Seminararbeit

Programmierung verteilter Systeme

Spezifikation von Webservices (WSDL/WSFL)

Sascha Paape
Christian-Albrechts-Universität zu Kiel
Sommersemester 2003

Betreuer
Michael Hanus

Literatur
http://www.w3.org/TR/wsdl
http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf

Inhaltsverzeichnis

1 Einleitung **4**
- 1.1 Webservices Definition . . . 4
- 1.2 Beispiele für Webservices . . . 4

2 Grundlagen **5**
- 2.1 Grundlegendes Problem der Webservices . . . 5
- 2.2 Web Services Standards . . . 5
  - 2.2.1 Webservice suchen über UDDI . . . 5
  - 2.2.2 Informationsaustausch per XML . . . 5
  - 2.2.3 Das SOAP Protokoll . . . 6
  - 2.2.4 Die WSDL Schnittstelle . . . 6
- 2.3 Web Services-Architektur . . . 6

3 Web Services Description Language (WSDL) **7**
- 3.1 Eigenschaften . . . 7
- 3.2 Aufbau eines WSDL-Dokuments . . . 7
- 3.3 Beispiel Aufbau . . . 8
- 3.4 Wiederverwendbarkeit . . . 8
- 3.5 Alternativer Aufbau . . . 8
- 3.6 WSDL-Elemente . . . 10
  - 3.6.1 Element Definitions . . . 10
  - 3.6.2 Element Import . . . 11
  - 3.6.3 Element Types . . . 11
  - 3.6.4 Element Message . . . 12
  - 3.6.5 Element PortType . . . 12
  - 3.6.6 Element Binding . . . 13
  - 3.6.7 Element Port . . . 14
  - 3.6.8 Element Service . . . 14
  - 3.6.9 Element Documentation . . . 14
- 3.7 SOAP-Bindung . . . 15
  - 3.7.1 Element soap:binding . . . 15
  - 3.7.2 Element soap:body . . . 15
  - 3.7.3 Element soap:header . . . 16
  - 3.7.4 Element soap:operation . . . 16
  - 3.7.5 Element soap:address . . . 17
- 3.8 Erzeugung von WSDL-Dateien . . . 17
- 3.9 Beispiel zur Erzeugung von WSDL-Dateien mit Idoox' IdooXoap for Java . . . 18

4 Web Services Flow Language (WSFL) 20
4.1 Beispiele für integrierte Webservices 20
4.2 Grobgliederung der Sprache 20
4.3 Implementierung des flow model 21
4.4 Implementierung des global model 24
4.5 Rekursive Komposition 25

5 Zusammenfassung 26

Kapitel 1

Einleitung

1.1 Webservices Definition

Erst seit kurzem schmückt das Wort *WebService* die Werbeseiten der Software Hersteller. In dieser Seminararbeit soll geklärt werden welche Struktur sich hinter dieser Marke verbirgt und die wesentlichen Schlagworte der neuen Applikationsentwicklung aufgezeigt.
Um die Vorstellungen über Web Services zu festigen, soll hier eine Definition gegeben werden:

> Web Services sind lose gekoppelte, wieder verwendbare Software Komponenten, die Funktionalität zu logischen Einheiten kapseln, und zugreifbar über standardisierte Internet-Protokolle sind.[3]

1.2 Beispiele für Webservices

Was bedeutet dies nun? Es geht um verteilte Anwendungen. Heutzutage wird auf das Web hauptsächlich über einen Browser zugegriffen. Der Browser wird (zwangsläufig) von einem Menschen bedient. In vielen Fällen wäre es jedoch einfacher, wenn ein spezielles Programm auf dem lokalen Rechner die Daten übers Web sendet und empfängt. Es unterhalten sich also zwei Anwendungen über das Internet miteinander. Die eine Anwendung befindet sich auf einem Webserver, die andere beim Benutzer. Ob das nun ein Desktop-PC, ein PDA oder ein Handy ist, ist nicht relevant. Was ist nun der Vorteil davon gegenüber dem Surfen mit einem Browser?

Nehmen wir ein Beispiel: Börsendaten. Die lokale Anwendung setzt sich über das Internet mit einem Webservice in Verbindung. Sie fragt ihn, welche Prozeduren und Funktionen er öffentlich zur Verfügung stellt und ruft dann eine davon auf. Wenn sie ein Ergebnis zurückgeliefert bekommt, kann sie dieses auswerten. Im Beispiel könnte das so aussehen: Die lokale Anwendung möchte alle Börsenkurse bestimmter Firmen. Wenn der Webservice eine Funktion zur Verfügung stellt, der man den Firmennamen (oder ein entsprechendes Kürzel) mitgibt und die dann den Kurswert zurückliefert, hat man das Problem gelöst, ohne sich per Hand von Webseite zu Webseite durchklicken zu müssen. Die Anwendung könnte den aktuellen Kurs sogar automatisch in einer Datenbank ablegen und die Kursentwicklung grafisch darstellen.

Ein weiteres Beispiel wäre die Möglichkeit einer Reisebuchung, bei der sich die Anwendung evtl. mit mehreren Webdiensten in Verbindung setzt, freie Hotelzimmer und Mietautos zu einem bestimmten Termin ermittelt und anschließend bucht.

Kapitel 2

Grundlagen

2.1 Grundlegendes Problem der Webservices

Das grundlegende Problem der Webservices ist immer, dass sich die unterschiedlichsten Anwendungen auf verschiedenen Plattformen und in verschiedenen Programmiersprachen entwickelt verständigen müssen. Außerdem müssen die Clients Webservices finden und herausfinden können, welche Funktionen sie zur Verfügung stellen. Diese Funktionen müssen dann auch aufgerufen werden können, ohne dass bei der Anwendungsentwicklung bereits der Funktionsname oder gar die nötigen Parameter bekannt waren.

2.2 Web Services Standards

WSDL, SOAP und UDDI sind drei Standards, die sich ergänzen und Web Services ermöglichen sollen. Alle drei Technologien sind dabei unabhängig voneinander, es ist also nicht zwingend notwendig WSDL mit SOAP und UDDI zusammen zu benutzen. Aufgrund der Unterstützung namhafter großer Firmen läuft jedoch alles auf diese Kombination hinaus. Dank Plattformunabhängigkeit und Unabhängigkeit von der Anbindung einer Programmiersprache ermöglichen diese Standards das Vergleichen und Kombinieren von Web Services.

2.2.1 Webservice suchen über UDDI

Über UDDI (Universal Description, Discovery and Integration) können Anbieter von Webdiensten in standardisierter Form bekannt geben, worum es bei ihnen geht. Entwickler von Clients können diese Daten über Hersteller und Nutzen eines Webdienstes über einen Browser auslesen (sie sind gegliedert wie ein Webverzeichnis). Durch die standardisierte Registrierung können aber auch die Clients selbst Anfragen stellen.

2.2.2 Informationsaustausch per XML

Wie oben bereits erwähnt, müssen verschiedene Anwendungen auf verschiedenen Plattformen miteinander kommunizieren können. Die Informationen, die über das Web gesandt werden, werden mit Hilfe der Extensible Markup Language (XML) dargestellt. Dieses Format ist textbasiert und sehr variabel, so dass es auf allen Plattformen und von allen Programmiersprachen ausgelesen werden kann.

2.2.3 Das SOAP Protokoll

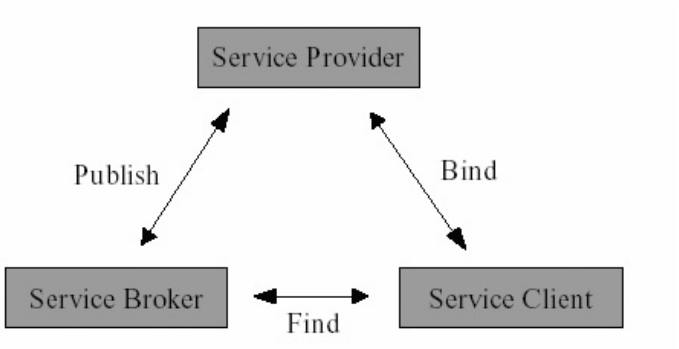
Um Operationen eines Webservices aufzurufen, wird ein Protokoll benötigt. Normalerweise wird hier SOAP (Simple Object Access Protocoll) verwendet, das die Daten (wie Funktionsnamen und Parameter) in XML-Format übermittelt. Der eigentliche Transport über das Web geschieht normalerweise über HTTP (Hypertext Transfer Protocoll).

2.2.4 Die WSDL Schnittstelle

Auch WSDL (Web Services Description Language) baut auf XML auf. WSDL stellt das Interface, also die nach außen sichtbare Schnittstelle eines Webservices dar. Dazu gehören die Operationen mit ihren Parametern und Rückgabewerten.

2.3 Web Services-Architektur

Eine Web Services-Architektur besteht typischerweise aus drei Beteiligten: Service Provider, Service Broker und Service Client.



Der Service Provider, der einen Web Service anbieten will, legt dazu auf seinem Server eine WSDL-Datei ab. Um auf seinen Web Service aufmerksam zu machen, veröffentlicht er danach seinen Web Service bei einem Service Broker. Dieser soll als Vermittler zwischen Service Provider und Service Client agieren. In dem UDDI-Register des Service Brokers ist die Adresse der WSDL-Datei hinterlegt. Der Service Client wendet sich zuerst an den Service Broker, um einen gewünschten Web Service zu suchen. Hat er einen Web Service gefunden, kann er anhand der angegebenen Adresse der WSDL-Datei diese Datei nun ausfindig machen und sie auswerten. Die Definitionen in der WSDL-Datei spezifizieren das Protokoll, die Adresse und andere Dinge, die es ihm ermöglichen, mit dem Service Provider in Verbindung zu treten, um den gewünschten Web Service zu benutzen.

Kapitel 3

Web Services Description Language (WSDL)

WSDL ist ein Standard der von den Firmen Ariba, Microsoft und IBM entwickelt wurde. Er ist noch nicht fertiggestellt, sondern befindet sich noch mitten in der Weiterentwicklung. Die Spezifikation des Standards wurde dem W3C als Vorschlag zur Beschreibung von Web Services vorgelegt. Die Veröffentlichung des Standards durch das W3C gibt dem W3C selber aber keine Ermächtnis zur Änderung oder Weiterentwicklung des Standards.

3.1 Eigenschaften

WSDL ist eine XML-basierte Beschreibungssprache für Web Services. Beschrieben wird der Nachrichtenaustausch zwischen Netzwerkendpunkten, der sowohl dokument-orientierte wie auch prozedur-orientierte Informationen enthalten kann. Die Operationen und Nachrichten werden zuerst abstrakt beschrieben und dann an ein konkretes Netzwerkprotokoll und Nachrichtenformat gebunden. WSDL ermöglicht Erweiterbarkeit, da Beschreibungen von Nachrichten erlaubt werden, unabhängig von den Nachrichtenformaten und Netzwerkprotokollen, die zur Kommunikation benutzt werden. Bisher werden in der WSDL-Spezifikation Bindungen zu den Formaten SOAP 1.1, HTTP GET/POST und MIME unterstützt. WSDL soll Teil eines Rezepts sein, das die Kommunikation von Anwendungen erleichtern und automatisieren soll.

3.2 Aufbau eines WSDL-Dokuments

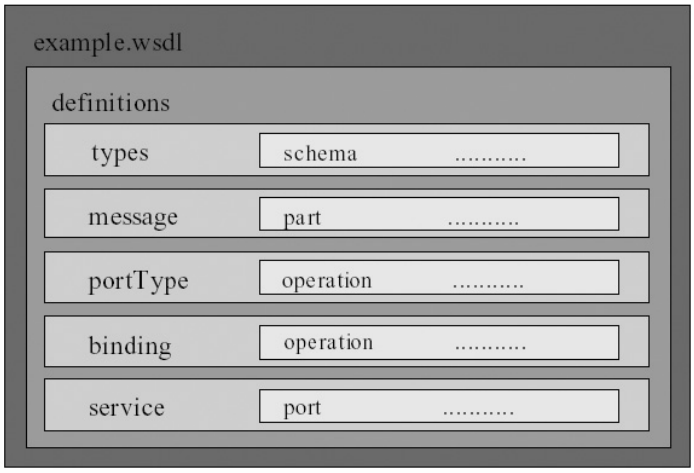
Ein WSDL-Dokument kann man sich vorstellen als eine Ansammlung von Definitionen. Diese können in zwei Gruppen aufgeteilt werden:

- abstrakte Definitionen, in diesen werden Typen, Nachrichten und Sende- bzw Empfangsoperationen definiert.
- konkrete Definitionen, in diesen werden die oben genannten Definitionen einer Bindung und einer Netzwerkadresse zugeordnet.

Zu den abstrakten Definitionen gehören die Elemente *types*, *message* und *portType*. Zu den konkreten Definitionen werden die Elemente *binding* und *service* gezählt.

3.3 Beispiel Aufbau

Die Abbildung *example.wsdl* zeigt einen beispielhaften Aufbau einer WSDL Struktur. Alle Elemente werden nach bekannten XML-Regeln mit Anfangs- und Ende-Tag eingefügt.
Das Element *definitions* ist das Wurzelelement und enthält alle Definitionen für den Webservice. Das Element *types* umfasst Datentypen, die für den Austausch von Nachrichten verwendet werden. *Message* repräsentiert eine abstrakte Definition der Nutzdatenpakete, die übertragen werden. Eine Nachricht (*message*) entspricht auf dieser Ebene noch nicht einer Methode oder Funktion. *PortType* fasst eine Menge von Operationen zusammen und in *binding* werden konkrete Protokolle und Datenformate für Operationen und Nachrichten definiert. *Port* ist eine Spezifikation einer Adresse für eine Bindung und *service* ist eine Aggregation von zusammengehörigen Ports.



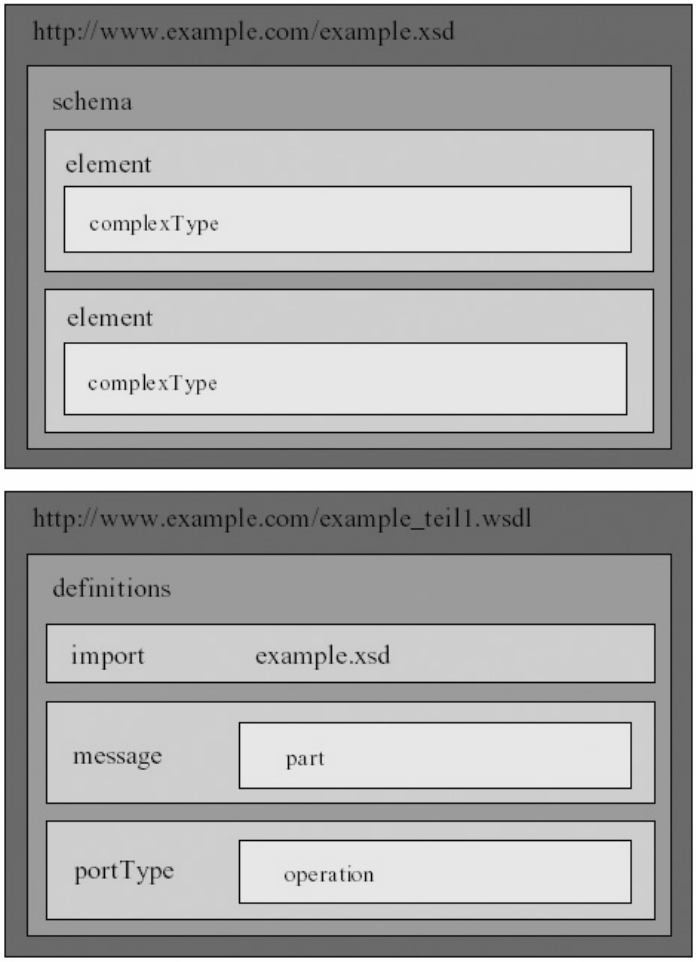
3.4 Wiederverwendbarkeit

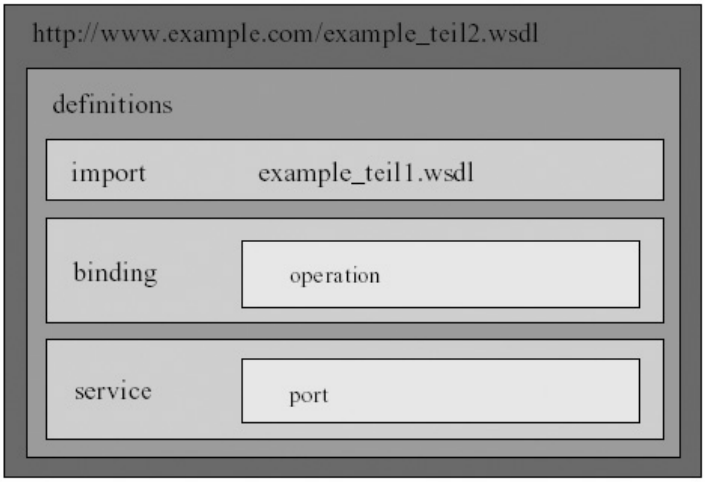
Die Dokumentstruktur des obigen Abschnitts hat zwei Nachteile. Als erstes können Dateien sehr umfangreich und somit unübersichtlich werden. Desweiteren ist eine Wiederverwendbarkeit einzelner Teile der Datei nicht möglich. Wäre Wiederverwendbarkeit möglich, könnte man z.B. das Nachrichtenelement *message* für mehrere unabhängige Web Services benutzen ohne dieses Element zweimal zu definieren.
WSDL unterstützt die Wiederverwendbarkeit durch Aufteilen der erforderlichen Elementdefinitionen auf mehrere Dateien. Die Referenz zwischen den Dateien kann durch ein *import*-Element hergestellt werden. Ein wenig Vorsicht ist geboten, damit keine Situation eintritt, in der ein Element mehrmals in verschiedenen Dateien definiert wird.

3.5 Alternativer Aufbau

In diesem Beispiel erstrecken sich die Definitionen über drei Dateien. Die Datei *example.xsd* (XML Schema Dataytype) umfasst die Datentypdefinition. Diese werden in die abstrakten Definitionen in der zweiten Datei *example teil1.wsdl* durch das *import*-Element importiert. In der dritten Datei *example teil2.wsdl* werden die Service-Bindungen definiert und zuvor die zweite

Datei wiederum durch ein *import*-Element eingefügt. Dieser Aufbau kann soweit fortgeführt werden, daß jede benötigte Elementdefinition in einer eigenen Datei spezifiziert wird und diese dann durch das *import*-Element zusammengefügt werden.





3.6 WSDL-Elemente

Dieses Kapitel soll die Elemente einer WSDL-Datei näher vorstellen.

3.6.1 Element Definitions

Dieses Element ist das äußerste Element, das alle anderen Elemente beinhaltet. Es enthält ein Attribut *name*, das dem *definitions*-Element einen eindeutigen Namen verleiht. Das *targetNamespace*-Attribut gibt den Namensraum der Datei an und die *xmlns*-Attribute machen Angaben zu XML-Namensräumen. Im unten angegebenen Beispiel werden Angaben zu Namensräumen für SOAP mit *xmlns:soap*, einer XSD-Datei mit *xmlns:xsd1*, einer WSDL-Datei mit *xmlns:tns* und WSDL selber gemacht. Die angegebene XSD-Datei ist eine Datei, in der sich Definitionen von Datentypen befinden, auf die in diesem Web Service zurückgegriffen wird. In einer WSDL-Datei benötigt man öfters einen Mechanismus, um auf schon definierte Elemente zurückgreifen zu können. Dieser Mechanismus funktioniert, indem eine Referenz durch *tns:elementname* aufgebaut wird. *Tns* ist eine Abkürzung, die für *thisnamespace* steht. Diese Bezeichnung ist nicht zwingend vorgeschrieben, hat sich allerdings als Konvention durchgesetzt. Für das Funktionieren dieses Mechanismus benötigt man jetzt noch die Angabe des Namensraums für *tns*. Genau dies wird mit dem Attribut *xmlns:tns* im *definitions*-Element gemacht, wobei als Namensraum die eigene Datei angegeben wird.

Beispiel Element Definitions

```
<definitions name="StockQuote"
    targetNamespace="http://example.com/stockquote.wsdl"
    xmlns:tns="http://example.com/stockquote.wsdl"
    xmlns:xsd1="http://example.com/stockquote.xsd"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
    ..........................
</definitions>
```

3.6.2 Element Import

Mit diesem Element werden WSDL-Elemente importiert, die in einer anderen Datei definiert worden sind. Dabei wird die Adresse der zu importierenden Datei mit einem Namensraum assoziiert.

Beispiel Element Import

```
<definitions .....................>
    <import namespace="http://example.com/stockquote/schemas"
    location="http://example.com/stockquote/stockquote.xsd" />
    ..................
</definitions>
```

3.6.3 Element Types

Das Element *types* enthält Definitionen von Datentypen. Bevorzugt wird dabei das xsd-Typsystem aufgrund seiner Plattformunabhängigkeit. In dem Element *types* wird zunächst ein Element *schema* benötigt, das nochmals Angaben zu Namensräumen macht. Im *schema*-Element können nun Elemente definiert werden. Sie haben einen eindeutigen Namen und müssen genau einem Typ entsprechen. Dieser Typ kann auch ein komplexer Typ sein. Im Beispiel werden zwei Elemente namens *TradePriceRequest* und *TradePrice* definiert. Das Erstere enthält ein Element namens *tickerSymbol* das vom Datentyp *string* ist, das Letztere ein Element namens *price* das von Datentyp *float* ist. Im zweiten Beispiel wird ein komplexer Typ gezeigt der aus Elementen unterschiedlicher Typen besteht.

Beispiel Element Types

```
<types>
<schema targetNamespace="http://example.com/stockquote.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="TradePriceRequest">
        <complexType>
            <all>
                <element name="tickerSymbol" type="string"/>
            </all>
        </complexType>
    </element>
    <element name="TradePrice">
        <complexType>
            <all>
                <element name="price" type="float"/>
            </all>
        </complexType>
    </element>
</schema>
</types>
```

Beispiel Komplexer Typ

```
<complexType name="Item">
    <all>
```

```
        <element name="quantity" type="int"/>
        <element name="product" type="string"/>
        <element name="price" type="float"/>
    </all>
</complexType>
```

3.6.4 Element Message

Das Element *message* besteht aus mehreren logischen Teilen. Diese logischen Teile sind die *part*-Elemente innerhalb des *message*-Elements. Sowohl das *message*-Element wie auch die *part*-Elemente erhalten einen eindeutigen Namen durch das *name*-Attribut. Die *part*-Elemente enthalten ein *element*-Attribut, in dem eine Referenz zu einem Datentyp hergestellt wird, und zwar zu einem jener Datentypen, die im vorangegangenen *types*-Element definiert worden sind. Im Beispiel werden zwei Nachrichten mit jeweils einem *part*-Element definiert. Das erste *part*- Element stellt eine Beziehung mit dem Element namens *TradePriceInput* her. *TradePriceInput* enthielt in der obigen Definition ein Element vom Datentyp *string*. Dementsprechend wird dem zweiten *part*-Element nun der Typ *float* zugeordnet.

Beispiel Element Message

```
<message name="GetLastTradePriceInput">
    <part name="One" element="tns:TradePriceRequest"/>
</message>

<message name="GetLastTradePriceOutput">
    <part name="Two" element="tns:TradePrice"/>
</message>
```

3.6.5 Element PortType

Dieses Element besteht aus einer Menge von Empfangs- und Sende-Operationen. Bisher wurde nur gesagt wieviele Nachrichten ausgetauscht werden sollen und von welchem Datentyp sie sind. Jetzt wird einer Nachricht die Funktion einer Sende- oder Empfangsoperation zugeordnet. Dabei gibt es vier primitive Kommunikationsformen:

- One-Way: Empfangen einer Nachricht
- Request-Response: Empfangen einer Nachricht, dann Senden einer Nachricht
- Solicit-Response: Senden einer Nachricht, dann Empfangen einer Nachricht
- Notification: Senden einer Nachricht

Das Empfangen einer Nachricht wird dabei als *input*-Element dargestellt, das Senden als *output*-Element. Die *input*- und *output*-Elemente werden zusammen in das *operation*-Element eingefügt, das ein Unterelement des *portType*-Elements ist. Sowohl das *portType*-Element wie auch das *operation*- Element besitzten einen eindeutigen Namen der durch das *name*-Attribut verliehen wird. Auch hier wird wieder der Mechanismus des Referenzierens benutzt. Dabei wird ein oben definiertes Nachrichtenelement genommen und als *input*- oder *output*-Nachricht spezifiziert. Zudem gibt es noch ein optionales *fault*-Element, das ebenfalls innerhalb des *operation*-Elements eingefügt werden kann. Es kann für den Fall eines Fehlers beim Austausch von Nachrichten herangezogen werden. Die vier Möglichkeiten werden in den kommenden Beispielen aufgezeigt.

Beispiel Element PortType: One-Way

```
<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
    </operation>
</portType>
```

Beispiel Element PortType: Request-Response

```
<portType name="StockQuotePortType">
    <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
    </operation>
</portType>
```

Beispiel Element PortType: Solicit-Response

```
<portType name=".............">
    <operation name="...........">
        <output message="............"/>
        <input message="............."/>
    </operation>
</portType>
```

Beispiel Element PortType: Notification

```
<portType name=".......">
    <operation name="............">
        <output message="................."/>
    </operation>
</portType>
```

3.6.6 Element Binding

Im Element *binding* wird die Bindung zu einem Protokoll und zu einem Nachrichtenformat hergestellt. Als Protokolle kommen nach derzeitiger WSDL-Spezifikation SOAP 1.1, HTTP Get/Post und MIME in Betracht. Jedes *binding*-Element muß die Beziehung zu genau einem Protokoll herstellen. In jedem *binding*-Element befindet sich ein *operation*-Element, das wiederum ein *input*- und *output*- Element und ein *fault*-Element enthalten kann. Auch hier gilt, daß das *binding*- und das *operation*-Element wieder einen eindeutigen Namen erhalten. Diesmal wird die Referenz zu einem definierten *portType* im *type*-Attribut vorgenommen. Dieses *type*- Attribut ist im *binding*-Tag enthalten. Im folgenden Beispiel sind alle Stellen, an denen konkrete Protokollinformationen stehen durch *extensibility*-Elemente gekennzeichnet, die im nächsten Abschnitt vorgestellt werden.

Beispiel Element Binding

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <-- extensibility element (1) -->
```

```
    <operation name="GetLastTradePrice">
        <-- extensibility element (2) -->
        <input>
            <-- extensibility element (3) -->
        </input>
        <output>
            <-- extensibility element (4) -->
        </output>
        <fault>
            <-- extensibility element (5) -->
        </fault>
    </operation>
</binding>
```

3.6.7 Element Port

Das Element *port* spezifiziert für jedes *binding*-Element genau eine Netzwerkadresse. Möchte man mehrere Netzwerkadressen für die Benutzung eines Web Service anbieten, so kann man mehrere *port*- Elemente definieren. Im Beispiel steht das *extensibility*-Element für die Stelle, an der die Netzwerkadresse, abhängig vom verwendeten Protokoll, angegeben wird. Wie gewohnt hat auch das *port*-Element ein *name*-Attribut. Die Referenz zu dem oben definierten *binding*-Element wird hier durch das *binding*-Attribut hergestellt.

Beispiel Element Port

```
<port name="StockQuotePort"
        binding="tns:StockQuoteSoapBinding">
    <-- extensibility element -->
</port>
```

3.6.8 Element Service

Das *service*-Element faßt alle *port*-Elemente zu einer Gruppe zusammen. Dabei dürfen die Ports nicht miteinander kommunizieren. Der Output eines Ports darf nicht der Input eines anderen Ports sein.

Beispiel Element Service

```
<service name="StockQuoteService">
    <port name="StockQuotePort1" binding="tns:StockQuoteSoapBinding1">
        ..............
    </port>
    <port name="StockQuotePort2" binding="tns:StockQuoteSoapBinding2">
        ..............
    </port>
</service>
```

3.6.9 Element Documentation

Dieses Element ist für den Benutzer gedacht, um den Quellcode zu dokumentieren. Es kann beliebig oft und an beliebiger Stelle eingefügt werden.

Beispiel Element Documentation

```
<service name="StockQuoteService">
    <documentation>Hier wird die Adresse der Bindung angegeben </documentation>
    <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
        ..............
    </port>
</service>
```

3.7 SOAP-Bindung

Jede Bindung benötigt spezielle Elemente, die sich nach den Anforderungen der Bindung richten. Die aktuelle WSDL-Spezifikation beinhaltet diese speziellen Elemente für SOAP 1.1, HTTP Get/Post und MIME. Alle Elemente werden in der Form

- `<soap:elementname ..... />`
  bei SOAP
- `<http:elementname ..... />`
  bei HTTP Get/Post
- `<mime:elementname ..... />`
  bei MIME

eingefügt. In diesem Kapitel wird nur die SOAP-Bindung behandelt.

3.7.1 Element soap:binding

Dieses Element verdeutlicht die Bindung zum SOAP-Format mit seiner Struktur aus Envelope, Header und Body. Es wird in das *binding*-Element eingefügt.

```
<binding .... >
    <soap:binding transport="uri" style="rpc | document"/>
    ..............
</binding>
```

Das *transport*-Attribut gibt an auf welchem Protokoll die SOAP-Daten transportiert werden. Hier kann z.B. HTTP, SMTP oder FTP verwendet werden. Das *style*-Attribut gibt Aufschluß darüber, ob eine RPC-orientierte (die Nachricht enthält Parameter und Rückgabewerte), oder eine dokument-orientierte Operation benutzt wird (die Nachricht enthält Dokumente). Wenn keine Angabe gemacht wird, dann wird von einer dokument-orientierten Operation ausgegangen.

3.7.2 Element soap:body

Das *soap:body*-Element beinhaltet mehrere Attribute, mit denen das Erscheinungsbild der Nachricht im Body der SOAP-Nachricht beeinflußt werden kann. Es liegt im *input*-, bzw *output*-Element innerhalb des *operation*-Elements im *binding*-Konstrukt. Das Attribut *parts* ist optional. Wird es weggelassen, dann sind alle *parts*, also alle Nachrichtenteile, die zuvor im *message*-Element definiert worden sind, angesprochen. Es kann also für jedes *part*-Element ein eigenes

soap:body-Element definiert werden, wobei *nmtokens* der Name des *parts* ist. Das *use*-Attribut gibt an, ob ein *part* eine encoding-Regel benutzt oder nicht. Das *namespace*-Attribut gibt wieder einen Namensraum an, und das *encodingStyle*-Attribut besteht aus einer Liste von URIs die Encodings repräsentieren, die in der Nachricht verwendet werden.

```
<binding ..............>
    <soap:binding ...........>
    <operation ..................>
        ........
        <input>
            <soap:body parts="nmtokens" use="literal | encoded"
                encodingStyle="uri-list" namespace="uri" />
        </input>
        <output>
            <soap:body parts="nmtokens" use="literal | encoded"
                encodingStyle="uri-list" namespace="uri" />
        </output>
        ........
    </operation>
</binding>
```

3.7.3 Element soap:header

Mit dem *soap:header*-Element ist es möglich, den Header der SOAP-Nachricht zu beeinflussen. Im Header können dabei beliebige Informationen wie z.B. eine User-ID eingefügt werden. Es ist am selben Platz gelegen wie das *soap:body*-Element. Die Attribute sind dieselben wie die des *soap:body*-Elements. Zusätzlich gibt es noch das *message*- Element, das den Namen der *message* angibt, die das *part*-Element enthält. Neben der *soap:header*-Definition gibt es noch eine *soap:headerfault*-Definition die an derselben Stelle gelegen ist, die gleichen Attribute hat und für eine Fehlerbehandlung zuständig ist.

```
<binding ..............>
    <soap:binding ...........>
    <operation ..................>
        ........
        <input>
            <soap:body ............/>
            <soap:header message="qname" part="nmtokens" use="literal | encoded"
                encodingStyle="uri-list" namespace="uri" />
        </input>
        ........
    </operation>
</binding>
```

3.7.4 Element soap:operation

Das *soap:operation*-Element stellt Informationen für die Operation als Ganzes zur Verfügung. Es befindet sich ebenfalls im *binding*-Element. Das Attribut *style* ist schon bekannt aus der Definition von *soap:binding*. Das *soapAction*-Attribut spezifiziert einen Wert für den SOAP-Action-Header bei einer SOAP-HTTP-Anfrage. Dieser Wert, er muß kein URI, sondern kann

ein beliebiger String sein, kann als Identifizierung einer Absicht verstanden werden, die eine SOAP-HTTP-Anfrage verfolgt. Nützlich kann dies z.B. für Firewalls sein, um bestimmte SOAP-Anfragen zu filtern.

```
<binding ..............>
    <soap:binding ...........>
    <operation ..................>
        <soap:operation soapAction="uri" style="rpc | document" />
        <input>
            ............
        </input>
        ........
    </operation>
</binding>
```

3.7.5 Element soap:address

Das *soap:address*-Element spezifiziert die Adresse des Ports. Es wird im *port*-Element eingefügt, das ein Subelement vom *service*-Element ist. Die genaue Art der Adresse wird in dem *location*-Attribut angegeben. Dies können HTTP-, SMTP-, FTP-Adressen oder auch andere sein.

```
<service ..................>
    <port .... >
        <soap:address location="http://example.com/stockquote"/>
    </port>
</service>
```

oder

```
<service>
    <port ......>
        <soap:address location="mailto:subscribe@example.com"/>
    </port>
</service>
```

3.8 Erzeugung von WSDL-Dateien

In vielen Fällen wird ein Web Services-Entwickler keine WSDL-Datei schreiben, sondern auf andere Programmiersprachen wie Java zurückgreifen. Grund kann z.B. der relativ große Quellcode einer WSDL-Datei verglichen mit Java sein. Zum Erzeugen einer WSDL-Datei wird dann ein Toolkit benutzt, das aus der Java-Datei die WSDL-Datei generiert. Im Anschluß muß dann die WSDL-Datei eventuell noch manuell geändert werden um z.B. eine Netzwerkadresse anzugeben. Mögliche Toolkits sind z.B.

- WSTK (Web Services Toolkit) von IBM
- Idoox' IdooXoap for Java auf Basis von Jakarta Tomcat

3.9 Beispiel zur Erzeugung von WSDL-Dateien mit Idoox' IdooXoap for Java

Als erstes wird die Java-Datei *HelloWorld.java* erstellt:

```
package example1;
public final class HelloWorld {
    public String getMessage() {
        return "Hello, World!";
    }
    public static void main(String[] args) {
        HelloWorld hw = new HelloWorld();
    System.out.println(hw.getMessage());
    }
}
```

Dann kann man sich mit dem Command-Line-Tool *java2wsdl* eine WSDL-Beschreibung generieren lassen:

```
C:\> java2wsdl.bat -o %HELLOWORLD%\HelloWorld.wsdl \ -u
http://localhost:8080/soap/servlet/soap/helloworld \
http://www.idoox.com \ tutorial.helloworld.server.HelloWorld
```

Mit der Option *-o* wird der Name der WSDL-Datei festgelegt, mit *-u* die URL für das SOAP-Protokoll. Die letzten beiden Parameter geben den Namensraum und die Klassendatei an. Die generierte WSDL-Datei *HelloWorld.wsdl* sieht folgendermaßen aus:

```
<definitions targetNamespace="http://www.idoox.com"
    xmlns:"http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.idoox.com">
<message name="getMessage"/>
<message name="getMessageResponse">
    <part element="tns:el1" name="result"/>
</message>
<portType name="HelloWorld">
    <operation name="getMessage">
        <input message="tns:getMessage" name="getMessage"/>
        <output message="tns:getMessageResponse"/>
    </operation>
</portType>
<binding name="HelloWorldSOAPBinding" type="tns:HelloWorld">
    <soap:binding style="rpc" transport:"http://schemas.xmlsoap.org/soap/http"/>
    <operation name="getMessage">
        <soap:operation soapAction="" style="rpc"/>
            <input>
                <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                    namespace="http://www.idoox.com" use="literal"/>
            </input>
            <output>
                <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
```

```
                    namespace="http://www.idoox.com" use="literal"/>
            </output>
    </operation>
</binding>
<service name="JavaClasses">
    <port binding="tns:HelloWorldSOAPBinding" name="HelloWorld">
        <soap:address location="http://localhost:8080/soap/servlet/soap/helloworld"/>
<   /port>
</service>
<types xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
    <schema targetNamespace=...
        xmlns="http://www.w3.org/2000/10/XMLSchema">
        <complexType base="soap:Array" name="ArrayOfstring">
            <sequence>
                <element name="a" nullable="true" type="xsd:string"/>
            </sequence>
        </complexType>
        <element name="el0" nullable="true" type="tns:ArrayOfstring"/>
        <element name="el1" nullable="true" type="xsd:string"/>
    </schema>
</types>
</definitions>
```

Die Benutzung von WSTK von IBM geht ähnlich vonstatten. Hier werden im Unterschied zu dem gerade gezeigten Beispiel zwei Dateien generiert, die durch das *import*-Element zusammengefügt werden.
Genauso wie die WSDL-Spezifikation sind auch diese Tools noch mitten in der Entwicklungsphase und müssen der sich ändernden WSDL-Spezifikation ständig angepaßt werden.

Kapitel 4

Web Services Flow Language (WSFL)

Die Beschreibung durch WSDL stellt das public interface, die öffentliche Schnittstelle, des Web Services dar. Dieses Konzept wollen wir weiterverwenden und Web Services anhand ihrer public interfaces verknüpfen und somit neue Web Services kreieren. WSFL ist IBMs Beitrag zu dieser Problematik. WSFL ist eine XML-Grammatik, die es ermöglicht, die Arbeitsaufteilung innerhalb der Web-Service-Architektur zu beschreiben. Mit Hilfe der WSFL werden wir also einen neuen Web Service als Komposition schon existierender definieren oder andersherum gesagt für unseren neu kreierten Web Service auf schon existierende zurückgreifen und diese integrieren.

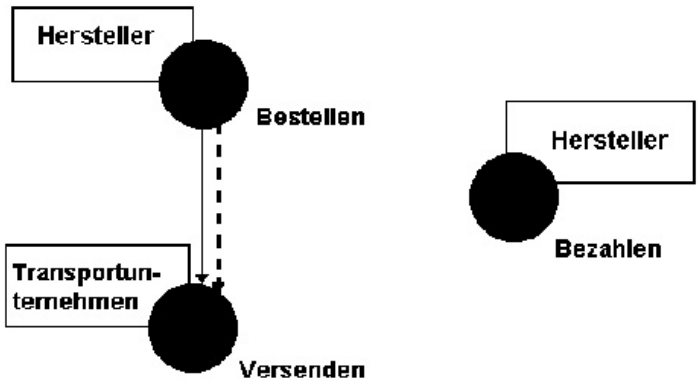
4.1 Beispiele für integrierte Webservices

Es gibt zahlreiche Beispiele für den Einsatz von integrierten Web Services. So könnte man sich z.B. einen Web Service vorstellen, der eine Reise organisiert. Bisher waren dafür mehrere Schritte notwendig: Es musste ein Zugticket bis zum Start gekauft, ein Flug gebucht und ein Taxi vom Ziel zum Hotel bestellt werden. Nehmen wir nun an, dass schon Web Services existieren, die diese einzelnen Schritte übernehmen können. Es gibt also einen Service, über den man ein Zugticket lösen kann, einen weiteren, der für die Buchung des Flugs sorgt und schließlich noch einen dritten, der das Taxi bestellt. Unser neu geplanter Web Service zur Organisation einer Reise lässt sich nun sehr einfach realisieren, indem er die anderen drei Web Services einbindet, so dass die benötigten Handlungsschritte von schon existierenden Web Services ausgeführt werden. Weitere Beispiele, z.B. das eines Händlers, werden später behandelt.

4.2 Grobgliederung der Sprache

Die Komposition von Web Services mittels WSFL erfolgt in zwei Schritten, die durch zwei verschiedene Dokumente verdeutlicht werden: das *flow model* und das *global model*. Im *flow model* wird kurz gesagt die Ablaufreihenfolge der einzelnen internen Handlungsschritte definiert, die der Web Service ausführt. Im *global model* werden die noch abstrakt gehaltenen verwendeten Web Services konkret angegeben und somit gebunden. Doch betrachten wir das alles am oben erwähnten Beispiel des allgemeinen Händlers. Der Begriff allgemeiner Händler soll hierbei verdeutlichen, dass wir keine bestimmte Sparte von Händler (Autohändler, Buchhändler etc.) betrachten, sondern nur das grundsätzliche Prinzip, das einen Händler beschreibt. Unser Händler bietet die

Produkte eines Herstellers an und versendet sie an den Käufer. Zu diesem Zweck muss er zwei Web Services integrieren, zwischen denen er sozusagen vermittelt: den Hersteller und das Transportunternehmen. Es sind die in der Abbildung als Kreise dargestellten Bearbeitungsschritte, im folgenden *Activities* genannt, notwendig. Eine *acitvity* repräsentiert eine Aufgabe, etwas, das erledigt werden muss. Die *Activities* unseres neuen Web Services Händler sind "Bestellen", "Bezahlen" und "Versenden". Für jede *activitiy* wird der entsprechende Web Service angegeben, der für die Ausführung verantwortlich ist. "Bestellen" und "Bezahlen" soll also von dem eingebundenen Web Service Hersteller ausgeführt werden, "Versenden" vom Transportunternehmen. Pfeile stellen die Reihenfolge dar, in der die Aktivitäten ausgeführt werden müssen. Somit muss die Bestellung vor dem Versenden erfolgen, die Bezahlung kann zu jedem Zeitpunkt stattfinden. Es gibt jedoch noch eine weitere Verbindung zwischen den Handlungsschritten, hier durch gestrichelte Pfeile dargestellt. Sie repräsentiert den Datenfluss. So muss die *Activity* "Bestellen" Daten an "Versenden" weiterreichen. Hierbei könnte es sich z.B. um die Adresse des Käufers handeln. Die Abbildung ist die graphische Darstellung des *flow models* unseres Web Services Händler.



4.3 Implementierung des flow model

Analog zur obigen Abbildung kann die Implementierung des *flow models* erstellt werden:

```
<flowModel name="HaendlerX" serviceProviderType="Versorgungskette">
    <serviceProvider name="HerstellerX" type="Hersteller">
        <locator type="static" service="qualitySupply.com"/>
    </serviceProvider>
    <serviceProvider name="TransportunternehmenX"
        type="Transportunternehmen">
        <locator type="static" service="worldShipper.com"/>
    </serviceProvider>
    <activity name="Bestellen">
        <performedBy serviceProvider="HerstellerX"/>
        <implement>
            <export>
                <target portType="VersorgungskettePT"
                    operation="bestelleArtikel"/>
            </export>
        </implement>
```

```
    </activity>
    <activity name="Versenden">
        <performedBy serviceProvider="TransportunternehmenX"/>
        <implement>
            <export>
                <target portType="VersorgungskettePT"
                    operation="versendeArtikel"/>
            </export>
        </implement>
    </activity>
    <activity name="Bezahlen">
        <performedBy serviceProvider="HerstellerX"/>
        <implement>
            <export>
                <target portType="VersorgungskettePT"
                    operation="bezahleArtikel"/>
            </export>
        </implement>
    </activity>
    <controlLink source="Bestellen" target="Versenden"/>
    <dataLink source="Bestellen" target="Versenden"/>
</flowModel>
```

Eine *flow model* Spezifikation beginnt mit dem *flow model*-Element als Wurzelelement. Unser *flow model* trägt den Namen "HaendlerX" und ist vom *serviceProviderType* "Versorgungskette". Hinter jedem *serviceProviderType* steckt eine Sammlung von WSDL *port types*, die ein Web Service dieses Typs implementieren muss. Dadurch werden mit dem Typ die Funktionen, die der Service zur Verfügung stellen muss, festgelegt. Da durch unser *flow model* ein neuer Web Service (der Produkte verkauft und versendet) entsteht, wird auch ein neuer *serviceProviderType* definiert, dessen öffentliche Schnittstelle aus den weiter unten im *flow model* angegebenen *port types* besteht.
Als nächster Schritt werden die Web Services angegeben, die in unseren neuen Händler Web Service eingebunden werden sollen.

```
<serviceProvider name="HerstellerX" type="Hersteller">
    <locator type="static" service="qualitySupply.com"/>
</serviceProvider>
```

Jeder Web Service wird durch ein eigenes *serviceProvider*-Element dargestellt, benannt und mit einem Typ versehen. So benötigt unser Händler einen Service Provider vom Typ "Hersteller" (der Typ fasst wieder die dahinter stehenden *port types* zusammen), der innerhalb des *flow models* unter dem Namen "HerstellerX" angesprochen werden kann.
Das *locator* Element beinhaltet die Bindungsinformationen für den jeweiligen Web Service. In unserem Fall wurde hier *static* gewählt, was bedeutet, dass die direkte Adresse des zu verwendenden Herstellers statisch als Wert des Attributs *service* angegeben wird. WSFL bietet alternativ die Möglichkeit auf lokale Programme zu verweisen (`type = "local"`), den von einer UDDI-Anfrage zurückgelieferten Service Provider einzubinden (`type = "uddi"`), die Auswahl zu einem späteren Zeitpunkt zu treffen (`type = "any"`) oder sich irgendwann während der Abarbeitung des *flows* für einen Service Provider zu entscheiden (`type = "mobility"`). Durch die Verwendung eines Service Provider *types* wird grundsätzlich nur eine Rolle (z.B. die Rolle des Herstellers) angegeben, die innerhalb unseres Händlers übernommen werden muss.

Anstatt mit *static* die Rolle einem bestimmten Service Provider direkt zuzuweisen, kann die Zuweisung zu diesem Zeitpunkt noch offen bleiben. Von dieser Tatsache machen die *types uddi*, *any* und *mobility* Gebrauch. Grundsätzlich kann man sagen, dass jeder Web Service die Rolle des Herstellers übernehmen kann, der die durch den *type* gestellten Anforderungen erfüllt, d.h. die benötigten *port types* aufweist.

Nach den Service Providern werden die benötigten Activities jeweils durch ein *activity*-Element angegeben. Wie oben erwähnt repräsentieren Activities einzelne Aufgaben und somit die Verwendung einer Operation (oder mehrerer) innerhalb des Flusses. Für jede Aktivität wird im *performedBy*-Element angegeben, welcher Service Provider diese Operation ausführt. Das Attribut *serviceProvider* verweist dabei auf einen Namen der zuvor angegebenen beteiligten Service Provider, in unserem Fall "HerstellerX" oder "TransportunternehmenX". Die Operation, sozusagen die Implementierung der Activity wird innerhalb des *implement*-Elements angegeben. In unserem Beispiel sollen alle benötigten Operationen, die eine Activity implementieren, von schon existierenden Web Services übernommen werden. Um dies zu ermöglichen, muss die Operation nach außen hin, also für eben diese externen Web Services, sichtbar gemacht werden. Dies geschieht, indem sie durch ein eingebettetes *export*-Element in das public interface unseres neuen Web Service aufgenommen werden. Somit müssen also alle Operationen exportiert werden, die in Beziehung zu einem externen Service Provider stehen. Das Gegenstück zu *export* ist *internal* und wird für alle Operationen verwendet, die eben diese Beziehung zur Außenwelt nicht benötigen und damit im public interface auch nicht auftauchen müssen.
Innerhalb des *export*-Elements gibt ein *target*-Element den Namen der Operation an, so wie er dann im public interface auftreten soll, sowie den *portType*, dem die Operation zugewiesen wird. Wie oben schon erwähnt machen diese *port types* den *serviceProviderType* aus, der als Attribut des *flowModel*-Elements angegeben wurde.

```
<activity name="Bestellen">
    <performedBy serviceProvider="HerstellerX"/>
    <implement>
        <export>
            <target portType="VersorgungskettePT"
                operation="bestelleArtikel"/>
        </export>
    </implement>
</activity>
```

Der obige Ausschnitt unseres Beispiels besagt, dass eine *activity* "Bestellen" ausgeführt werden muss, und dass dies der *serviceProvider* "HerstellerX" übernimmt. Die *operation*, welche die Implementierung der Activity darstellt, wird unter dem Namen "bestelleArtikel" innerhalb des *portTypes* "VersorgungskettePT" in das public interface unseres Händlers aufgenommen. Dadurch kann sie später mit der eines anderen Web Services verknüpft werden.
Sinnvollerweise gibt es in unserem Händler Web Service die Einschränkung, dass das Versenden erst nach dem Bestellen erfolgen kann. In der Abbildung wurden wir dieser Tatsache durch einen Pfeil gerecht, im *flow model* gibt es dafür das *controlLink*-Element.

```
<controlLink source="Bestellen" target="Versenden"/>
```

Der Wert des Attributs *source* ist der Name der zuerst auszuführenden Activity, hinter *target* befindet sich die darauf folgende. Beide Attribute müssen angegeben werden. Da eine Operation möglicherweise in mehreren Activities auftauchen kann, werden durch einen *controlLink* nicht

Operationen, sondern Activities verbunden.
Die zweite Verbindungsart, der Daten wird durch *dataLink*-Elemente dargestellt. Sie definieren also den Informationsaustausch zwischen Activities. Als Attribute werden die die Nachricht versendende Activity (als *source*) und die die Nachricht empfangende (als *target*) angegeben.

```
<dataLink source="Bestellen" target="Versenden" />
```

Obgleich in unserem Beispiel (wie in den meisten Fällen), *controlLink* und *dataLink* übereinander verlaufen, muss dies jedoch nicht zwingend sein. So könnte zum Beispiel eine bestimmte Reihenfolge erwünscht sein, obwohl keine Daten ausgetauscht werden. Andersherum ist dies allerdings nicht möglich, parallel zu jedem *dataLink* muss auch ein *controlLink* verlaufen.

4.4 Implementierung des global model

Durch das *flow model* wurde ein neuer Web Service definiert. Es wurde jedoch noch nicht angegeben, wie die Activities und Operationen unseres Händlers mit denen der eingebundenen Web Services konkret in Verbindung stehen. Dies ist die Aufgabe des *global models*.
Für unseren Händler haben wir einige Aufgaben definiert, die abgearbeitet werden müssen, damit der Prozess zu einem erfolgreichen Ende kommt. Wir haben Activities als Repräsentation für diese Aufgabe gewählt, uns überlegt, welche Operationen dahinterstecken und sie einem port type zugewiesen. Die entsprechenden Namen für Activities, port types und Operationen wurden jedoch von uns selbst gewählt, als geltende Namen innerhalb unserer Händler-Spezifikation. Nun ist es aber nicht zu erwarten, dass die entsprechenden Operationen der eingebundenen Web Services, die wir für unseren Händler verwenden wollen dieselben Namen besitzen. Aus diesem Grund werden im *global model* sogenannte *plug links* definiert, die die Verbindung zwischen zwei zusammengehörigen Operationen explizit darstellen können.

```
<globalModel name="Haendler"
    serviceProviderType="VersorgungsketteGlobal">
    <serviceProvider name="HerstellerX" type="Hersteller"/>
    <serviceProvider name="TransportunternehmenX"
        type="Transportunternehmen"/>
    <serviceProvider name="HaendlerX" type="Versorgungskette"/>
    <plugLink>
        <source serviceProvider="HaendlerX"
            portType="VersorgungskettePT"
            operation="bestelleArtikel"/>
        <target serviceProvider="HerstellerX"
            portType="supplyService"
            operation="order"/>
    </plugLink>
    <plugLink>
        <source serviceProvider="HaendlerX"
            portType="VersorgungskettePT"
            operation="bezahleArtikel"/>
        <target serviceProvider="HerstellerX"
            portType="supplyService"
            operation="pay"/>
    </plugLink>
    <plugLink>
```

```
        <source serviceProvider="HaendlerX"
            portType="VersorgungsketttePT"
            operation="versendeArtikel"/>
        <target serviceProvider="TransportunternehmenX"
            portType="shipService"
            operation="send"/>
    </plugLink>
</globalModel>
```

Wie schon das *flow model* hat auch das *global model* einen Namen und einen *serviceProviderType*. Alle *serviceProvider*, deren Operationen verknüpft werden sollen, werden aufgeführt. Somit tauchen unsere schon bekannten Service Provider "HerstellerX" und "TransportunternehmenX" auf, jedoch auch unser neu kreierter Web Service "HaendlerX" vom Typ "Versorgungskette". Als nächstes werden die *plugLink*s, also die expliziten Verbindungen angegeben. *source* und *target* definieren den Ausgangs- und den Endpunkt der Verknüpfung. Als Attribute dieser Elemente werden der entsprechende *serviceProvider*, innerhalb dessen der zugehörige *portType* und wieder innerhalb dessen die benötigte *operation* angegeben. In unserem Beispiel wird so z.B. die *operation* "bestelleArtikel" des *portTypes* "VersorgungsketttePT" des *serviceProviders* "HaendlerX" mit der *operation* "order" des *portTypes* "supplyService" des *serviceProviders* "HerstellerX" verknüpft.

4.5 Rekursive Komposition

Wie schon des öfteren erwähnt, haben wir einen neuen Web Service kreiert. Durch das Attribut *serviceProviderType* des *flowModel*-Elements wurde sein Typ und somit seine öffentliche Schnittstelle angegeben. Mit diesen Daten kann nun dieser neue Web Service mit anderen kombiniert und zu einem wiederum neuen Service zusammengeschlossen werden. Dies kann beliebig weitergeführt werden, da jeder Zusammenschluss wieder als Web Service verstanden werden kann.

Kapitel 5

Zusammenfassung

Web Services werden im Auftrag verschiedenster Unternehmen entwickelt und von verschiedenen Programmierern unterschiedlich implementiert. Dennoch sollen sie in der Lage sein, miteinander zu kommunizieren und sogar selbstständig untereinander zu agieren. Der eben vorgestellte Ansatz verwirklicht dies, indem auf das von der Implementierung unabhängige nach außen sichtbare Verhalten, das Interface, zugegriffen wird. Web Services bleiben so flexibel bezüglich der Zusammenarbeit mit anderen Web Services.

Literaturverzeichnis

[1] W3 Web Services Description Language
http://www.w3.org/TR/wsdl

[2] IBM Webservices
http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
http://www-3.ibm.com/software/solutions/webservices

[3] T-LAN Glossar
http://www.t-lan.de/GLOSSAR/

[4] Web Services Essentials
http://www.oreilly.de/catalog/webservess/chapter/ch06.html