Wesentlichen wie im Servlet, mit dem Unterschied, dass hierbei das `Message`-Objekt und nicht der `Request`-Eingabestrom zum Auslesen verwendet wird.

```
import java.io.*;
import java.net.*;
import org.xml.sax.*;
import org.apache.soap.*;
import org.apache.soap.messaging.*;
import org.apache.soap.transport.*;

public class WebServiceClient {
  private static final String TARGET_URL=
    "http://localhost:8080/webserviceexample/"
    +"servlet/WebServiceServlet";
  private static final String URI=
    "urn:addison-wesley-webservice-example";

  public Message sendMessage() {
    System.out.println("Anfrage senden...");
    Envelope env = new Envelope();
    Body body = new Body();
    env.setBody(body);

    Message msg = new Message();
    try {
      msg.send(new URL(TARGET_URL),URI,env);
    }
    catch (Exception e) {
      e.printStackTrace();
    }
    return msg;
  }

  public void receiveMessage(Message msg) {
    System.out.println("Warten auf Antwort...");
    SOAPTransport st = msg.getSOAPTransport();
    try {
      BufferedReader br = st.receive();
      String line = br.readLine();
      while (line != null) {
        System.out.println(line);
        line = br.readLine();
      }
    }
    catch (Exception e) {
```

```
        e.printStackTrace();
      }
  }

  public static void main(String args[]) {
    WebServiceClient wsClient=new WebServiceClient();
    Message msg=wsClient.sendMessage();
    wsClient.receiveMessage(msg);
  }
}
```

Listing 3.2: Ein einfacher Web Service HTTP-Client

Bibliotheken

Für die Ausführung unseres ersten Web Services werden einige externe Pakete benötigt, die in verschiedenen Bibliotheken untergebracht sind. Tabelle 3.2 zeigt die verschiedenen Bibliotheken, die daraus verwendeten Pakete und ihre Funktion.

Bibliothek	Paket	Funktionen
servlet.jar	javax.servlet	Java Servlet API (JSDK)
sax2.jar	org.xml.sax	Simple API for XML (SAX)
soap.jar	org.apache.soap	Simple Object Access Protocol API (SOAP)

Tabelle 3.2: Verwendete Bibliotheken für den ersten Web Service

Installation des Tomcat-Applikationsservers

Wie bereits erwähnt, verwenden wir den Tomcat-Applikationsserver zur Ausführung des Servlets. Tomcat ist ein Open-Source-Projekt, das die jeweils aktuelle Referenzimplementierung der Java Servlet- und Java Server Pages-Implementierung von SUN enthält (vgl. jcp *http://www.jcp.org*). Installationsversionen für alle gängigen Plattformen finden sich auf der Homepage des Tomcat-Projektes (vgl. tomcat *http://jakarta.apache.org/tomcat/index.html*). Bei Unix- oder Unix-ähnlichen Systemen findet die Installation durch das Auspacken eines Archivs statt, für Windows existiert ein recht komfortables Installationsprogramm.

Nach der Installation lässt sich das System über die Skripte

`$CATALINA_HOME/bin/startup.sh` bzw.

`$CATALINA_HOME/bin/shutdown.sh` (unter Unix) bzw.

`%CATALINA_HOME%/bin/startup.bat` bzw.

`%CATALINA_HOME%/bin/shutdown.bat` starten bzw. stoppen.

Nach dem Start stellt Tomcat einen lokalen HTTP-Service auf Port 8080 zur Verfügung, der sich über einen Webbrowser über `http://localhost:8080/` kontaktieren lässt. Die Seite sollte dann Abbildung 3.2 entsprechen.

Abbildung 3.2: Tomcat HTTP-Startseite

Kompilieren und Deployen mit Ant

Wir haben bis jetzt ein Web Service-Server- und -Client-Programm entwickelt und einen Applikationsserver für das Server-Programm aufgesetzt. Nun ist es an der Zeit, ein Konstrukt zu entwickeln, das es ermöglicht, die einzelnen Systemteile effizient zu kompilieren und zu installieren (deployen).

Apache Jakarta Ant

Als Projektverwaltungswerkzeug, das alle diese Zwecke erfüllt, verwenden wir *Apache Jakarta Ant* (vgl. ant *http://jakarta.apache.org/ant/index.html*). Einfach ausgedrückt stellt Ant das Pendant zum C- bzw. C++-Build-Werkzeug *GNU make* (vgl. GNU make *http://www.gnu.org/software/make/make.html*) dar. Es dient dazu, eine Projektstruktur zu definieren, diese automatisiert zu kompilieren und die entsprechenden Ergebnisse an die richtige Stelle zu installieren. Im Gegensatz zu make, das ein eigenes Eingabeformat zur Projektbeschreibung benutzt, verwendet Ant eine XML-basierte Struktur. Diese XML-Datei wird in aller Regel als `build.xml` abgelegt.

Build-Dateien

Eine Build-Datei besteht im Wesentlichen aus fünf Elementen, die sich hierarchisch einordnen lassen. Die unterste Ebene hierbei stellt das `property`-Tag, das ein Name-Wert-Paar darstellt und somit dazu dient, übergreifende Eigenschaften des Systems zu definieren. Dies sind typischerweise Verzeichnisnamen oder Umgebungsvariablen.

Im Gegensatz dazu dient das `project`-Tag zur Kapselung aller Elemente, die zu einem gesamten Projekt gehören. Zu diesen Elementen gehören die `target`- und `task`-Tags. Das `target`-Tag dient zur Festlegung eines Entwicklungszieles wie z. B. `compile` oder `deploy` zum Kompilieren oder Installieren eines Programms.

Zudem existiert das `task`-Tag zur Referenzierung und Definition von Kommandozeilen-Skripten, die bei Bedarf ausgeführt werden können und zum Aufruf externer Werkzeuge dienen. Dies macht Ant-Build-Dateien sehr flexibel einsetzbar.

Außerdem existieren verschiedene Elemente zum Ausdruck von Pfadstrukturen. Hierzu gehören die Tags `dirset`, `fileset`, `filelist`, die dazu dienen, Verzeichnis- und Dateimengen sowie geordnete Dateilisten zu definieren. Dies ist besonders wichtig für die Festlegung der Dateien, die in den Compile- und Deploy-Prozess einbezogen werden sollen.

Abbildung 3.3 zeigt die Gesamthierarchie innerhalb einer Build-Datei.

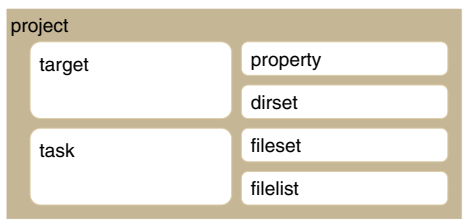

Abbildung 3.3: Struktur einer Ant-Build-Datei

Projektstruktur des ersten Web Service

Betrachten wir einmal die Struktur unseres Projektes. Wir benutzen im Wesentlichen zwei Bibliotheken, die uns die Servlet-API und die SOAP-API zur Verfügung stellen (`servlet.jar` und `soap.jar`), sowie zwei Quelldateien namens `WebServiceClient.java` und `WebServiceServlet.java` (siehe Abbildung 3.4). Hierbei liegen die Bibliotheken im Unterverzeichnis `lib`, während der Quellcode im Unterverzeichnis `code` liegt. Viele Entwickler benutzen statt des Verzeichnisnamens `code` auch gerne `src`. Die beiden Quellcode-Dateien sollen kompiliert werden und auf eine einfache Art und Weise in den Servlet-Container deployed werden können. Hierbei spielt für den Servlet-Container eigentlich nur das WebServiceServlet eine wirkliche Rolle, da diese Klasse die einzige ist, die das Interface `HTTPServlet` implementiert und somit auch die einzige Klasse ist, die innerhalb des Containers ausgeführt werden kann. Auf oberster Verzeichnisebene des Projektes liegt die Build-Datei, die angibt, wie das Projekt kompiliert, paketiert und deployed werden soll.

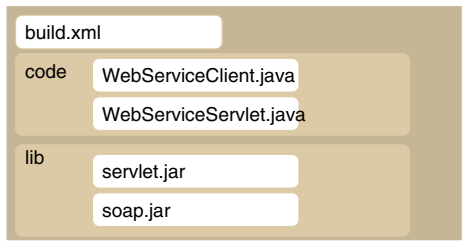

Abbildung 3.4: Schematische Darstellung der Projektstruktur

Build-Datei für den ersten Web Service

Nachdem wir uns über die Projektstruktur im Klaren sind, müssen wir lediglich noch eine Build-Datei dafür erzeugen. Betrachten wir zunächst die Aufgaben (Targets), die in dieser Build-Datei realisiert sein sollten. Üblicherweise legt man innerhalb jedes build.xml ein Target zum Vorbereiten des Compilierungsvorgangs namens `prepare` sowie eines zum Löschen der Dateien, die beim Kompilieren entstanden sind (`clean`), an. Neben dem Erzeugen von Verzeichnissen und Löschen von Verzeichnissen wollen wir in unserem Beispiel eine war-Datei erstellen, die die kompilierten Klassen und Bibliotheken unseres gesamten Projekts enthält. Aus diesem Grund sollten wir sowohl für das eigentliche Kompilieren, für das Erstellen einer Distribution sowie für das Erstellen des eigentlichen war-Archivs entsprechende Tasks schreiben (`compile`, `dist`, `war`). Eine Distribution ist hierbei ein Verzeichnisbaum, in dem alle Dateien in der Form enthalten sind, wie sie später in der war-Datei benötigt werden. Neben den bisherigen fünf Targets ist es sehr praktisch, wenn man eine Methodik erstellt, die die Distribution auf Kommando auf den Applikationsserver kopiert (`deploy`).

Die für unseren ersten Web Service verwendete Build-Datei soll also insgesamt sechs Targets beinhalten, die in Tabelle 3.3 zusammengefasst sind.

Target	Zweck
`prepare`	Vorbereiten des Compiliervorgangs (Anlegen der Compile-Verzeichnisse, kopieren von Konfigurationsdateien usw.)
`compile`	Kompilieren der Applikation
`dist`	Erstellen der Distribution (Zusammenführen aller Dateien für das System in eine Verzeichnisstruktur)
`war`	Verpacken der Distribution in eine .war-Datei
`deploy`	Kopieren der `.war`-Datei in den Applikationsserver
`clean`	Löschen des Distributionsverzeichnisses

Tabelle 3.3: Targets im build.xml für den ersten Web Service

Machen wir uns nun an die Implementierung. Zunächst einmal ist es sinnvoll, eine Reihe von unter Umständen aufeinander aufbauender Verzeichnisnamen sowie sonstige feste Werte als Konstanten über das `property`-Tag zu definieren. Wir definieren hierbei explizit

die Position der Quelldateien (über das Property `src`) und der Distribution (über das Property `dist`). Jede Servlet-basierte Applikation, die als war-Datei deployed werden soll, legt Bibliotheken und kompilierte Klassen in dem Verzeichnis `WEB-INF` ab, so dass es Sinn macht, hierfür eine eigene Konstante `dist.home.webinf` festzulegen.

Um Zugriff auf die Umgebung (Environment), also auf Variablen zu bekommen, die außerhalb der Build-Datei, zum Beispiel innerhalb des Kommandozeileninterpreters oder im Betriebssystem selbst gesetzt worden sind, wird die Variable `ENV` benutzt, die über die Definition `<property environment="ENV"/>` bereitgestellt wird. Wir machen uns dies zu Nutze, um die Konstante `deploy.home` zu setzen, die auf Basis der Umgebungsvariable `CATALINA_HOME` definiert wird. Diese Variable zeigt dann auf das Verzeichnis, in das die erzeugte war-Datei beim Deployment kopiert werden soll:

```
<property name="deploy.home"
 value="${ENV.CATALINA_HOME}/webapps/"/>
```

Neben diesen Konstanten erzeugen wir eine Liste der verwendeten Bibliotheken durch das `fileset`-Tag, die beim späteren Kompilieren und Zusammenstellen der Distribution verwendet werden.

Nach diesen vorbereitenden Maßnahmen können wir nun endlich zu den tatsächlichen Targets der Build-Datei vorstoßen. Hierbei legt `prepare` zunächst alle erforderlichen Verzeichnisse für die Distribution über das `mkdir`-Tag an. Das Target `compile` ist in unserem Beispiel recht bescheiden ausgefallen. Es benutzt den `javac`-Task, um die in `code` liegenden Java-Quelldateien unter Zuhilfenahme der Bibliotheken der Dateimenge `libraries` zu kompilieren und in das Verzeichnis, das durch `dist.home.classes` festgelegt wurde, abzulegen. Das Zusammenstellen der Distribution beschränkt sich dann innerhalb des `dist`-Targets nur noch auf das Kopieren der entsprechenden Bibliotheken in den Distributionszweig. Ähnlich einfach ist die Entwicklung einer entsprechenden war-Datei. Dabei werden alle Dateien, die innerhalb des Distributionsverzeichnisses liegen über den `jar`-Task in einem Jar-Archiv zusammengefasst.

Im Anschluss daran verbleiben noch die beiden Targets `deploy` und `clean`, wobei der erste einen Kopierbefehl ausführt, um die war-Datei in den Applikationsserver zu kopieren, und die zweite das Distributionsverzeichnis über einen `delete`-Task löscht.

Wichtig an der Struktur der Build-Datei ist noch die Erkenntnis, dass verschiedene Targets voneinander abhängig sind. So hängt zum Beispiel `compile` von `prepare` ab und `dist` wiederum von `compile`. Wenn Targets voneinander abhängen, werden die Vorgelagerten vom Ant-Prozess automatisch angestoßen, so dass es möglich wird, das gesamte Projekt nur durch Aufruf eines Targets zu bauen und zu deployen.

Die komplette Build-Datei ist in Listing 3.3 dargestellt.

```
<?xml version='1.0' encoding='ISO-8859-1' standalone='no' ?>
<project name="Web Service Example" default="dist" basedir=".">
  <property name="app.name"    value="webserviceexample"/>
```

```
<property name="src"          value="code" />
<property name="dist"         value="dist"/>
 <property name="dist.home"    value="${dist}/${app.name}"/>
<property name="dist.home.webinf" value="${dist.home}/WEB-INF"/>
<property name="dist.home.classes"
               value="${dist.home.webinf}/classes"/>
<property name="dist.home.lib" value="${dist.home.webinf}/lib"/>
<property name="dist.war"     value="${app.name}.war"/>
<property environment="ENV"/>
<property name="deploy.home"
               value="${ENV.CATALINA_HOME}/webapps/"/>

<!-- libraries -->
<fileset id="libraries" defaultexcludes="yes" dir="lib">
  <include name="servlet.jar"/>
  <include name="soap.jar"/>
</fileset>

<!-- create directories -->
<target name="prepare">
   <mkdir dir="${dist}"/>
   <mkdir dir="${dist.home}"/>
   <mkdir dir="${dist.home.webinf}"/>
   <mkdir dir="${dist.home.classes}"/>
   <mkdir dir="${dist.home.lib}"/>
</target>

<!-- compile files -->
<target name="compile" depends="prepare">
   <javac srcdir="${src}" destdir="${dist.home.classes}" >
      <classpath>
         <fileset refid="libraries"/>
         <pathelement location="${src}" />
      </classpath>
   </javac>
</target>

<!-- build distribution -->
<target name="dist" description="create distribution dist subdirectory"
    depends="compile">

   <!-- copy libraries -->
   <copy todir="${dist.home.lib}">
      <fileset refid="libraries" />
   </copy>
</target>
```

```
  <!-- build .war file -->
  <target name="war" depends="dist" description="build war file.">
     <jar jarfile="${dist}/${dist.war}">
        <fileset dir="${dist.home}" />
     </jar>
  </target>

  <!-- deploy .war file to application server -->
  <target name="deploy" depends="war" description="distribute to $
      CATALINA_HOME/webapps">
    <copy file="${dist}/${dist.war}"
       tofile="${deploy.home}/${dist.war}">
    </copy>
  </target>

  <!-- delete intermediate files -->
  <target name="clean" description="delete intermediate files">
     <delete dir="${dist}"/>
  </target>
</project>
```

Listing 3.3: Build-Datei zum Kompilieren und Deployen des Projekts

Testen der Applikation

Nach dem Aufruf des Befehls `ant deploy` im Wurzelverzeichnis des Projektes erstellt Ant eine *.war*-Datei und kopiert diese in das `webapps`-Verzeichnis des Tomcat-Applikationsservers.

```
[manfred@lindan webservice]$ ant deploy
Buildfile: build.xml
prepare:
[mkdir] Created dir: dist
[mkdir] Created dir: dist/webserviceexample
[mkdir] Created dir: dist/webserviceexample/WEB-INF
[mkdir] Created dir: dist/webserviceexample/WEB-INF/classes
[mkdir] Created dir: dist/webserviceexample/WEB-INF/lib
compile:
[javac] Compiling 2 source files to
dist/webserviceexample/WEB-INF/classes
dist:
[copy] Copying 2 files to dist/webserviceexample/WEB-INF/lib
war:
[jar] Building jar: dist/webserviceexample.war
deploy:
```

```
[copy] Copying 1 file to
/opt/jakarta-tomcat-4.1.7/webapps
BUILD SUCCESSFUL
Total time: 14 seconds
```

Nachdem die .war-Datei ordnungsgemäß kopiert wurde, entdeckt Tomcat die neue Applikation und stellt über einen HTTP-Port-Connector auf Port 8080 den Dienst `webserviceexample/servlet/WebServiceServlet` zur Verfügung, der mit Hilfe eines HTTP-Clients angesprochen werden kann.

Einen HTTP-Client implementiert ebenfalls unser `WebServiceClient`, der sich gegen diesen Dienst verbindet und entsprechend bereitstehende Informationen abholen kann.

Der Aufruf des `WebServiceClients` funktioniert dabei wie folgt:

```
$ java WebServiceClient
Anfrage senden...
Warten auf Antwort...
```

Ist der Client erst einmal aufgerufen, stellt er zunächst eine Anfrage an den Server und wartet anschließend auf die Antwort. Wenn nun alle Komponenten richtig installiert sind, liefert Tomcat in dem Kommandozeileninterpreter in dem er gestartet wurde, die folgende Ausgabe:

```
Receive Request
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV=
  "http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi=
  "http://www.w3.org/1999/XMLSchema-instance"
 xmlns:xsd=
  "http://www.w3.org/1999/XMLSchema">
 <SOAP-ENV:Body>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Der Durchstich ist also geschafft! Wir verfügen nun über die technische Infrastruktur, SOAP-konforme Anfragen an einen HTTP-Server zu stellen und die entsprechende SOAP-Antwort entgegenzunehmen.

3.4 Web Service Description Language (WSDL)

Wir haben nun bereits die Möglichkeiten von XML zur Ablage und von SOAP zur Strukturierung von Daten sowie die technischen Grundlagen zur Übertragung von XML/SOAP über HTTP kennen gelernt. Für die standardisierte Versendung von SOAP in

Netzen fehlt uns bisher jedoch noch der »Klebstoff«, der das bloße Vorhandensein mit einer inhaltlichen Struktur verbindet. Selbst wenn ein Entwickler das Übertragungsprotokoll und die Art der Ablage der Daten der Applikation kennt, die er ansprechen möchte, fehlt ihm dennoch jedwede Spezifikation darüber, wie diese Daten strukturiert sind.

WSDL stellt hierbei eine weitere Basistechnologie für die Entwicklung von Web Services dar. Sie ist im Wesentlichen eine Spezifikation, die festlegt, wie Web Services über eine XML-Grammatik beschrieben werden können. Die Aufgabe bei der Entwicklung einer Web-Service-Beschreibung in WSDL besteht darin, eine Schnittstelle zwischen einem bereitgestellten Dienst und einem anfragenden Client zu definieren. Durch die Verwendung der WSDL-Beschreibung wird es möglich, Web Services automatisch zu propagieren und neue Funktionalität mit einem minimalen Anpassungsaufwand am Code der Server- und Client-Applikationen zu entwickeln.

WSDL ist das Ergebnis einer Zusammenarbeit von IBM und Microsoft. Beide Firmen hatten schon vor WSDL eine eigene Meta-Beschreibung für ihre noch experimentellen SOAP-Implementierungen, die natürlich nicht miteinander kompatibel waren. Diese beiden Meta-Beschreibungen, IBMs *Network Accessible Service Specification Language* (NASSL) und Microsofts *SOAP Contract Language* (SCL), formten dann die Basis für WSDL.

WSDL besitzt hierbei eine Vielzahl von Gemeinsamkeiten mit den *Interface Description Languages* (IDL) von CORBA (vgl. [CORBA IDL]) oder Microsoft (vgl. [Microsoft IDL]). Diese Beschreibungssprachen haben alle zum Ziel, Schnittstellen in Form von Methodensignaturen sowie zugehörige Datentypen zu spezifizieren. Die IDL ist allerdings eine eigene Sprache, die die Interfaces sehr ähnlich wie Java Interfaces beschreibt; WSDL hingegen ist — konsequentermaßen — in XML definiert, und damit nicht unbedingt so menschenlesbar wie IDL.

Gleichsam jedoch bietet WSDL ein weitaus größeres Maß an Erweiterbarkeit als dies bei den IDLs der Fall ist. So ist es z. B. möglich, Kommunikationsendpunkte für Nachrichten zu definieren, ohne das Format der Nachricht oder das Protokoll zu kennen, das für den Datenaustausch verwendet wird.

WSDL umfasst insgesamt vier Bereiche:

- Schnittstellendefinitionen , die alle öffentlich verfügbaren Funktionen beschreiben
- Beschreibung der Datentypen für alle Nachrichten, die ausgetauscht werden
- Informationen über Bindungen an ein Transportprotokoll und wie dieses benutzt wird
- Informationen für das Auffinden von benutzten Diensten

Ist einem Client die Web-Service-Beschreibung eines Dienstes bekannt, kann dieser einen Web Service lokalisieren und jede der öffentlich zugänglichen Funktionen aufrufen.

3.4.1 WSDL-Spezifikation

Die aktuelle WSDL-Spezifikation (siehe [W3C02b]) ist zurzeit lediglich ein Vorschlag an das W3-Consortium, dessen Standardisierung noch aussteht. An dieser Spezifikation haben unter anderem Ariba, IBM and Microsoft mitgearbeitet. In der Zusammenarbeit dieser unterschiedlichen Unternehmen liegt die Haupt-Chance, jedoch auch die Haupt-Gefahr für einen einheitlichen WSDL-Standard. Während sich einerseits die drei Firmen nach außen auf diesen Vorschlag geeinigt haben, heißt das nicht, dass dies auf der Umsetzungsebene auch der Fall ist. Es gibt zurzeit noch zu viel Interpretationsspielraum im Spezifikationsvorschlag. So kommt es vor, dass Microsofts .NET-WSDL nicht unbedingt immer einwandfrei mit der SUN-Implementierung zusammenarbeitet. Näheres hierzu findet sich in der praktischen Umsetzung der Kommunikation zwischen Java und .NET in Kapitel 5.

Wie bereits erwähnt, besteht eine WSDL-Definition aus verschiedenen Teilen, die innerhalb einer XML-Datei zusammengefasst werden. Jede dieser Teile wird über ein eigenes Tag eingeleitet. Wir werden im Folgenden zunächst die sieben wichtigsten Elemente betrachten.

Typdefinitionen

Die Typdefinition wird durch das `types`-Tag beschrieben und beschreibt alle Datentypen, die zwischen Client und Server ausgetauscht werden. WSDL verfügt nicht über ein eigenes plattform- und sprachunabhängiges Typ-System, so dass es möglich ist, hier beliebige Typ-Spezifikationen zu verwenden. Durch die enge Kopplung an SOAP hat sich jedoch die Verwendung von SOAP-Typdefinitionen als Standard herauskristallisiert. Für den Fall, dass die Spezifikation nur einfache SOAP-Datentypen, wie z. B. Integer oder Strings verwendet, die nicht zusammengesetzt werden, kann das `types`-Tag in der Spezifikation weggelassen werden.

Nachrichtendefinitionen

Definitionen von Nachrichten werden über das `message`-Tag festgelegt. Jede dieser `message`-Definitionen beschreibt dabei eine unidirektionale Nachricht entweder eine Anfrage (Client an Server) oder eine Antwort (Server an Client). Jede Nachrichtendefinition besteht aus dem Namen der Nachricht und optionalen `part`-Elementen, die Anfrageparameter bzw. Rückgabewerte festlegen.

Funktionsdefinitionen

Nachrichten lassen sich über Funktionsdefinitionen zusammenfassen. Hierdurch wird es möglich, bidirektionale Kommunikationswege zu beschreiben. Zu diesem Zweck wird das `portType`-Element benutzt.

Protokolldefinitionen

Die Protokolldefinition dient zur Festlegung des Übertragungsprotokolls. WSDL bietet hierzu bereits vordefinierte Spezifikationen für die Verwendung von SOAP. SOAP-spezifische Protokollinformationen werden deshalb ebenfalls hier definiert.

Lokalisierungsdefinitionen

Das `service`-Element beschreibt die Adresse für den Aufruf des Web Services. So unterschiedlich wie die unterstützten Protokolle sein können, sind hierbei auch die möglichen Lokalisierungsangaben. In der Regel werden hierbei jedoch HTTP-konforme URLs verwendet.

Dokumentation

Bei rein technischen Definitionen macht es Sinn, Erläuterungen und Beispiele mitzuliefern, um eine bessere Verständlichkeit zu erreichen. Zu diesem Zweck wird das `documentation`-Tag benutzt, das unterhalb von jedem anderen WSDL-Tag definiert werden kann.

Importdefinitionen

Das `import`-Element wird verwendet, um andere WSDL-Dokumente oder XML-Schema-Definitionen einzubinden. Somit wird es möglich, modulare WSDL-Dokumente zu erzeugen, die gegebenenfalls hierarchisch definiert sein können. Importe werden z. B. benutzt, wenn es bereits eine Datentypdefinition gibt, auf die innerhalb mehrerer anderer WSDL-Spezifikationen zugegriffen werden soll.

Abbildung 3.5 zeigt die schematische Struktur einer WSDL-Spezifikation.

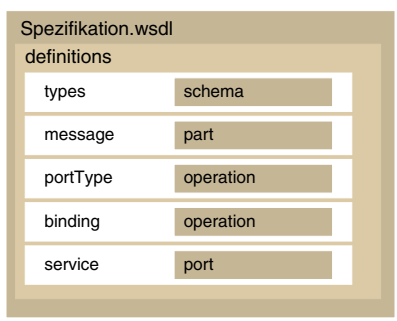

Abbildung 3.5: Struktur einer WSDL-Spezifikation

3.4.2 Datentypdefinition

Wenn SOAP-basierte Clients und Server effektiv miteinander kommunizieren sollen, müssen sie sich auf ein einheitliches Datentypsystem einigen. XML 1.0 liefert kein

Datentypsystem, während jede Programmiersprache in der Regel zumindest eine Basisfunktionalität zur Definition von Datentypen bereitstellt. Zudem sind viele Datentypen in Programmiersprachen plattformabhängig. Der Wertebereich eines Ganzzahlwertes wie z. B. `short` wird von einem C++-Compiler auf Windows nicht zwingend so definiert wie bei einem C-Compiler unter UNIX. Wenn nun unterschiedliche Programmiersprachen auf unterschiedlichen Architekturen über ein Protokoll, das bis auf die Datentypebene herabreicht, kommunizieren sollen, bedarf dies einer einheitlichen Datentypdefinition.

WSDL versucht nicht, Datentypen zu standardisieren, sondern ist auf maximale Flexibilität angelegt. Hierdurch wird es möglich, verschiedene Typdefinitionssysteme einzubinden. Das hierbei meist benutzte Format ist die Datentyp-Definition von XML Schema des W3C (vgl. [W3C01d]). Die Schema-Definition haben wir bereits in Abschnitt 2.3 kennen gelernt.

An dieser Stelle möchten wir die Verbindung zwischen der Datentypdefinition von XML Schema und WSDL schaffen und zeigen, wie man Datentypen in WSDL definiert.

Primitive Datentypen

Betrachten wir zunächst die Definition der primitiven Datentypen der XML Schema-Definition für Datentypen (vgl. [W3C01d]). Die Basis-Typliste umfasst Zeichenketten, Fließkommazahlen, Ganzzahlen, Zeit- und Datumsdefinitionen. Applikationen, die lediglich diese Typen benutzen, benötigen keine eigene Typdefinition. Hierdurch entfallen für die meisten WSDL-Anwendungen die Typdefinitionen komplett. Tabelle B.1 in Anhang B zeigt alle primitiven Datentypen von XML SOAP.

Felder und komplexe Datentypen

Neben der direkten Verwendung von primitiven Datentypen kann man zudem in XML Schema neue Datentypen definieren. Somit wird es möglich, Web Services mit komplexen Datentypen oder Feldern zu entwickeln. Die Mächtigkeit dieser Definitionen steht den Datentypkombinationen von objektorientierten Programmiersprachen wie Java oder C# in nichts nach.

3.5 Universal Description and Discovery Interface (UDDI)

UDDI steht für *Universal Description and Discovery Interface*. Die UDDI-Schnittstelle stellt eine standardisierte Methode für das Publizieren und das Auffinden von Web Services zur Verfügung. Die Schnittstelle wurde durch eine Initiative aus der Industrie mit dem Ziel der Schaffung eines offenen und plattformunabhängigen Frameworks für das Beschreiben, Registrieren und Auffinden von Web-basierten Diensten und Produkten entwickelt. Der Schwerpunkt liegt hierbei im Bereich des Suchens und Auffindens von Diensten.

Das UDDI-Projekt wird von der UDDI-Community betrieben (siehe [UDDI-WebSite]), die zwei Bereiche betreut. Die *UDDI Working Group* beschäftigt sich als geschlossene Kommune hauptsächlich mit der Entwicklung von Spezifikationen, während die *UDDI Advisory Group* als öffentliche Gruppe Anforderungen sammelt und die Spezifikationen der Working Group validiert.

UDDI löst das Problem des Auffindens von Web Services. Dies ist ein extrem wichtiger Aspekt bei der Einführung einer neuen Technologie. Wir wissen beispielsweise, dass zu den meisten denkbaren Themen Informationen im Internet zur Verfügung stehen. Diese Informationen lassen sich über Hyperlinks komfortabel systemübergreifend miteinander verbinden. Jedoch haben die Erfinder des Wold Wide Web nicht daran gedacht, einen effizienten Mechanismus zum Auffinden von Informationen bereitzustellen. Als Folge benutzen wir heute Suchmaschinen, die verschiedenste Strategien anwenden, um Inhalte zu indizieren und somit durchaus unterschiedliche Ergebnisse liefern, mit denen wir in aller Regel nicht zufrieden sind.

Noch komplexer verhält es sich im Kontext von Diensten im Netz, wenn sich diese nicht nur mehr auf einfache Business-to-Business-Prozesse beschränken. Selbst wenn es möglich ist, Dienste im Netz aufzufinden, stellt sich immer noch die Frage, ob es sich dabei wirklich um den für die aktuelle Anwendung passenden handelt.

Für diese Probleme bietet UDDI die Lösung. Die UDDI-Registry stellt ein dezentrales System zur Registrierung und Verbreitung von Dienstbeschreibungen zur Verfügung. Sie besteht aus mehreren UDDI-Knotenpunkten, die sich untereinander abgleichen. Wird nun beispielsweise ein neuer Service an einem dieser Knoten registriert, wird er wenige Minuten später in allen anderen Knoten verfügbar sein.

Die Funktionsweise von UDDI ist dabei auf den ersten Blick recht einfach. Im Kontext einer Web-Service-Architektur bildet UDDI die Schnittstelle für das Auffinden und Registrieren von Diensten zwischen Service-Anbietern (Providern) und Nutzern (Clients). Dies geschieht unter Verwendung einer klassischen Service-Broker-Architektur, wie Abbildung 3.6 veranschaulicht.

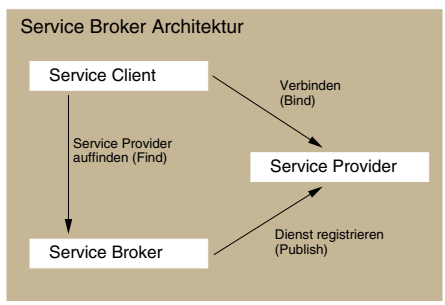

Abbildung 3.6: UDDI im Kontext einer Web Service-Architektur

Die UDDI-Spezifikation 1.0 wurde von Microsoft, IBM und Ariba im September 2000 freigegeben. Inzwischen wird die UDDI-Initiative von zirka 300 Unternehmen unterstützt. Im Mai 2001 starteten Microsoft und IBM die ersten UDDI-Server. Im Juni 2001

wurde eine Version 2.0 von UDDI angekündigt, die folgende zusätzlichen Features enthalten soll:

- Verbesserung der Internationalisierung, um die Beschreibbarkeit von Diensten in verschiedenen Sprachen zu ermöglichen
- Verbesserungen im Bereich der Beschreibung komplexer Unternehmensstrukturen, so dass es möglich wird, Business-Units, Abteilungen und Unterabteilungen direkt über eine Referenz zugreifbar zu machen
- Erweiterung der Suchoptionen

UDDI 2.0 wird zurzeit testweise bei SAP, Hewlett-Packard, Microsoft und IBM eingesetzt, so dass davon auszugehen ist, dass eine endgültige Freigabe unmittelbar bevorsteht.

Dieses Kapitel bietet einen Überblick über die UDDI-Funktionen und wie sie verwendet werden. Es beschreibt, welche Informationen innerhalb der UDDI-Registry verwaltet werden und welche technischen Möglichkeiten die UDDI-Architektur bietet. Des Weiteren liefern wir einen Überblick über die Programmierschnittstelle und zeigen praktische Anwendungsbeispiele für den Zugriff auf UDDI von Java aus.

3.5.1 Verwendung von UDDI

UDDI lässt sich in verschiedenen Kontexten nutzen. Aus einer reinen Anwendersicht betrachtet stellt UDDI eine Suchmaschine dar, in der unterschiedliche kommerzielle Dienste gefunden werden können. Die Verwendung von UDDI als Suchmechanismus bietet jedoch eine Reihe von Vorteilen im Vergleich zu z. B. Internet-Suchmaschinen. Während die Informationen im Internet unstrukturiert und abstrakt abgelegt sind, da es kein verbindliches, einheitliches Format gibt, ist innerhalb einer UDDI-Registry das Format für jeden Zweck entsprechend vorgegeben. Andererseits jedoch sind die Informationen nicht unbedingt benutzungsfreundlich abgelegt, so dass eine weiterführende Aufbereitung mit dem Ziel einer besseren Lesbarkeit für den Endanwender vonnöten ist, wenn diese direkt in einer UDDI-Registry browsen wollen. Zu diesem Zweck bieten einige Unternehmen bereits entsprechende Portale an.

Neben der reinen Informationssuche für Anwender benutzen Entwickler die UDDI-Registry zum Publizieren und Auffinden von IT-Diensten. Der Vorteil von UDDI liegt dabei in der Eingrenzbarkeit der Suchparameter und dem hohem Automationsgrad des Dienstes, so dass sich das Publizieren und Auffinden ohne manuelle Eingriffe realisieren lässt. So ist es möglich, dass verschiedene Dienste automatisch durch Software erkannt und angesprochen werden können. Dies macht Updates von Diensten sehr einfach, da keinerlei Anpassungsaufwand auf Seiten der Applikation erforderlich ist.

Somit können sowohl Endanwender als auch Softwareentwickler Dienste in verschiedenen oder in der gleichen Registry ablegen und auffinden. Des Weiteren können auf diese Dienste über Endanwender-freundliche, Web-basierte Portale oder über Programmierschnittstellen zugegriffen werden. Abbildung 3.7 veranschaulicht diesen Zusammenhang.

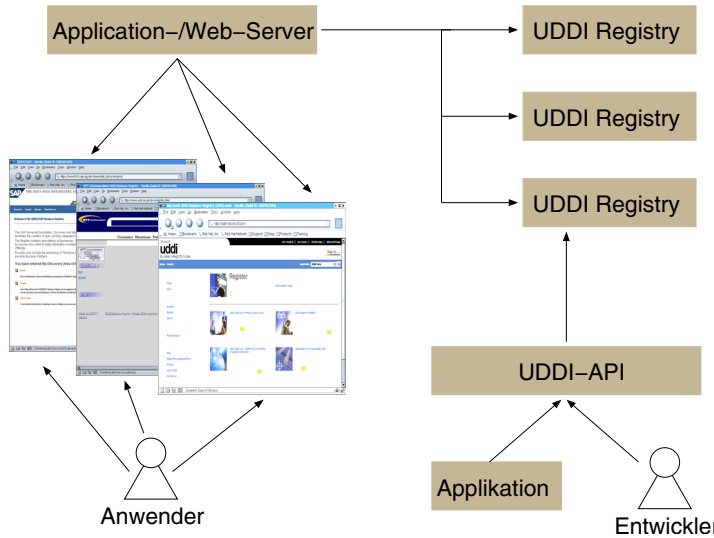

Abbildung 3.7: Benutzergruppen im UDDI-Kontext

Informationen in UDDI-Registries

UDDI wurde für die Ablage von technischen Endpunkten in Geschäftsprozessen entwickelt. Aus diesem Grund lassen sich die gespeicherten Informationen in drei wesentliche Bereiche abgrenzen:

- **Kontaktinformationen**
  Kontaktinformationen dienen zur Identifizierung eines Geschäftspartners. Sie enthalten Namen, Adressen, Steuernummern, usw. Diese Informationen ermöglichen es, Web Services aufgrund der Person bzw. des Unternehmens aufzufinden, die den Dienst bereitstellt. In der angelsächsischen Literatur wird dieser Teil als *White Pages* bezeichnet.

- **Service-Informationen**
  Service-Informationen ordnen Web Services in Kategorien ein. Somit wird es möglich, unabhängig vom Service-Anbieter verschiedenste Services aufgrund einer Kategorie zu ermitteln. Dieser Informationsteil wird *Yellow Pages* genannt.

- **Technische Informationen**
  Die technischen Informationen beschreiben das Verhalten und die Support-Funktionen eines Web Service. Hierzu gehören ebenfalls Gruppierungsinformationen und Angaben über den physischen Standort der Services, die als *Green Pages* bezeichnet werden.

3.5.2 Technische Struktur

Kern der technischen Struktur von UDDI ist die *UDDI Business Registry (UBR)*, welche konzeptionell einem dezentralen System entspricht, dessen Elemente über einen Replikationsmechanismus abgeglichen werden. Dieses Vorgehen ist äquivalent zum sehr bekannten *Domain Name Service (DNS)*, der zur Propagierung von Rechnern im Internet benutzt wird, in dem virtuelle Namen (z. B. *www.addison-wesley.de*) auf physische IP-Adressen (z. B. `62.245.190.22`) übersetzt werden.

Die einzelnen Elemente bzw. Knoten werden als *Operator Nodes* bezeichnet. Mehrere Operator Nodes können zu einem Gesamtsystem zusammengeschlossen werden und replizieren die Daten untereinander. Die Gesamtheit dieser Knoten bildet dann die Business Registry (UBR). Somit bleibt sichergestellt, dass die Anfrage an einem spezifischen Knoten im System dieselben Antworten bringt, wie an jedem anderen Knoten im System.

Wenn eine Information an einem Knoten eingefügt wird, wird dieser der logische Eigentümer (*Master Owner*) dieser Information. Alle Löschungen und Änderungen dieser Information müssen dann ebenfalls an diesem Knoten erfolgen.

Durch die dezentrale Struktur des UBR ist es möglich, sowohl private als auch öffentliche UDDI-Server zu betreiben. Somit sind innerhalb von Unternehmen z. B. Mischformen aus privaten und öffentlichen Ansätzen möglich, wenn das Unternehmen beispielsweise eine Reihe von firmeneigenen Diensten besitzt, die nicht nach außen propagiert werden sollen, jedoch ebenfalls auf externe Dienste zugegriffen werden soll. Dies ist durch eine einfache Klassifizierung in zu replizierende und nicht zu replizierende Einträge erreichbar.

Die UBR besitzt eine Reihe von Möglichkeiten zum Auffinden von gespeicherten Diensten, die von den jeweiligen Besitzern unter verschiedenen Autorsierungsmodellen eingestellt werden können. Jedes Unternehmen kann somit ganze UBR-Knoten in das Gesamtsystem über das Internet integrieren sowie Teile davon schützen.

3.5.3 Kommerzielle Nutzung von UDDI

UDDI war ursprünglich zum Publizieren von Diensten in einem globalen Netzwerk gedacht. Zurzeit liegt der Schwerpunkt in den Unternehmen jedoch ganz klar auf der Verteilung von Diensten in eigenen privaten UBR-Knoten, um firmeninterne Dienste zu propagieren, um die internen Abläufe und technische Prozesse zu optimieren.

Übergreifende oder gar globale Dienste werden zurzeit noch recht wenig propagiert. Hier kristallisiert sich jedoch heraus, dass verschiedene Industrie- und Dienstleistungsverbände sowie Softwareunternehmen versuchen, einzelne Registries für spezifische Interessengruppen zu etablieren. Somit ist zukünftig zu erwarten, dass es eine Reihe verschiedener UDDI-Registries von unterschiedlichen Industrie- und Dienstleistungsgruppen für die jeweiligen Anforderungen ihrer Mitglieder geben wird.

In den letzten zwei Jahren wurden eine Reihe von Produkten entwickelt, die es erlauben, eigene öffentliche und private UDDI-Registries aufzusetzen. So wird beispielsweise der BEA Weblogic Applikationsserver ab Version 7 (vgl. *www.bea.com*) mit einer eigenen UDDI-Registry ausgeliefert. Ebenso ist für den IBM Applikationsserver WebSphere (*www.ibm.com/websphere*) ein entsprechendes UDDI-Server Plugin im Internet verfügbar (*http://www.alphaworks.ibm.com/tech/uddiregextensions*). Des Weiteren existieren eine Reihe von Implementierungen namhafter Firmen für Java, die wir in Tabelle 3.4 aufgeführt haben.

Name	URL	Bemerkung
Systinet	*www.systinet.com*	WASP UDDI Registry, kommerziell
Hewlett-Packard	*www.hpmiddleware.com*	HP UDDI Business Registry, kommerziell
Oracle	*otn.oracle.com*	OTN UDDI Registry, kommerziell
SAP	*uddi.sap.com*	SAP UDDI Business Registry, kommerziell
The Mind Electric	*www.themindelectric.com/glue*	GLUE UDDI Registry, kommerziell
Silverstream	*www.silverstream.com*	Novell JEDDI Server, UDDI 1.0 kompatibel, kommerziell, aber Download zur Evaluierung
jUDDI	*juddi.sourceforge.net* oder *www.juddi.org*	UDDI 2.0 kompatibel, Open Source, noch nicht released
pUDDIng	(zurzeit keine gültige Website)	Open Source UDDI Registry
SOAPUDDI	*soapuddi.sourceforge.net*	Open Source UDDI Registry, UDDI 2.0 kompatibel

Tabelle 3.4: Implementierungen von UDDI-Registries

Eine Reihe von Firmen bietet öffentliche UDDI-Registries an, die zum einem tatsächlich dem primären Zweck des Informationsaustauschs für Dienste dienen, zum anderen jedoch auch zum Testen von entsprechenden UDDI-Schnittstellen benutzt werden können. Tabelle 3.5 stellt eine Liste dieser frei zugänglichen Registries dar. Diese stehen sowohl in einer Test- als auch in einer Produktivversion zur Verfügung.

Name	URL
IBM	*http://www-3.ibm.com/services/uddi/*
Microsoft	*http://uddi.microsoft.com*
SAP AG	*http://uddi.sap.com*

Tabelle 3.5: Öffentlich zugängliche UDDI-Registries

3.5.4 UDDI-Datenmodell

In diesem Abschnitt betrachten wir die internen Strukturen einer UDDI-Registry und die Nachrichten, die eine solche Registry zur Kommunikation nach außen verwendet.

Datenstrukturen

Um die Struktur von UDDI-Nachrichten zu verstehen, benötigt man einen grundsätzlichen Überblick über die internen Datenstrukturen und XML-Formate, in die diese Strukturen übersetzt werden. Dieser Abschnitt behandelt die Strukturen, die in Abbildung 3.8 dargestellt sind.

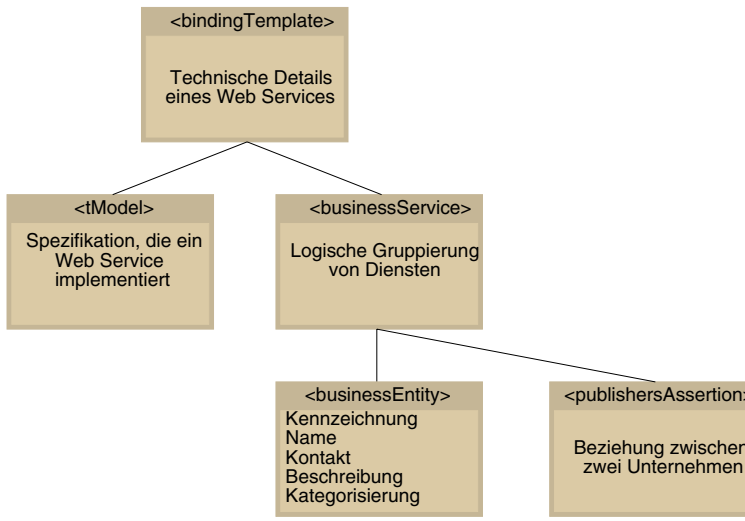

Abbildung 3.8: Elementare UDDI-Datenstrukturen

businessEntity-Struktur

Eine `businessEntity`-Struktur repräsentiert eine grundlegende Geschäftsinformation (*business information*). Diese Information beinhaltet Kontaktinformationen, die Kategorisierung, eine Beschreibung, die Beziehung zu anderen Objekten und einen eindeutigen Schlüssel (*identifier*). UDDI ist darauf angelegt, dass unterschiedliche Unternehmen Kontakte untereinander knüpfen, die innerhalb der Registry abgelegt werden können. Hierbei sind eine ganze Reihe unterschiedlicher Beziehungen definiert. So ist es beispielsweise möglich, dass ein Konzern auf seine Konzerntöchter und diese wiederum auf ihre Niederlassungen referenzieren. Ebenfalls ist es möglich, dass zwei Firmen untereinander eine Relation, wie beispielsweise eine »Partnerschaft« oder ein Lieferanten/Abnehmer-Verhältnis definieren. In beiden Fällen müssen alle Unternehmen zunächst als `businessEntity` definiert werden, bevor Verknüpfungen in irgendeiner Art und Weise erstellt werden dürfen. Ein Beispiel für ein solches Objekt liefert Listing 3.4, in der eine Firma namens »My little test Web Service Inc.« beschrieben ist, die genau eine Kontaktperson samt Anschrift und eMail-Adresse besitzt.

```
<businessEntity businessKey="">
 <name xml:lang="en">My little test Web Service Inc.</name>
 <description xml:lang="en">Just a dummy company for testing
 </description>
 <contacts>
  <contact>
   <description/>
   <personName>Manfred Hein</personName>
   <phone useType="">+49/40/12345678</phone>
   <email>manfred.hein@addison-wesley.de</email>
   <address sortCode=""
    tModelKey="uuid:6EAF4B50-4196-11D6-9E2B-000629DC0A2B"
    useType="">
    <addressLine keyName="StreetAddressNumber"
     keyValue="StreetAddressNumber">123</addressLine>
    <addressLine keyName="StreetAddress"
     keyValue="StreetAddress">Teststrasse</addressLine>
    <addressLine keyName="City"
     keyValue="City">Hamburg</addressLine>
    <addressLine keyName="ZipCode"
     keyValue="ZipCode">20251</addressLine>
    <addressLine keyName="Country"
     keyValue="Country">Germany</addressLine>
   </address>
  </contact>
 </contacts>
 <businessServices>
  ..
 </businessServices>
</businessEntity>
```

Listing 3.4: Eine UDDI businessEntity-Struktur

publisherAssertion-Struktur

Die `publisherAssertion`-Struktur wird benutzt, um eine Beziehung zwischen zwei `businessEntity`-Strukturen zu schaffen. Eine Beziehung zweier Unternehmen ist nur dann nach außen hin sichtbar, wenn beide Partner jeweils eine äquivalente Beziehung (über die `publisherAssertion`-Struktur) geschaffen haben, so dass es, wie im richtigen Leben, nicht reicht, wenn einer der beiden Partner meint, er habe eine Beziehung. Wenn wir also zwei Firmen A und B annehmen und Firma A eine Beziehung x zu Firma B aufgebaut hat, so wird diese nur dann sichtbar, wenn auch Firma B eine Beziehung x zu Firma A aufbaut. Dieses Vorgehen wird verwendet, damit keine versehentlichen oder nicht erwünschten Unternehmensstrukturen publiziert werden.

businessService-Struktur

Wie in Abbildung 3.8 dargestellt, besteht eine Beziehung zwischen der `businessEntity`- und der `businessService`-Struktur. Hierbei kann eine `businessEntity` einen oder mehrere `businessService`-Strukturen referenzieren (beinhalten). Ein `businessService` ist hierbei eine einzelne logische Klassifikation von Diensten, die dazu dient, einzelne Dienstleistungen eines Unternehmens zu beschreiben. Diese Dienste können dabei sowohl »echte« (Dienst)-Leistungen aus wirtschaftlichen Bereichen, wie auch Web-Service-Dienste, sein. `businessService`-Strukturen sind wiederverwendbar, was konkret bedeutet, dass sie nur einmal definiert werden müssen und eine beliebige Anzahl von `businessEntity`-Strukturen darauf verweisen können.

Eine beispielhafte `businessService`-Struktur ist in Listing 3.5 dargestellt.

```
<businessService serviceKey="">
  <name xml:lang="en">Web Service Development</name>
  <description xml:lang="en">Source for advanced Web Service technology.</
      description>
  <bindingTemplates>
    ...
  </bindingTemplates>
</businessService>
```

Listing 3.5: Eine UDDI businessService-Struktur

bindingTemplate-Struktur

Ein `businessService` kann einen oder mehrere `bindingTemplate`-Strukturen enthalten. Diese enthält ihrerseits eine (optionale) technische Beschreibung und eine Zugriffs-URL auf den entsprechenden Service. Sie dient also dazu, den Dienst tatsächlich zugreifbar zu machen. Eine `bindingTemplate`-Struktur besitzt zudem eine oder mehrere Referenzen auf eine `tModel`-Struktur und vor allem ein `accessPoint`-Element, das die Zugriffsadresse des Dienstes bereitstellt. Listing 3.6 zeigt diesen Zusammenhang, wobei hier auf eine einfache URL im Internet verwiesen wird.

```
<bindingTemplate bindingKey="">
  <description>Web Service Examples</description>
  <accessPoint URLType="http">http://jwsbuch.com/</accessPoint>
  <tModelInstanceDetails/>
</bindingTemplate>
```

Listing 3.6: Eine UDDI businessService-Struktur

tModel-Struktur

Die `tModel`-Struktur dient zur eindeutigen Identifikation eines Dienstes und legt gleichzeitig fest, wie mit dem entsprechenden Web Service kommuniziert werden kann. Allerdings enthält diese Struktur keinerlei direkte Spezifikation des Dienstes, sondern Zeiger

auf Ressourcen im Web oder in der realen Welt, die diese Spezifikationen beinhalten. Die Informationen, die hierunter abgelegt sind, werden verwendet, um zu prüfen, ob ein Dienst tatsächlich auf die gewünschten Erfordernisse passt.

Die Datenstrukturen, die hinter UDDI liegen, sind zugegebenermaßen recht abstrakt und generisch. Dies hat damit zu tun, dass UDDI für die Publizierung von Diensten aller Art ausgelegt ist. Dies können sowohl Web-Service-Dienste sein, wobei eine Strukturierung sicherlich recht einfach wäre, als auch reale Dienste, bei denen eine eingehende Strukturierung sicherlich deutlich komplexer ist und ein großes Maß an Generizität benötigt.

UDDI-Nachrichten

Es existieren eine Reihe von Nachrichtentypen, die es erlauben, eine UDDI-Registry anzusprechen, um beispielsweise ein Unternehmen, einen Dienst oder Metadaten einer Spezifikation abzufragen. Alle hierbei ausgetauschten Nachrichten sind SOAP-Nachrichten (siehe Abschnitt 3.2).

Diese Nachrichtentypen lassen sich grob in 5 Kategorien einteilen:

- Authentifikation
- Suchen
- Detailinformationen abfragen
- Hinzufügen/Ändern
- Löschen

Im Folgenden möchten wir jede dieser einzelnen Kategorien vorstellen und die tatsächlich implementierten Nachrichtentypen beschreiben.

Authentifikation

Die Authentifikationsnachrichten dienen dazu, sich bei der UDDI-Registry an- bzw. abzumelden. Generell wird für alle schreibenden Zugriffe auf die Registry eine vorangegangene Anmeldung benötigt, die eine Authentifizierungsmarke zurückliefert. Diese

Anfrage	Antwort	Beschreibung
`get_authToken`	`authToken`	Auf Basis eines gültigen Benutzernamens und Passwortes für die UDDI-Registry liefert diese eine entsprechende Authentifikationsmarke zurück, mit der sich schreibend auf die Registry zugreifen lässt.
`discard_authToken`	`disposition Report`	Wird dieser Anfrage eine gültige Authentifikationsmarke übergeben, wird diese von der Registry für ungültig erklärt. Dies kommt einem Logout gleich.

Tabelle 3.6: UDDI-Nachrichtentypen zur Authorisierung

Marke ist vergleichbar mit einer Session-ID, die der Client bei jeder Anfrage an den Server reicht, um sicherzustellen, dass er die angeforderten Operationen ausführen darf. Tabelle 3.6 beschreibt die beiden Nachrichten zum An- und Abmelden.

Listing 3.7 zeigt eine Authorisierungsanfrage des Benutzers `soapuddi` mit dem Passwort `soapuddi` an einen UDDI-Server.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
  <soap-env:Body>
    <get_authToken xmlns="urn:uddi-org:api_v2" cred="soapuddi"
      generic="2.0" userID="soapuddi"/>
  </soap-env:Body>
</soap-env:Envelope>
```

Listing 3.7: Anfrage einer UDDI-Authentifizierung

Die darauf folgende Antwort ist in Listing 3.8 dargestellt. Sie enthält die Authentifikationsmarke als ID im Element `authInfo`.

```
<?xml version='1.0' encoding='UTF-8'?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <authToken generic="2.0">
      <authInfo>8f5f2076-f300-0000-0080-e4c98e9846ec</authInfo>
    </authToken>
  </Body>
</Envelope>
```

Listing 3.8: Antwort einer UDDI-Authentifizierung

Suchen

Alle Nachrichten, die zum Auffinden von Diensten dienen, beginnen im SOAP-Body mit einen Element, das mit `find` anfängt. Es lassen sich Informationen über Firmen, Sparten und Dienste abfragen. Für diese Nachrichtentypen ist keine vorangehende Authentifikation notwendig. Tabelle 3.7 zeigt die Nachrichtentypen dieser Klasse mit Anfrage- und Antwort-Elementen sowie einer kurzen Beschreibung. Beispiele für diese Art von Nachrichten werden in Abschnitt 3.5.6 behandelt.

Detailinformationen abfragen

Zu bestehenden Einträgen lassen sich Detailinformationen beschaffen. Alle Nachrichtentypen, die sich mit diese Thematik befassen, beginnen mit `get_` (siehe Tabelle 3.8). Für jede der Strukturtypen `bindingTemplate`, `businessEntity`, `businessService` und `tModel` ist ein entsprechender Anfragemechanismus definiert. Diese nehmen in der Regel

Anfrage	Antwort	Beschreibung
`find_binding`	`bindingDetail`	Eine Anfrage dieses Typs sucht zu einer gegebenen `businessService`-Struktur nach auf die Anfrage passenden `bindingTemplate`-Strukturen, die innerhalb eines `bindingDetail`-Elements zurückgegeben werden.
`find_business`	`businessList`	Auf Basis eines eingegebenen regulären Ausdrucks, einer Kategorie, eines Business-Identifiers oder einer Struktur des Typs `tModel` gibt eine Anfrage dieses Typs entsprechend gefundene `businessInfo`-Strukturen innerhalb des `businessList`-Elements zurück.
`find_relatedBusinesses`	`relatedBusinessesList`	Eine Anfrage dieses Typs liefert eine Liste von IDs für `businessEntity`-Strukturen zurück, die zu einer eingegebenen ID für eine `businessEntity`-Struktur passen.
`find_service`	`serviceList`	Auf Basis einer gegebenen ID einer `businessEntity`-Struktur zusammen mit dem Namen eines Dienstes, der tModel-Struktur einer implementierten Spezifikation oder einer Dienstkategorie liefert diese Anfrage eine Liste aller passenden `businessService`-Strukturen zurück.
`find_tmodel`	`tModelList`	Auf einen gegebenen `tModel`-Namen, einer Kategorie oder einer ID liefert diese Anfrage alle passenden `tModel`-Strukturen zurück.

Tabelle 3.7: UDDI-Nachrichtentypen zum Auffinden von Diensten

eine oder mehrere Primärschlüssel (IDs) von entsprechenden Datenstrukturen entgegen und suchen innerhalb der Registry nach diesen IDs. IDs werden von der Registry beim Einfügen einer Struktur angelegt und bezeichnen jede Struktur eindeutig.

Für die Nachrichtentypen zum Finden von Detailinformationen ist keine vorangehende Authentifikation notwendig.

Hinzufügen/Ändern/Löschen

Die Hauptunterschiede zwischen dem Auffinden und Auslesen von Information aus einer UDDI-Registry und dem Schreiben und Löschen sind in der Sensibilität der Daten und die technische Infrastruktur von UDDI zu suchen. Die folgenden Punkte stellen diese Unterschiede dar:

Anfrage	Antwort	Beschreibung
get_binding Detail	binding Detail	Auf Basis einer oder mehrerer IDs von bindingTemplate-Strukturen liefert diese Anfrage die bindingTemplate-Dokumente für alle passenden IDs zurück.
get_business Detail	business Detail	Auf Basis einer oder mehrerer IDs von businessEntity-Strukturen liefert diese Anfrage die businessEntity-Dokumente für alle passenden IDs zurück.
get_service Detail	service Detail	Auf Basis einer oder mehrerer IDs von businessService-Strukturen liefert diese Anfrage die businessService-Dokumente für alle passenden IDs zurück.
get_tmodel Detail	tModel Detail	Auf Basis einer oder mehrerer IDs von tModel-Strukturen liefert diese Anfrage die tModel-Dokumente für alle passenden IDs zurück.

Tabelle 3.8: UDDI-Nachrichtentypen zum Auslesen von Detailinformationen

- **Benutzerauthentifikation**
  Alle publizierenden Nachrichten benötigen einer vorhergehende Authentifikation. Dieser Mechanismus ist in der UDDI-Spezifikation festgehalten und verbindlich für alle Operator Nodes. Nachdem eine Authentifikation erfolgreich war, kann die zurückgelieferte Authentifikationsmarke so lange benutzt werden, bis ein entsprechender Abmeldevorgang initiiert wurde.

- **Unterschiedliche Zugriffspunkte**
  Die Zugriffs-URLs zum Auslesen und Publizieren von Informationen sind unterschiedlich. Der Grund dafür liegt in der unterschiedlichen Sicherheitseinstufung. Zum einen ist ein schreibender Benutzer am System registriert, so dass der Austausch seiner Authentifikationsdaten über eine verschlüsselte Verbindung erfolgen sollte. Zum anderen kann es sein, dass seine Informationen nicht unbedingt sofort der Öffentlichkeit zur Verfügung stehen sollten, also eine gewisse Vertaulichkeitskomponente beinhalten und nicht ungeschützt über das Netz transportiert werden sollten. Beim Auslesen von Informationen spielt dies kaum eine Rolle, da alle nach außen gesendeten Informationen sowieso öffentlich sind. Dies führt in der Regel dazu, dass die Zugriffs-URL für das Auslesen als normaler Dienst im Netz öffentlich zugänglich ist, während schreibende Komponenten über eine spezielle, meist SSL-verschlüsselte Schnittstelle verfügbar sind.

- **Größenlimitierung**
  Der Betreiber eines UDDI-Knotens kann Regelungen für Größenbeschränkungen für die Einträge treffen. So ist es beispielsweise denkbar, dass gewisse Firmen in der Anzahl ihrer businessEntity-Strukturen beschränkt sind, um zu vermeiden, dass

zusätzliche oder unnütze Informationen willentlich oder versehentlich in die Registry geschrieben werden.

- **Gebundenheit an den Operator Node**
  Wie wir bereits in der Einleitung dargestellt haben, können UDDI-Registries verteilt werden. UDDI selbst besitzt jedoch keine Funktionalitäten zum Auflösen von redundanten Informationen oder Konflikten innerhalb von Daten. Aus diesem Grund wird der Operator Node, der die Daten das erste Mal eingefügt bekommen hat, Besitzer der so genannten Master-Copy, also dem Dokument, das verbindlich die richtige Information enthält. Alle nachfolgenden Änderungen müssen direkt auf der Master-Copy vollzogen werden, so dass der Client seine Änderungen immer auf demselben Operator Node durchführen muss.

Hinzufügen/Ändern

Die Nachrichtentypen für Hinzufügen und Ändern von Strukturen sind so aufgebaut, dass in der Regel das Hinzufügen und Ändern mit einem Nachrichtentyp durchgeführt werden kann. Ist eine Struktur bereits vorhanden, wird sie geändert, ist sie der Registry neu, wird sie angelegt. Die Nachrichten für diese Operationen beginnen namentlich mit `save_`. Die einzige Ausnahme ist die Nachricht `add_publisherAssertions`, die dazu dient, Beziehungen zwischen Unternehmen zuzufügen. Dieser Nachrichtentyp wurde ganz bewusst zugefügt, da es durchaus sein kann, dass ein Unternehmen sehr viele Beziehungen hat und es deshalb unpraktikabel wäre, zunächst die gesamte alte Beziehungsmenge auszulesen, eine neue Beziehung hinzuzufügen und wieder zurückzuschreiben. Ein einfacher Zufügemechanismus ist hier deutlich schneller und einfacher zu handhaben. Eine Liste dieser Operationen ist in Tabelle 3.9 zu finden.

Für alle zufügenden und ändernden Nachrichtentypen ist eine vorangehende Authentifikation notwendig, wobei die zurückgegebene Authentifikationsmarke bei jeder Operation mitgegeben werden muss.

Beispiele für diese Art von Nachrichten werden in Abschnitt 3.5.7 behandelt.

Löschen

Tabelle 3.10 zeigt eine Liste der in UDDI möglichen Löschnachrichten. Es ist hierbei möglich, eine oder mehrere IDs von `bindingTemplate`, `businessEntity`, `publisherAssertion`, `businessService und tModel` übergeben, die innerhalb der Registry gelöscht werden sollen. Nachrichtentypen für das Löschen fangen immer mit `delete_` an und benötigen, wie die Operationen zum Hinzufügen, eine vorhergehende Authentifikation.

Anfrage	Antwort	Beschreibung
add_publisherAssertions	dispositionReport	Auf Basis einer gültigen Authentifikationsmarke und einer publisherAssertion-Struktur trägt diese Anfrage eine neue Beziehung zwischen zwei Unternehmen in die Registry ein.
save_binding	bindingDetail	Diese Anfrage nimmt eine gültige Authentifikationsmarke sowie eine bindingTemplate-Struktur entgegen. Ist die Struktur innerhalb der Registry neu, wird sie zugefügt. Andernfalls werden bereits bekannte Strukturen verändert.
save_business	businessDetail	Diese Anfrage nimmt eine gültige Authentifikationsmarke sowie eine businessEntity-Struktur entgegen. Ist die Struktur innerhalb der Registry neu, wird sie zugefügt. Andernfalls werden bereits bekannte Strukturen verändert.
save_service	serviceDetail	Diese Anfrage nimmt eine gültige Authentifikationsmarke sowie eine businessService-Struktur entgegen. Ist die Struktur innerhalb der Registry neu, wird sie zugefügt. Andernfalls werden bereits bekannte Strukturen verändert.
save_tModel	tModelDetail	Auf Basis einer gültigen Authentifikationsmarke und einer oder mehrerer tModel-Strukturen fügt die Registry diese zu bzw. ändert bereits bestehende in der Eingabe referenzierte tModels. Vormals mit dem hidden-Flag versehene referenzierte tModels werden wieder auf »sichtbar« gesetzt.
set_publisherAssertion	publisherAssertions	Auf Basis einer gültigen Authentifikationsmarke und einer oder mehrerer publisherAssertion-Strukturen trägt diese Anfrage alle Beziehungen zwischen zwei Unternehmen in die Registry ein. Alle Beziehungen zu referenzierten Unternehmen, die nicht in der übergebenen Liste enthalten sind, werden dabei gelöscht.

Tabelle 3.9: UDDI-Nachrichtentypen zum Hinzufügen und Ändern von Informationen

3.5.5 UDDI Java APIs

Die UDDI-Spezifikation (vgl. [UDDI-WebSite]) umfasst keine direkte Definition für eine Programmierschnittstelle. Stattdessen wird ein genereller Vorschlag für Remote Procedure Calls via SOAP (siehe Abschnitt 3.1 und Abschnitt 3.2) gemacht. Eben aus diesem Grund gibt es verschiedene Ansätze zum Zugriff auf UDDI. Als Java-Entwickler können wir insgesamt aus drei dieser Ansätze wählen:

Anfrage	Antwort	Beschreibung
`delete_binding`	`dispositionReport`	Diese Anfrage nimmt eine gültige Authentifikationsmarke sowie die ID einer oder mehrerer `bindingTemplate`-Strukturen entgegen. Daraufhin löscht sie das durch die ID bezeichnete `bindingTemplate` aus der Registry.
`delete_business`	`dispositionReport`	Auf Basis einer gültigen Authentifikationsmarke und der ID einer `businessEntity`-Struktur löscht diese Anfrage sowohl die entsprechende `businessEntity` als auch alle davon referenzierten Unterstrukturen.
`delete_publisherAssertions`	`dispositionReport`	Diese Anfrage nimmt eine gültige Authentifikationsmarke sowie eine oder mehrere `publisherAssertion`-Strukturen entgegen. Daraufhin löscht sie die entsprechenden Strukturen in der Registry.
`delete_service`	`dispositionReport`	Auf Basis einer gültigen Authentifikationsmarke und der ID einer `businessService`-Struktur löscht diese Anfrage die entsprechende `businessEntity`-Struktur.
`delete_tModel`	`dispositionReport`	Diese Anfrage nimmt eine gültige Authentifikationsmarke sowie eine oder mehrere `tModel`-Strukturen entgegen. Daraufhin dereferenziert die Registry die entsprechenden Strukturen, indem sie ein spezielles hidden-Flag setzt (sie also unsichtbar macht). Der Grund für dieses Vorgehen, anstelle des Anwendens eines Löschmechanismus, liegt darin, dass einzelne Unternehmensdaten nicht komplett gelöscht werden können. So sind die entsprechenden `tModel`s immer noch durch die Anfragen `get_tModelDetail` und `get_registeredInfo` auffindbar.

Tabelle 3.10: UDDI-Nachrichtentypen zum Löschen von Informationen

- SOAP
  Bei der Verwendung von SOAP kann der Entwickler UDDI-Dokumente via SOAP versenden und empfangen. Der Nachteil an diesem Vorgehen ist die Tatsache, dass diese Dokumente selbst erzeugt werden müssen, was natürlich sehr zeitaufwändig und fehleranfällig ist.

- Proprietäre APIs
  Nahezu alle Anbieter von UDDI-Registries (siehe Tabelle 3.4) bieten eigene (nichtstandardisierte) Programmierschnittstellen zum Zugriff an, wobei diese als schneller oder robuster gepriesen werden, als z. B. die Standard-API JAXR. Der Vorteil solcher APIs ist in der Regel ihre Einfachheit, da sie direkt auf spezielle Bedürfnisse zugeschnitten werden können. Der große Nachteil liegt jedoch wiederum darin, dass

eine Anwendung, die eine solche Programmierschnittstelle benutzt, auf Gedeih und Verderb auf das entsprechende Registry-Produkt angewiesen ist.

- JAXR
  Mittels JAXR ist es möglich, über spezielle API-Element direkt auf eine UDDI-Registry zuzugreifen. Der Vorteil an dieser Programmierschnittstelle ist deren Standardisiertheit. Somit ist es möglich, jede UDDI-kompatible Registry anzusprechen, anstatt wie bei den proprietären APIs nur jeweils eine spezielle. Dies hat den Vorteil, dass die tatsächliche Registry-Implementierung unabhängig von darüber liegenden System ausgetauscht werden können. Nachteilig hierbei ist die Tatsache, dass JAXR ein extrem breites Feld an Funktionalität bieten muss, damit die meisten Registry-Features abgedeckt werden können. Dies führt dazu, dass die JAXR-Funktionen sehr generisch und dadurch komplex sind.

Wir werden bei den in diesem Kapitel benutzten Beispielen hauptsächlich auf JAXR als Programmierschnittstelle zurückgreifen, da diese die am weitesten verbreitete, komfortabelste und standardisierteste API der Probanden darstellt. Einen Gesamtüberblick über die Models, die JAXR für UDDI-Registries verwendet, bietet Abbildung 3.9. Wir werden diese in den folgenden Abschnitten gut gebrauchen können, wenn es darum geht, JAXR zu benutzen, um Funktionen auf einer UDDI-Registry durchzuführen.

3.5.6 Suchen über UDDI

Wie in Abschnitt 3.5.1 dargestellt, existieren zwei grundsätzliche Möglichkeiten, um auf eine UDDI-Registry zuzugreifen. Die eine ist für den reinen Anwender gedacht, der sich lediglich über unterschiedliche Unternehmen und Dienste informieren möchte, während die zweite eine technische Schnittstelle für Systeme darstellt. In den folgenden beiden Abschnitten werden wir diese beiden Ansätze im Kontext der Suche von Diensten detaillieren.

Web-basierte Suche

Eine Web-basierte Suche nach Einträgen einer UDDI-Registry stellt technisch nichts anderes dar, als ein System, das über eine Schnittstelle mit einem UDDI-Server kommuniziert und die entsprechenden Informationen als Webseite zurückgibt. Abbildung 3.10 zeigt die Startseite der SAP-UDDI-Registry, die über *http://uddi.sap.com* zu erreichen ist. Die Seite besteht aus einem öffentlichen Testbereich, auf der man nach Anmeldung nach Belieben ausprobieren kann.

Die Suchfunktionen sind äquivalent zur Schnittstelle von UDDI implementiert. Abbildung 3.11 zeigt beispielsweise die Suchfunktionalität für Dienste, Unternehmen und tModels über eine Freitextsuche. An der Oberfläche wird bereits sichtbar, dass das System einen UDDI-Server benutzt, da beispielsweise sprachabhängige Funktionalitäten angeboten werden.

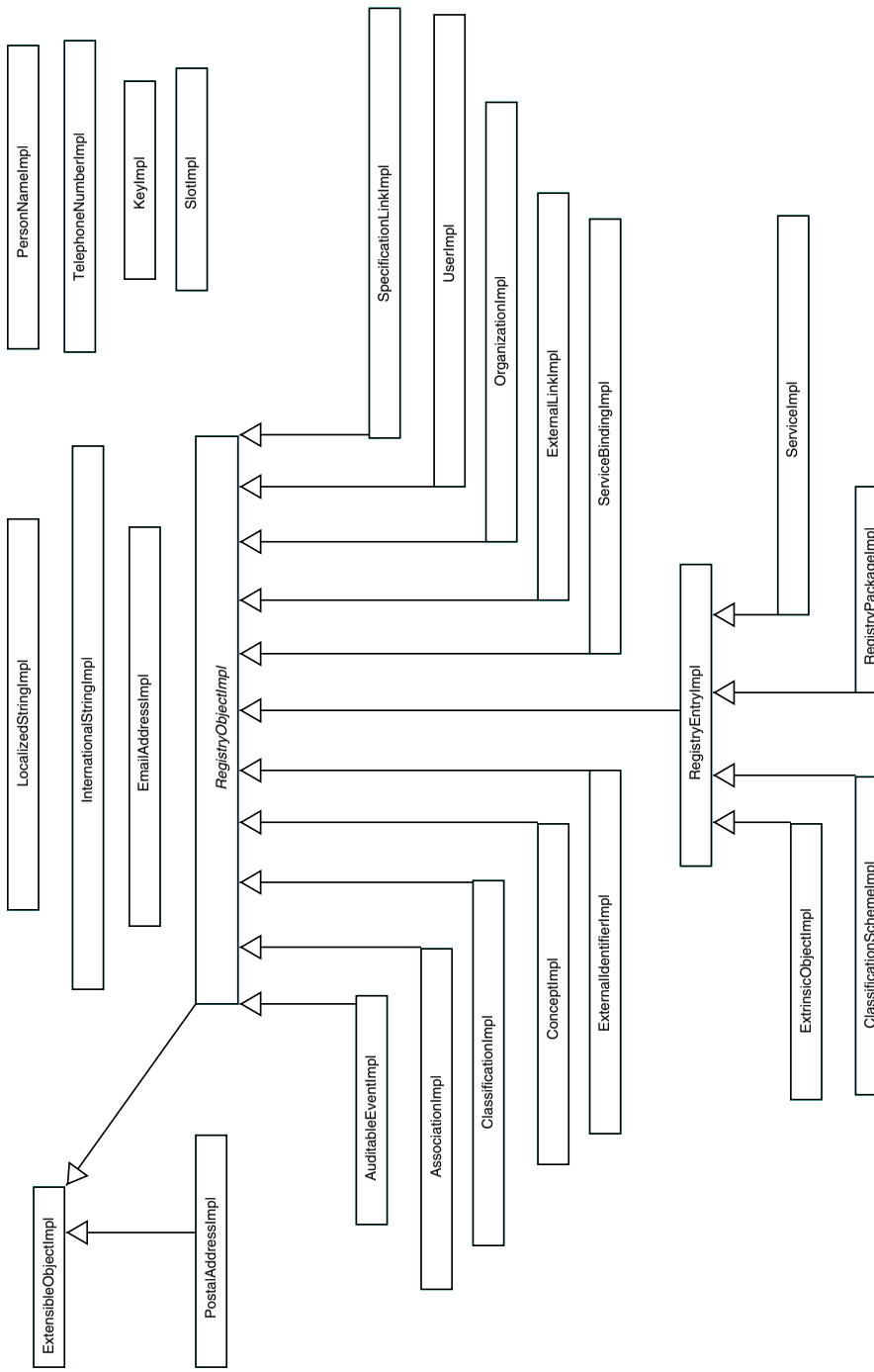

Abbildung 3.9: Klassendiagramm der JAXR-API

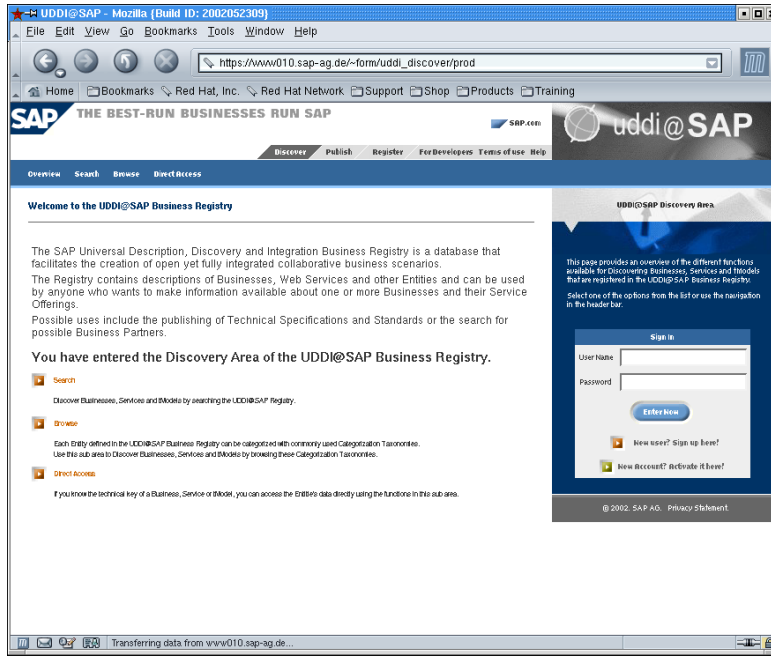

Abbildung 3.10: UDDI-Suche bei UDDI@SAP

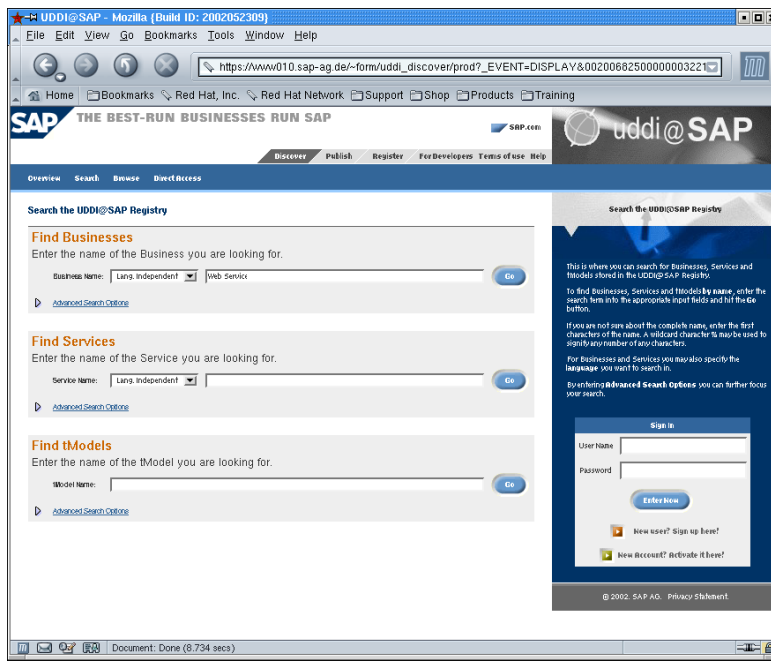

Abbildung 3.11: UDDI-Freitextsuche bei UDDI@SAP

Des Weiteren ist es möglich, über Kategorien zu suchen, wie dies in Abbildung 3.12 dargestellt ist. Somit wird es möglich, über eine baumartige Struktur die gesuchten Dienste zu finden.

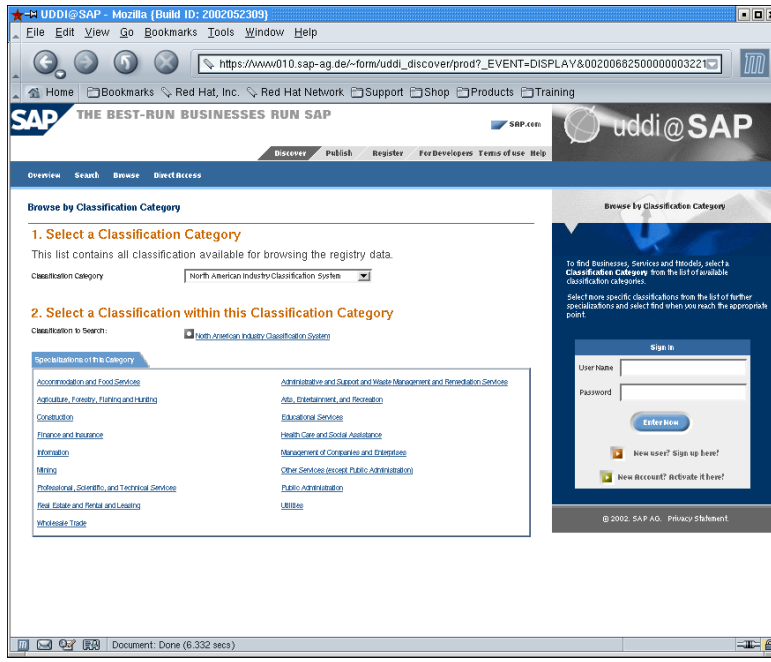

Abbildung 3.12: UDDI-Klassifikationssuche bei UDDI@SAP

API-gestützte Suche

In diesem Abschnitt betrachten wir den Zugriff auf eine UDDI-Registry aus einem Programm heraus. Das Ziel hierbei ist das Auffinden eines Dienstes und die Ausgabe der relevanten Informationen. Ein Programm, das dieses leistet, ist in Listing 3.9 dargestellt.

```
import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;
import java.net.*;
import java.util.*;

public class JAXRfind {
  public JAXRfind() {
  }

  public void query(String queryString) {
    Connection con=null;
    Properties props=new Properties();
    props.setProperty("javax.xml.registry.queryManagerURL",
```

```
                "http://uddi.sap.com/UDDI/api/inquiry/");
    props.setProperty("javax.xml.registry.factoryClass",
                "com.sun.xml.registry.uddi.ConnectionFactoryImpl");

  try {
    ConnectionFactory factory=ConnectionFactory.newInstance();
    factory.setProperties(props);
    con=factory.createConnection();

    RegistryService rs=con.getRegistryService();
    BusinessQueryManager bqm=rs.getBusinessQueryManager();

    Collection qualifiers=new ArrayList();
    qualifiers.add(FindQualifier.SORT_BY_NAME_DESC);
    Collection namePatterns=new ArrayList();
    namePatterns.add(queryString);

    BulkResponse response=bqm.findOrganizations(qualifiers,
                  namePatterns, null, null, null, null);
    Collection orgs=response.getCollection();
    Iterator orgIter=orgs.iterator();
    while(orgIter.hasNext()) {
      Organization org=(Organization)orgIter.next();
      System.out.println("Organization:"+org.getName().getValue());
      printServices(org);
      printPrimaryContact(org);
      System.out.println("\n");
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}

private void printServices(Organization org) throws JAXRException {
  Collection services=org.getServices();
  Iterator sIter=services.iterator();
  while(sIter.hasNext()) {
    Service service=(Service)sIter.next();
    System.out.println(" Service name:"+service.getName().getValue());
    System.out.println(" Service description:"+
                  service.getDescription().getValue());

    Collection bindings=service.getServiceBindings();
    Iterator bIter=bindings.iterator();
    while(bIter.hasNext()) {
      ServiceBinding binding=(ServiceBinding)bIter.next();
```

```
            System.out.println(" Binding description:"+
                               binding.getDescription().getValue());
            System.out.println(" Access URI:"+binding.getAccessURI());
          }
        }
      }


      private void printPrimaryContact(Organization org) throws JAXRException {
        User primaryContact=org.getPrimaryContact();
        if(primaryContact!=null) {
          PersonName name=primaryContact.getPersonName();
          System.out.println(" Contact: "+name.getFullName());
          Collection emails=primaryContact.getEmailAddresses();
          Iterator iter=emails.iterator();
          while(iter.hasNext()) {
            EmailAddress email=(EmailAddress)iter.next();
            System.out.println(" EMail: "+email.getAddress());
          }
        }
      }

      public static void main(String[] args) {
        JAXRfind finder=new JAXRfind();
        if(args.length!=1) {
          System.out.println("Usage: java JAXRfind <query>");
          return;
        } else {
          finder.query(args[0]);
        }
      }
    }
```

Listing 3.9: Programm zum Auffinden von Diensten über UDDI

Das Programm besteht im Wesentlichen aus zwei Methoden, die die Ergebnisse einer Anfrage ausgeben, und aus der Methode `query`, die diese Anfrage an eine UDDI-Registry ausführt. `query` generiert sich zunächst ein Verbindungsobjekt, das die relevanten Verbindungsinformationen zur UDDI-Registry enthält. Ausgehend von dieser Verbindung kann auf ein Objekt der Klasse `RegistryService` zurückgegriffen werden, das von JAXR bereitgestellt wird. Dieser Service besitzt seinerseits wieder den Zugriff auf lesende und schreibende Komponenten der Verbindung. Da wir in diesem Fall Abfragen machen wollen, beschaffen wir uns den `BusinessQueryManager`, der die Methode `findOrganization` bereitstellt, mit der sich die tatsächliche Abfrage durchführen lässt. Als Antwortobjekt kommt eine Liste von Organisationselementen zurück, die sich mit Hilfe der entsprechenden API ausgeben lässt.

Ein entsprechender Testlauf des Programms mit dem Suchwort »Carbon« sieht folgendermaßen aus:

```
java JAXRfind Carbon
Organization:Carbon Composites
Contact: Jerome Eberhardt
EMail: jaker@carb.com
```

Interessant hierbei ist nun, welche Anfragen tatsächlich an den UDDI-Server gestellt wurden. Durch Zwischenschalten eines einfachen Analysewerkzeuges zur Überwachung des HTTP-Datenverkehrs (siehe Anhang A) ist es relativ einfach möglich, die entsprechenden Nachrichten »mitzuschneiden«.

Zunächst einmal stellt JAXR, wie in Listing 3.10 dargestellt, eine `find_business`-Anfrage, um die Menge aller Unternehmen zurückgeliefert zu bekommen, die »Carbon« in ihrem Namen oder ihrer Beschreibung tragen.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope
    /">
  <soap-env:Body>
    <find_business xmlns="urn:uddi-org:api_v2" generic="2.0">
      <findQualifiers>
        <findQualifier>sortByNameDesc</findQualifier>
      </findQualifiers>
      <name xml:lang="en">Carbon</name>
    </find_business>
  </soap-env:Body>
</soap-env:Envelope>
```

Listing 3.10: find_business-Anfrage an eine UDDI-Registry

Als Antwort wird (siehe Listing 3.11) genau eine `businessInfo`-Struktur zurückgeliefert, die die entsprechenden Basisinformationen über die Firma »Carbon Composites« enthält.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <businessList generic="2.0" operator="SAP AG" xmlns="urn:uddi-org:api_v2"
        >
      <businessInfos>
        <businessInfo
          businessKey="91b67755-14e9-4a0b-8507-fc8c26acc652">
          <name xml:lang="en">Carbon Composites</name>
          <description xml:lang="en">This is a UDDI Business Registry entry
              for "Carbon Composites".</description>
          <serviceInfos></serviceInfos>
        </businessInfo>
```

```
    </businessInfos>
   </businessList>
  </Body>
</Envelope>
```

Listing 3.11: businessInfo-Antwort einer UDDI-Registry

An dieser Stelle ist der JAXR-API noch nichts über eventuelle Kontaktpersonen oder Dienstbeschreibungen bekannt. Sobald ein Zugriff auf ein detaillierendes Element erfolgt, wie z. B. der Telefonnummer oder eMail-Adresse, stellt die API fest, dass sie diese Informationen noch nicht hat, und führt eine entsprechende `get_businessDetail`-Anfrage aus (siehe Listing 3.12).

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope
  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/">
  <soap-env:Body>
    <get_businessDetail xmlns="urn:uddi-org:api_v2" generic="2.0">
      <businessKey>91b67755-14e9-4a0b-8507-fc8c26acc652</businessKey>
    </get_businessDetail>
  </soap-env:Body>
</soap-env:Envelope>
```

Listing 3.12: get_businessDetail-Anfrage an eine UDDI-Registry

Das darauf folgende Antwortdokument enthält dann eine entsprechende `business-Entity`-Struktur, die alle Detailinformationen zu diesem Unternehmen enthält (Listing 3.13).

```
<?xml version="1.0" encoding="UTF-8" ?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
  <Body>
    <businessDetail generic="2.0" operator="SAP AG" xmlns="urn:uddi-
        org:api_v2">
      <businessEntity businessKey="91b67755-14e9-4a0b-8507-fc8c26acc652"
          operator="Microsoft Corporation" authorizedName="RealNames UDDI
          Publisher">
        <discoveryURLs>
          <discoveryURL useType="businessEntity">http://uddi.microsoft.
              com/discovery?businesskey=91b67755-14e9-4a0b-8507-
              fc8c26acc652</discoveryURL>
        </discoveryURLs>
        <name xml:lang="en">Carbon Composites</name>
        <description xml:lang="en">This is a UDDI Business Registry entry
            for "Carbon Composites".</description>
        <contacts>
          <contact useType="Main">
```

```
        <description xml:lang="en">Main contact information</
            description>
        <personName>Jerome Eberhardt</personName>
         <phone useType="Main">808-579-8000</phone>
        <email useType="Main">jaker@carb.com</email>
        <address useType="Main" sortCode="">
          <addressLine>PO Box 791777</addressLine>
          <addressLine>Paia, HI 96779</addressLine>
          <addressLine>HI</addressLine>
        </address>
      </contact>
      <contact useType="Technical">
        <description xml:lang="en">Technical contact information</
            description>
        <personName>Jerome Eberhardt</personName>
        <phone useType="Technical">808-579-8000</phone>
        <email useType="Technical">jaker@carb.com</email>
        <address useType="Technical" sortCode="">
          <addressLine>PO Box 791777</addressLine>
          <addressLine>Paia, HI 96779</addressLine>
          <addressLine>HI</addressLine>
        </address>
      </contact>
      <contact useType="Billing">
        <description xml:lang="en">Billing contact information</
            description>
        <personName>Jerome Eberhardt</personName>
        <phone useType="Billing">808-579-8000</phone>
        <email useType="Billing">jaker@carb.com</email>
        <address useType="Billing" sortCode="">
          <addressLine>PO Box 791777</addressLine>
          <addressLine>Paia, HI 96779</addressLine>
          <addressLine>HI</addressLine>
        </address>
      </contact>
    </contacts>
    <identifierBag>
      <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
          b012d0e0" keyName="basic keyword" keyValue="carbon
          composites from Carbon Composites US"></keyedReference>
      <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
          b012d0e0" keyName="basic keyword" keyValue="carbon fabric
          from Carbon Composites BR"></keyedReference>
      <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
          b012d0e0" keyName="basic keyword" keyValue="carbon fabric
          from Carbon Composites US"></keyedReference>
```

```
            <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
                b012d0e0" keyName="basic keyword" keyValue="carbon fabric
                from Carbon Composites CA"></keyedReference>
             <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
                b012d0e0" keyName="basic keyword" keyValue="carbon fabric
                from Carbon Composites FR"></keyedReference>
            <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
                b012d0e0" keyName="basic keyword" keyValue="carbon fabric
                from Carbon Composites DE"></keyedReference>
            <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
                b012d0e0" keyName="basic keyword" keyValue="carbon fabric
                from Carbon Composites GB"></keyedReference>
            <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
                b012d0e0" keyName="basic keyword" keyValue="carbon fabric
                from Carbon Composites IT"></keyedReference>
            <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
                b012d0e0" keyName="basic keyword" keyValue="carbon fabric
                from Carbon Composites ES"></keyedReference>
            <keyedReference tModelKey="uuid:3bb93de3-cf9a-4f4d-b553-6537
                b012d0e0" keyName="basic keyword" keyValue="carbon fiber
                from Carbon Composites US"></keyedReference>
          </identifierBag>
        </businessEntity>
      </businessDetail>
    </Body>
</Envelope>
```

Listing 3.13: businessDetail-Antwort einer UDDI-Registry

3.5.7 Publizieren über UDDI

In unserem vorigen Beispiel haben wir Daten über JAXR aus der UDDI-Registry ausgelesen. Dieser Abschnitt beschäftigt sich mit dem Hinzufügen und Verändern von Informationen in einer UDDI-Registry. Wir werden hierbei die Publikation über Web-Interfaces nicht näher beleuchten, da sie ähnlich trivial ist, wie das Auffinden von Diensten.

Das Beispielprogramm, das in Listing 3.14 dargestellt ist, fügt einen vollständigen Datensatz mit Firmeninformationen, Ansprechpartnern und Dienstleistungen in eine UDDI-Registry ein. Es besteht aus vier Methoden. Für den Verbindungsaufbau ist die Methode `connect` zuständig. Sie beschafft sich zunächst eine Verbindung zum UDDI-Server. Hierzu werden eine Reihe von Verbindungsparametern gefolgt von einer Reihe von Parametern für eine eventuelle Adressablage in der Registry an die entsprechende Fabrikmethode übergeben. Im Anschluss daran folgt eine Authentifizierungssequenz über ein Objekt der Klasse `PasswordAuthentication`, worauf die zurückgelieferte Authentifikationsmarke an die bestehende Verbindung gehängt wird. Dieses Vorgehen stellt sicher, dass alle Operationen auf dieser Verbindung unter der aktuellen Authentifizierung laufen.

```
import java.util.*;
import java.net.*;
import javax.xml.registry.*;
import javax.xml.registry.infomodel.*;


public class JAXRpublish {

  public JAXRpublish() {
  }

  private Connection connect(String username,String password) throws
      JAXRException {
    ConnectionFactory factory = ConnectionFactory.newInstance();
    Properties props = new Properties();
    props.setProperty("javax.xml.registry.queryManagerURL",
                  "https://udditest.sap.com/UDDI/api/publish");
    props.setProperty("javax.xml.registry.lifeCycleManagerURL",
                  "http://udditest.sap.com/UDDI/api/publish");
    props.setProperty("javax.xml.registry.factoryClass",
                  "com.sun.xml.registry.uddi.ConnectionFactoryImpl");
    props.setProperty("javax.xml.registry.postalAddressScheme",
                  "uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b");
    props.setProperty("javax.xml.registry.semanticEquivalences",
                  "urn:uuid:PostalAddressAttributes/StreetNumber," +
                  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/
                      StreetAddressNumber|" +
                  "urn:uuid:PostalAddressAttributes/Street," +
                  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/
                      StreetAddress|" +
                  "urn:uuid:PostalAddressAttributes/City," +
                  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/City|" +
                  "urn:uuid:PostalAddressAttributes/State," +
                  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/State|" +
                  "urn:uuid:PostalAddressAttributes/PostalCode," +
                  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/ZipCode|"
                       +
                  "urn:uuid:PostalAddressAttributes/Country," +
                  "urn:uuid:6eaf4b50-4196-11d6-9e2b-000629dc0a2b/Country")
                      ;


    factory.setProperties(props);
    Connection connection = factory.createConnection();
    System.out.println("Created connection to registry");
```

```
    PasswordAuthentication passwdAuth = new PasswordAuthentication(
        username, password.toCharArray());
     Set creds = new HashSet();
    creds.add(passwdAuth);
    connection.setCredentials(creds);
    System.out.println("Established security credentials");

    return connection;
  }

  private Organization publishOrganization(Connection connection,
      BusinessLifeCycleManager blcm) throws JAXRException {
    Organization org = blcm.createOrganization("My little test Web Service
         Inc.");
    InternationalString s = blcm.createInternationalString("Just a dummy
        company for testing");
    org.setDescription(s);
    return org;
  }

  private void publishPrimaryContact(Organization org,
      BusinessLifeCycleManager blcm) throws JAXRException {
    User primaryContact = blcm.createUser();
    PersonName pName = blcm.createPersonName("Manfred Hein");
    primaryContact.setPersonName(pName);

    TelephoneNumber tNum = blcm.createTelephoneNumber();
    tNum.setNumber("+49/40/12345678");
    Collection phoneNums = new ArrayList();
    phoneNums.add(tNum);
    primaryContact.setTelephoneNumbers(phoneNums);

    PostalAddress postAddr = blcm.createPostalAddress("123", "Teststrasse"
        , "Hamburg", "", "Germany", "20251", "");
    Collection postalAddresses = new ArrayList();
    postalAddresses.add(postAddr);
    primaryContact.setPostalAddresses(postalAddresses);

    EmailAddress emailAddress = blcm.createEmailAddress("manfred.
        hein@addison-wesley.de");
    Collection emailAddresses = new ArrayList();
    emailAddresses.add(emailAddress);
    primaryContact.setEmailAddresses(emailAddresses);

    org.setPrimaryContact(primaryContact);
  }
```

```
 private void publishServices(Organization org,BusinessLifeCycleManager
    blcm) throws JAXRException {
  Collection services = new ArrayList();
  Service service = blcm.createService("Web Service Development");
  InternationalString is = blcm.createInternationalString("Source for
      advanced Web Service technology.");
  service.setDescription(is);

  Collection serviceBindings = new ArrayList();
  ServiceBinding binding = blcm.createServiceBinding();
  is = blcm.createInternationalString("Web Service Examples");
  binding.setDescription(is);
  binding.setAccessURI("http://localhost:8080");
  serviceBindings.add(binding);

  service.addServiceBindings(serviceBindings);
  services.add(service);
  org.addServices(services);
}

public void publish(String username,String password) throws JAXRException
    {
  Connection connection=connect(username,password);
  RegistryService rs        = connection.getRegistryService();
  BusinessLifeCycleManager blcm = rs.getBusinessLifeCycleManager();

  Organization org=publishOrganization(connection,blcm);
  publishPrimaryContact(org,blcm);
  publishServices(org,blcm);

  Collection orgs = new ArrayList();
  orgs.add(org);
  BulkResponse response = blcm.saveOrganizations(orgs);

  // check for exceptions within the response
  Collection exceptions = response.getExceptions();
  if(exceptions == null ) {
    System.out.println("Organization saved");
    Iterator keyIter = response.getCollection().iterator();
    if(keyIter.hasNext()) {
      javax.xml.registry.infomodel.Key orgKey = (javax.xml.registry.
          infomodel.Key) keyIter.next();
      System.out.println("Organization key is " + orgKey.getId() );
      org.setKey(orgKey);
    }
```

```
    } else {
        for(Iterator excIter = exceptions.iterator(); excIter.hasNext();) {
            Exception exception = (Exception) excIter.next();
            System.out.println(exception.toString());
        }
    }
    connection.close();
  }

  public static void main(String[] args) throws Exception
  {
    String username = args[0];
    String password = args[1];

    JAXRpublish publisher= new JAXRpublish();
    publisher.publish(username,password);
  }
}
```

Listing 3.14: Programm zum Publizieren von Diensten über UDDI

Nach Ablauf der Authentifizierung werden die Unternehmensdaten über die Methode `publishOrganization()` in entsprechende JAXR-Strukturen eingetragen. Dasselbe passiert anschließend mit den Daten für die Kontaktperson (`publishPrimaryContact()`) und den vom Unternehmen angebotenen Diensten (`publishServices()`). Nachdem diese Strukturen nun feststehen, werden die Gesamtinformationen über die Methode `publishOrganization()` eines `BusinessLifeCycleManager`-Objekts in die UDDI-Registry geschrieben. Im Anschluss daran lesen wir noch die Rückgabewerte aus, um den Primärschlüssel des angelegten Unternehmens zurückzuerhalten. Die entsprechende Anfrage, die JAXR absendet, ist in Listing 3.15 zu sehen.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope
    /">
  <soap-env:Body>
    <save_business xmlns="urn:uddi-org:api_v2" generic="2.0">
      <authInfo>f17e6676-f300-0000-0080-e4c98e9846ec</authInfo>
        <businessEntity businessKey="">
          <name xml:lang="en">My little test Web Service Inc.</name>
          <description xml:lang="en">Just a dummy company for testing</
              description>
          <contacts>
            <contact>
            <description/>
            <personName>Manfred Hein</personName>
            <phone useType="">+49/40/12345678</phone>
            <email>manfred.hein@addison-wesley.de</email>
```

```
          <address sortCode="" tModelKey="uuid:6EAF4B50-4196-11D6-9E2B
              -000629DC0A2B" useType="">
            <addressLine keyName="StreetAddressNumber" keyValue="
                StreetAddressNumber">123</addressLine>
            <addressLine keyName="StreetAddress" keyValue="StreetAddress">
                Teststrasse</addressLine>
            <addressLine keyName="City" keyValue="City">Hamburg</addressLine>
            <addressLine keyName="ZipCode" keyValue="ZipCode">20251</
                addressLine>
            <addressLine keyName="Country" keyValue="Country">Germany</
                addressLine>
          </address>
        </contact>
      </contacts>
      <businessServices>
        <businessService serviceKey="">
          <name xml:lang="en">Web Service Development</name>
          <description xml:lang="en">Source for advanced Web Service
              technology.</description>
          <bindingTemplates>
            <bindingTemplate bindingKey="">
              <description>Web Service Examples</description>
              <accessPoint URLType="http">http://jwsbuch.com/</accessPoint>
              <tModelInstanceDetails/>
            </bindingTemplate>
          </bindingTemplates>
        </businessService>
      </businessServices>
    </businessEntity>
  </save_business>
 </soap-env:Body>
</soap-env:Envelope>
```

Listing 3.15: UDDI-Anfrage zum Speichern von Firmeninformationen

Die Rückgabe des Servers besteht dann aus einem Echo der eingegebenen Daten, die zusätzlich um die IDs angereichert sind, mit denen sich die Datensätze in der Registry referenzieren lassen. Aus diesem Grund ist sie in Listing 3.16 auch nur verkürzt dargestellt.

```
<?xml version='1.0' encoding='UTF-8'?>
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">
 <Body>
  <businessDetail generic="2.0">
   <businessEntity businessKey="cf806676-f300-0000-0080-e4c98e9846ec">
   <name xml:lang="en" xmlns:xml="http://www.w3.org/XML/1998/namespace">My
       little test Web Service Inc.</name>
```

```
   ...
       <businessServices>
        <businessService serviceKey="">
          <name xml:lang="en" xmlns:xml="http://www.w3.org/XML/1998/
              namespace">Web Service Development</name>
   ...
        </businessService>
       </businessServices>
     </businessEntity>
   </businessDetail>
  </Body>
</Envelope>
```

Listing 3.16: UDDI-Antwort bei Speicherung von Firmeninformationen

3.5.8 Aufsetzen einer eigenen UDDI-Registry

In Abschnitt 3.5.3 sind einige öffentliche UDDI-Registries beschrieben, die zur produktiven Nutzung oder zum Testbetrieb eingerichtet worden sind. Wenn Sie die UDDI-Schnittstellen und Toolkits auf UDDI ausprobieren möchten, stellen diese Test-Registries eine gute Möglichkeit dar. Der Nachteil hierbei ist allerdings, dass man sich eine gemeinsame Umgebung mit vielen anderen Nutzern teilt und somit die publizierten Informationen auch eingesehen werden können. In diesem Abschnitt möchten wir uns mit dem Aufsetzen einer einfachen UDDI-Registry für lokale Netze oder für den Testgebrauch beschäftigen. Wir haben uns hierbei für den Open Source UDDI-Server SOAP-UDDI entschieden (siehe Tabelle 3.4), der komplett in Java geschrieben ist. Der Grund dafür liegt in der einfachen Installation und der Plattformunabhängigkeit des Systems. Einige der Beispiele in diesem Kapitel wurden direkt mit diesem System erstellt.

Das Ziel des Projektes SOAPUDDI ist die Entwicklung einer UDDI-Registry, die dem UDDI-Standard 2.0 entspricht, für die primäre Verwendung in Intranet-Systemen. Die Entwicklung ist noch nicht komplett abgeschlossen, so dass die zur zeit aktuellste Version 0.3.1 alpha ist und noch nicht alle Features komplett implementiert sind. Das System ist als Quellcode oder Binärdistribution unter *http://sourceforge.net/projects/soapuddi/* aus dem Netz herunterladbar. Bestandteil der Distribution sind Datenbankskripte für Microsoft SQL Server, PostgreSQL, Oracle, Sybase und Access, die die Datenbankstruktur für die interne Abbildung der Registry darstellen. Zudem existiert eine .war-Datei, die ein einfaches Deployment auf Jakarta Tomcat oder einen anderen Servlet-Container zulässt (siehe Abschnitt 3.3).

Nachdem man eine entsprechende Datenbank aufgesetzt hat, muss in der Datei `$CATALINE_HOME/WEB-INF/classes/conf/soapuddi.config` noch die entsprechende JDBC-Datenquelle eingetragen werden. Listing 3.17 zeigt die Einstellungen in dieser Datei beispielhaft für eine PostgreSQL-Datenbank.

```
#The Global properties of the registry site.
operator=www.induslogic.com
authorisedName=induslogic

URL=jdbc:postgresql://localhost/soapuddi
Class=org.postgresql.Driver
user=soapuddi
passwd=soapuddi
```

Listing 3.17: SOAPUDDI-Konfigurationsdatei

Wichtig hierbei sind die Einträge `URL`, `Class` sowie `user` und `password`, die die JDBC-URL zum Datenbankserver, die Java-Klasse für den JDBC-Treiber sowie Benutzernamen und Passwort für die Datenbank definieren.

Nach dem Neustart von Tomcat sollte sich die Applikation unter der URL *http://localhost:8080/soapuddi*, wie in Abbildung 3.13 gezeigt, melden.

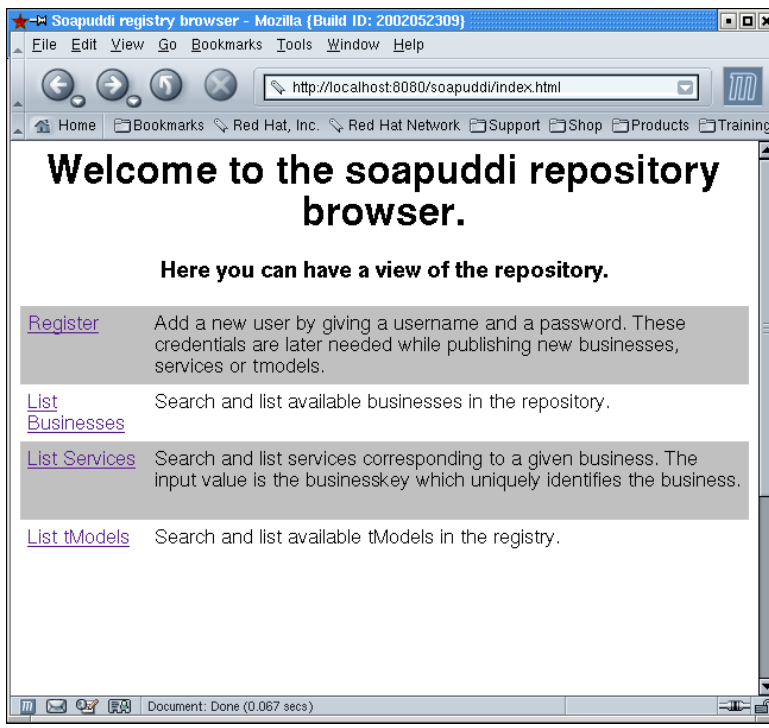

Abbildung 3.13: Einstiegsseite von SOAPUDDI

Wenn diese Schritte korrekt durchgeführt wurden, ist unter der Adresse *http://localhost:8080/soapuddi/uddi* die entsprechende UDDI-Schnittstelle von allen unseren Beispielen in diesem Kapitel zugreifbar.