Seminar: Agenten in verteilten Systemen

Diensteorientierte Architektur (SOA)

Kristof Hamann

SoSe 2006

Universität Hamburg
Fachbereich Informatik

Inhaltsverzeichnis

1 Diensteorientierte Architektur

Der Begriff „dienstorientierte Architektur“ (*service-oriented architecture*, *SOA*) ist insbesondere durch die Beliebtheit von *Webservices* in aller Munde. Oft einfach gleichgesetzt, sind die beiden Begriffe jedoch wohl zu unterscheiden. Während es sich bei Webservices um einen konkreten Standard handelt, sollen durch die Bezeichnung „diensteorientierte Architektur“ vielmehr die dahinter stehenden Ansätze in allgemeiner, wissenschaftlicher Form zusammengefasst werden. Webservices werden in Kapitel 2 behandelt, während in diesem Kapitel die diensteorientierte Architektur im Vordergrund steht.

1.1 Einführung

Bei der diensteorientierten Architektur handelt es sich also nicht um eine konkrete Technik, sondern vielmehr um ein Paradigma. Mit der Bezeichnung *Architektur*[1] ist hier natürlich nicht die Kunst Häuser zu bauen gemeint, sondern die Baukunst in Bezug auf Software – also eine Beschreibung der grundelegenden Elemente und der Struktur von Softwaresystemen.

Die zentrale Idee ist, Geschäftsprozesse durch Dienste abzuwickeln. Ein Geschäftsprozess bezeichnet eine Folge von Aktivitäten um ein Geschäftsresultat zu erzielen. Dabei soll ein Dienst mit seiner Funktionalität einen einzelnen Schritt oder einen kompletten Geschäftsprozess abwickeln. Zur Erfüllung kann der Dienst andere Dienste nutzen, wie auch Geschäftsprozess auf andere Geschäftsprozesse basieren können.

[1] aus dem Griechischen von *arché* (Anfang, Ursprung, Grundlage, das Erste) und *techné* (Kunst, Handwerk)

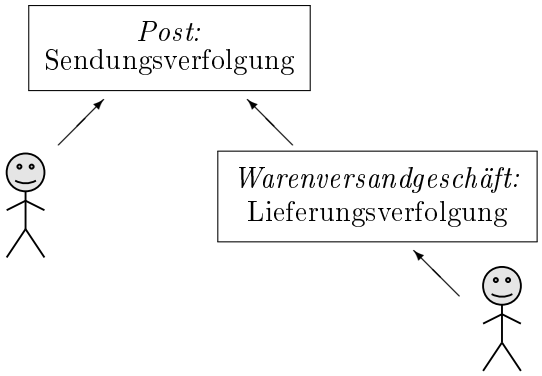

Abbildung 1.1: Beispiel für den Zugriff auf verschiedene Dienste

Ein Beispiel für zwei Dienste und den Zugriff auf diese zeigt Abbildung 1.1. Hierbei teilt ein Postunternehmen seinen Kunden beim Paketversand eine Tracking-Nummer mit. Dieser kann nun bei dem Dienst *Sendungsverfolgung* unter Angabe dieser Nummer den Fortschritt seiner Sendung abfragen. Der Dienst antwortet beispielsweise mit „Paketzentrale Hamburg–Mitte“.

Nehmen wir nun ein Unternehmen hinzu, welches Waren verkauft und seinen Kunden über das Postunternehmen beliefert. Ein Dienst soll dem Kunden eine Lieferungsverfolgung anbieten. Der Kunde kann unter Angabe seiner Bestellnummer den Stand seiner Bestellung abfragen. Der Dienst fragt nun unter Berücksichtigung der lokal gespeicherten Tracking-Nummer den Dienst des Postunternehmens ab und kann seinem eigenen Kunden eine entsprechende Antwort übermitteln.

1.2 Motivation

In diesem Abschnitt soll nun aus drei unterschiedlichen Blickwinkeln der Einsatz einer dienstorientierten Architektur motiviert werden.

1.2.1 Die softwaretechnische Sicht

Zunächst wagen wir einen Blick in die Vergangenheit: Eine kleine Geschichte der Softwareentwicklung soll uns die Sinnhaftigkeit einer diensteorientierten Architektur verdeutlichen.

Zu Beginn der Softwareära wurde der Code für jede Aufgabe, d. h. für jedes Programm, neu geschrieben. Mit der Zeit wurden die Programme größer und es wurde festgestellt, dass so unnötige Arbeit geleistet wird. Daher begonn man Funktionen und Prozeduren aus alten Programmen wieder zu verwenden und übernahm diese in neue Programme. Wurde jedoch ein Fehler gefunden, war die Korrektur schwierig, da alle Programme, in denen die fehlerhafte Funktion benutzt wird, individuell angepasst werden mussten.

Um dieses Problem zu umgehen, hat man größere Programmeinheiten in Module mit wohldefinierten Schnittstellen aufgeteilt und in Bibliotheken gesammelt, die von verschiedenen Programmen gemeinsam genutzt werden, während Implementierungsdetails durch Datenkapselung verborgen bleiben. Aber auch dieser Ansatz hat den Nachteil, dass die fehlerkorrigierte Bibliothek auf allen Installationen, in denen die fehlerhafte Version benutzt wird, verteilt werden muss. Darüber hinaus ist ein erhöhter Verwaltungsaufwand nötig, wenn verschiedene Versionen der gleichen Bibliothek parallel installiert sein müssen.

Nun soll ein neuer Ansatz diese Probleme vermeiden. Dabei wird der Code nur noch an einem Ort gehalten, wodurch Fehlerkorrekturen nur dort durchgeführt werden müssen. Auf die Funktionalität, die der Code bereitstellt, kann aber von verschiedenen Orten zugegriffen werden. Dieser Ansatz wird von der diensteorientierten Architektur verfolgt.

1.2.2 Die unternehmerische Sicht

Organisationen sind meist in einer heterogenen Struktur gegliedert, die durch autonome Abteilungen bestimmt ist. Dies hat zur Konsequenz, dass sich häufig jede Abteilung ihre eigene IT-Infrastruktur beschafft. Oft kommt dann der Wunsch auf, die gesamte IT zu zentralisieren um diese gemeinsam nutzen zu können. Dabei stehen jedoch einige Hürden im Weg:

- Die autonome Verwaltung der Abteilungen soll sich nach dem Willen der Entscheidungsträger auch in den Informatiksystemen wiederspiegeln. Jede Abteilung will für seine eigenen Systeme verantwortlich und nicht von anderen abhängig sein.
- Im Sinne des Datenschutzes wird häufig gefordert, dass bestimmte Daten in einer Abteilung lokal gespeichert werden, diese nicht verlassen und insbesondere der Zugriff darauf fest geregelt ist. Eine Zentralisierung macht dies zwar nicht unmöglich, aber dennoch ist ein erhöhter Aufwand nötig, diesem Anspruch gerecht zu werden.
- Die Zentralisierung der Infrastruktur bedingt auch die Zentralisierung der Fehlerquellen und somit der Ausfallsicherheit. Während der Ausfall eines Systems in der bisherigen Struktur dazu führt, dass lediglich eine Abteilung nicht mehr arbeiten kann während die restlichen Abteilungen nicht betroffen sind, ist bei Ausfall des zentralen Systems die gesamte Organisation lahmgelegt. Insbesondere die Angst vor einem solchen Totalausfall motiviert die Ausschau nach Alternativen.
- Darüber hinaus ist die Zentralisierung mit hohen Kosten verbunden, da eine komplette Umstrukturierung vorgenommen werden muss.

Um den großen Schritt zur Zentralisierung nicht gehen zu müssen, wird versucht, die bisherige Struktur beizubehalten, mit dem Ziel diese über Abteilungsgrenzen gemeinsam zu nutzen. Dabei ergeben sich jedoch auch einige Schwierigkeiten:

- Die lokale Datenhaltung führt dazu, dass auf Daten einer anderen Abteilung nicht direkt zugegriffen werden kann. Ein Dienst könnte jedoch über eine Schnittstelle bestimmte Aktionen möglichen.
- In vielen Organisationen dominieren *legacy*-Anwendungen[2] große Bereiche der Software. Da eine Neuentwicklung zu kostspielig ist, können Dienste den Zugriff von neueren Anwendungen ermöglichen und somit die Weiternutzung der alten Anwendungen erlauben.
- Oft setzen die verschiedenen Abteilungen komplett unterschiedliche Hard- und Software ein, die eine Interaktion zwischen den dort laufenden Programmen schwierig macht. Dienste mit plattformunabhängiger Schnittstelle können den Zugriff über diese Grenzen hinweg ermöglichen.

[2] engl. für Altlasten – also Anwendungen, die in nicht mehr benutzen Programmiersprachen geschrieben wurden oder auf nicht mehr benutzer Hardware laufen

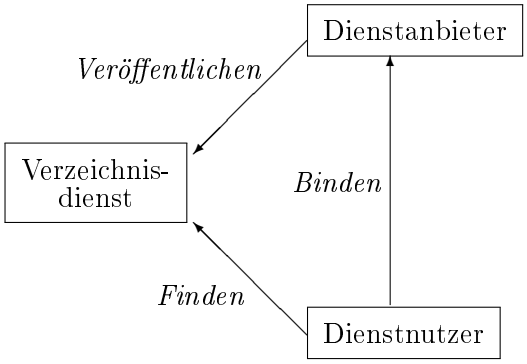

Abbildung 1.2: Struktur einer diensteorientierten Architektur

1.2.3 Die betriebswirtschaftliche Sicht

Ein Unternehmen muss in der Lage sein, sich ständig an neue Geschäftsprozesse anzupassen – beispielsweise bedingt durch die Erschließung neuer Märkte. Dabei muss der Kontakt zu neuen Kunden aufgenommen und diese als Geschäftspartner gewonnen werden. Eine dienstorientierte Architektur erlaubt das dynamische Nutzen neuer Dienste in Geschäftsprozessen und den Zugriff auf die Dienste neuer Kunden. Vielleicht kann es sich als sehr lukrativ herausstellen, eigene Dienste seinen Kunden zu öffnen und somit eine weitere Einnahmequelle zu schaffen.

1.3 Komponenten

Eine diensteorientierte Architektur beinhaltet im Wesentlichen drei Rollen.

- Dienstanbieter (*service provider*): Bereitstellung der Schnittstelle auf eine bestimmte Funktionalität.
- Dienstnutzer (*service consumer*): Klient, der auf den Dienst zugreift.
- Verzeichnisdienst (*service registry*): Verwaltet Informationen über Dienste und bietet Suchmöglichkeiten an (siehe Unterabschnitt 1.3.3)

Der Dienstanbieter veröffentlicht seinen Dienst bei einem Verzeichnisdienst. Dabei gibt er Informationen über seine Schnittstelle, seinen Ort und die Funktionalität des Dienstes an. Ein Dienstnutzer findet mit Hilfe des Verzeichnisdienstes einen konkreten Dienst und bindet sich an diesen um dessen Funktionalität zu nutzen (siehe Abbildung 1.2).

In diesem Abschnitt werden nun die einzelnen Komponenten der diensteorientierten Architektur näher erläutert und in Zusammenhang gebracht.

1.3.1 Dienst

Ein Dienst erfüllt eine Funktionalität in einem Geschäftsprozess. Bei der Auswahl von Diensten sollte berücksichtigt werden, dass diese Funktionalität wiederverwendbar, aber grobkörnig ist. So macht es keinen Sinn, technisch definierte Teile einer Anwendung – wie beispielsweise Ersetzungsoperationen an Strings – über einen Dienst bereitzustellen. Vielmehr soll ein Dienst einen kompletten Schritt innerhalb eines Geschäftsprozesses abbilden.

Um die Wiederverwendbarkeit zu garantieren, muss der Dienst über eine wohldefinierte Schnittstelle verfügen. Diese muss plattformunabhängig sein, um von allen Hard- und Softwarearchitekturen, sowie allen Programmiersprachen auf den Dienst zugreifen zu können.

Wie bei Modulen sollen auch bei Diensten die Details zur konkreten Implementierung verborgen werden (Geheimnisprinzip). Jedoch darf hier nicht nur der konkrete Code zur Erfüllung der Funktionalität nicht sichtbar sein, darüber hinaus soll nicht erkennbar sein, in welcher Programmiersprache und auf welcher Architektur der Dienst implementiert ist.

In der Regel soll ein Dienst auch Daten kapseln. Beispielsweise soll es nicht möglich sein, direkt auf die in der Buchhaltung gespeicherten Daten zuzugreifen. Statt dessen darf der Zugriff nur über die Schnittstelle eines Dienstes möglich sein, der die Zugriffsberechtigung prüfen kann.

Um nicht in komplizierte Abhängigkeitsprobleme zu verfallen, fordern wir eine lose Kopplung unter den Diensten.

Fast trivial klingt die Forderung nach Kommunikation: Es muss die Möglichkeit bestehen, auf einen Dienst zuzugreifen. Dies kann aber auf zwei unterschiedliche Arten geschehen. Ein klassischer Dienst besitzt Netzwerkschnittstellen, so dass von anderen Rechnern im Netzwerk (möglicherweise aus dem gesamten Internet) auf den Dienst zugegriffen werden kann. Dieser Zugriff erfolgt meist durch den Austausch von Nachrichten (siehe Unterabschnitt 1.3.2). Der Netzwerkzugriff ist jedoch nicht zwingend notwendig. Sollen auf einem System legacy Anwendungen in die neue Software integriert werden, kann es ausreichen, wenn lediglich lokal ein Zugriff auf den Dienst ermöglicht wird.

1.3.2 Nachricht

Die Kommunikation zwischen Dienstanbieter und Dienstnutzer wird über Nachrichten geregelt. Der Dienstnutzer schickt dabei dem Dienstanbieter eine Nachricht und erhält von diesem eine Antwort.

Wichtig ist, dass eine plattformunabhängige Nachrichten-Technologie verwendet wird, damit die Nutzung eines Dienstes nicht auf eine bestimmte Plattform eingeschränkt wird. Meistens wird daher XML[3] verwendet, da dieses Format plattformunabhängig und sehr flexibel erweiterbar ist. Von Vorteil ist außerdem, dass es relativ einfach vom Menschen lesbar ist, was insbesondere bei der Fehlersuche helfen kann. Bibliotheken zum Parsen von XML-Dateien gibt es nahezu für jede Programmiersprache.

[3] Extensible Markup Language, ein Standard für die Strukturierung von Dokumenten

Zu beachten ist jedoch unbedingt, dass das verwendete Nachrichtenformat substantiell ist. Wird das Format geändert, so gleicht dies einem Ändern der Schnittstelle des Dienstes, denn auch in diesem Fall kann der Dienstnutzer ohne Änderung seiner Programmierung nicht mehr auf den Dienst zugreifen.

1.3.3 Verzeichnisdienst

Ein Verzeichnisdienst hat die Aufgabe Informationen über Dienste zu verwalten. Dafür meldet sich ein Dienstanbieter am Verzeichnis an und teilt ihm dabei Informationen über seine Schnittstelle, seinen Ort und die Funktionalität seines Dienstes mit. Der Verzeichnisdienst bietet Dienstnutzern Suchmöglichkeiten an, damit diese die von ihnen benötigten Dienste finden können.

Für den Dienstnutzer bedeutet dies, dass er die Adresse des Dienstanbieters nicht hartkodiert im Programmcode speichern muss, sondern den Dienst erst zur Laufzeit wählen brauch. Dabei kann er sich dynamisch an die jeweiligen Begebenheiten anpassen und so den kostenmäßig günstigsten Dienst wählen oder einfach bei Ausfall eines Dienstes auf einen anderen, noch verfügbaren, zurückgreifen.

1.4 Fazit

Die diensteorientierte Architektur erlaubt das Wiederverwenden bestehender Funktionalitäten auf einer neuen Ebene. Nicht der Code wird kopiert, sondern über ein Netzwerk auf die Funktionalität selbst zugegriffen. Dadurch wird die Integration von heterogenen Systemen stark erleichtert, da der Zugriff auf einen Dienst plattformunabhängig ist. Hat sich erst einmal die diensteorientierte Architektur mit einer Menge von Diensten etabliert, so lassen sich komplexe Anwendungen relativ einfach durch Zusammenführen verschiedener Dienste realisieren. Mit einem Verzeichnisdienst können bestimmte Parameter eines Dienstes, wie die Adresse des Dienstes, transparent geändert werden. Der Dienstnutzer kann einen Dienst dynamisch finden und nutzen.

Allerdings ist die Erreichbarkeit eines Dienstes an die des angeschlossenen Netzwerkes gebunden: Fällt dieses aus, kann der Dienst nicht genutzt werden. Von der anderen Seite betrachtet, müssen aber gerade weil eine Funktionalität nun im Netzwerk verfügbar ist, neue Sicherheitsaspekte berücksichtigt werden, die bei einem lokalen Programmteil so nicht notwendig waren. Daher sollte beispielsweise überlegt werden, ob ein Dienst Authentifikation benötigt oder die Daten lieber verschlüsselt überträgt. In einem Netzwerk mit verschiedenen Diensten, die einander nutzen, kann die Fehlersuche deutlich erschwert werden, da sich Effekte über mehrere Knoten hinweg propagieren können. Wir haben plötzlich ein verteiltes System vorliegen, und müssen daher die Kehrseiten davon berücksichtigen – hier sei als Beispiel nur die Zeitsynchronisierung genannt. Meistens gibt es für diese Probleme Lösungen, dennoch darf nicht einfach darüber hinweg gesehen werden.

Die diensteorientierte Architektur lässt sich sicherlich nicht für alles sinnvoll einsetzen. Da die Kommunikation immer vom Dienstnutzer ausgeht, lassen sich Ereignisse nicht implementieren, da der Dienstanbieter in der Regel nicht den Dienstnutzer kennt, der

auf ein Ereignis reagieren möchte. Dieses Manko lässt sich auch nicht durch *Polling* (dabei würde der Dienstnutzer regelmäßig beim Dienstanbieter anfragen, ob ein Ereignis eingetreten ist) sinnvoll lösen, da entweder wichtige Ereignisse verpasst werden können oder aber Dienstanbieter und Dienstnutzer ununterbrochen mit Polling beschäftigt sind. Echtzeitsysteme sind somit ebenfalls nicht möglich.

2 Webservices

Webservices sind ein konkreter Standard auf Basis einer diensteorientierten Architektur. Verantwortlich für die Entwicklung und Standardisierung zeigen sich OASIS[1] und W3C[2], zwei Organisationen, die für offene Standards bekannt sind. Aus diesem Grund werden auch bei Webservices verbreitete, offene Internetprotokolle verwendet. Dies hat den Vorteil, dass das gesamte Produkt frei von Lizenzkosten ist und vorhandenes Wissen über Protokolle weitergenutzt werden kann. Die verwendeten Protokolle sind relativ einfach gehalten. Sie basieren auf XML und werden in den folgenden Abschnitten vorgestellt.

In letzter Zeit wurde ein großer Rummel um Webservices veranstaltet. Viele Firmen sehen darin eine große Zukunft und so ist es für ein Unternehmen, dass etwas auf sich hält, fast obligatorisch geworden Webservices anzubieten. Die großen Internetunternehmen machen es vor: So bieten Google und Yahoo ihre Suchdienstleistungen, Amazon einen Dienst zur Abfrage von Informationen zu Büchern und eBay die Nutzung der Autkionen per Webservice an. Dies sind nur sehr wenige Beispiele für die sinnvolle Nutzung von Webservices in der Praxis.

Die zentrale Rolle bei Webservices spielen drei Protokolle (siehe Abbildung 2.1). Zur Kommunikation zwischen Dienstnutzer und -anbieter wird SOAP verwendet. Dieses Protokoll bietet *Remote-Procedure-Calls* (RPC) sowie den Austausch von XML-Nachrichten und wird in Abschnitt 2.1 behandelt. Die Schnittstelle eines Dienstes wird mit WSDL beschrieben (siehe Abschnitt 2.2). Ein Dienstanbieter meldet sich per UDDI bei einem Verzeichnisdienst an und auch die Nutzung des Verzeichnisdienstes durch den Dienstnutzer läuft über UDDI, welches in Abschnitt 2.3 beschrieben wird.

[1] Organization for the Advancement of Structured Information Standards
[2] World Wide Web Consortium

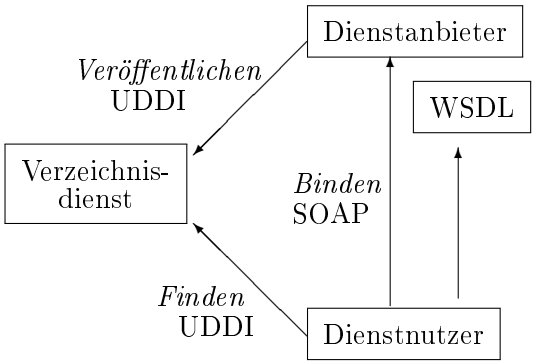

Abbildung 2.1: Die primären Protokolle bei Webservices

2.1 SOAP

SOAP stand ursprünglich für *Simple Object Access Protocol* (einfaches Protokoll für den Zugriff auf Objekte). In der aktuellen Version wird SOAP aber nicht mehr als Abkürzung aufgefasst, da das Protokoll keineswegs einfach ist und auch nicht nur den Zugriff auf Objekte ermöglicht. Entwickelt wurde es durch Microsoft, aber nachdem IBM sich anschloss wurde die Weiterentwicklung in einer Arbeitsgruppe beim W3C fortgeführt.

Die Nachrichten des Protokolls basieren auf XML, syntaktisch definiert durch XML-Schema[3]. Der Transport der Nachrichten kann von verschiedenen Transport-Schicht übernommen werden. Meistens wird HTTP[4] verwendet. Es kann aber auch HTTPS[5], SMTP[6] oder direkt TCP[7] verwendet werden. Gerade die Übertragung per HTTP hat den Vorteil, dass Firewalls leicht durchdrungen werden können, da der von HTTP genutzte Port meistens geöffnet ist.

2.1.1 Nachrichten

SOAP kennt drei Typen von Nachrichten: Die Anfrage (*Request*) sendet der Dienstnutzer an den Dienstanbieter um diesem mitzuteilen, auf welchen Dienst mit welchen Parametern zugegriffen werden soll. Der Dienstanbieter reagiert mit einer Antwort (*Response*) oder einer Fehlerbeschreibung (*Fault*).

Die XML-Nachrichten selbst bestehen aus dem Wurzelelement `Envelope` (Umschlag). Dieses beinhaltet einen optionalen Kopf (`Header`) und einen Rumpf (`Body`). Der Kopf kann aus beliebigen XML-Elementen bestehen, die nicht Teil der SOAP-Spezifikation sind. Es gibt weitere Standards, die beispielsweise das Routing von SOAP-Nachrichten (*WS-Routing*) oder die Authentifikation regeln.

Da nicht jeder Kommunikationspartner alle Standards kennen kann, sieht SOAP das Attribut `mustUnderstand` vor. Ist dieses gesetzt, darf der Empfänger der Nachricht dieses Element nicht ignorieren, weil er die verwendete Syntax und Semantik des Elements nicht versteht. Versteht der Empfänger einer Nachricht einen Teil im Kopf nicht, bei dem `mustUnderstand` gesetzt ist, so antwortet er mit einem Fehler, der beschreibt welches Element nicht verstanden wurde.

Das Attribut `actor` gibt an, für wen auf dem Pfad, den die Nachricht zu seinem Ziel einschlägt, das entsprechende Header-Element gedacht ist. So kann vor einer Menge von Diensten ein separater Dienst sich ausschließlich um die Authentifikation kümmern und (nur) bei Erfolg die Nachricht an den eigentlichen Dienst weiterleiten.

Auch der Rumpf der Nachricht ist nicht Gegenstand der SOAP-Spezifikation. Hier kann der Benutzer einen eigenen XML-Namensraum verwenden.

---

[3] neben der *Document Type Definition* (DTD) die häufigste Methode zur Beschreibung der Syntax von XML-Dokumenten

[4] *Hypertext Transfer Protocol*, zustandsloses Protokoll zur Datenübertragung, meist um Webseiten im *World Wide Web* zu übertragen

[5] per SSL geschütztes HTTP

[6] *Simple Mail Transfer Protocol*, zur Übertragung von E-Mails

[7] *Transmission Control Protocol*, verbindungsorientiertes Transportprotokoll der IP-Protokollfamilie

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP:Body>
    <ns1:BabelFish xmlns:ns1="urn:xmethodsBabelFish"
               SOAP:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <translationmode xsi:type="xsd:string">de_en</translationmode>
      <sourcedata xsi:type="xsd:string">Hallo Welt</sourcedata>
    </ns1:BabelFish>
  </SOAP:Body>
</SOAP:Envelope>
```

Abbildung 2.2: SOAP-Anfrage an einen Übersetzungsdienst

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP:Envelope xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
               SOAP:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <SOAP:Body>
    <namesp1:BabelFishResponse xmlns:namesp1="urn:xmethodsBabelFish">
      <return xsi:type="xsd:string">hello world</return>
    </namesp1:BabelFishResponse>
  </SOAP:Body>
</SOAP:Envelope>
```

Abbildung 2.3: Antwort auf die Anfrage aus Abbildung 2.2

Es werden zwei Methoden des Nachrichtenaustausches unterschieden, wobei die erste ein Spezialfall der zweiten ist. Im ersten Fall werden *Remote Procedure Calls* (entfernte Prozedur-Aufrufe, *RPC*) simuliert. Dabei wird im Rumpf der Anfrage der Name der aufzurufenden Methode und ihre Parameter samt Typen übermittelt. Die Antwort besteht aus dem Rückgabewert der Prozedur. Der zweite, allgemeinere Fall sieht vor, dass der Rumpf aus einer beliebigen XML-Nachricht besteht. Dafür wird ein eigener XML-Namensraum verwendet, der durch ein XML-Schema syntaktisch definiert ist.

Das Beispiel in Abbildung 2.2 und 2.3 zeigt die Anfrage und Antwort an einen Dienst zur Übersetzung von Sätzen in eine andere Sprache. Als Attribute des Elements `Envelope` werden die verwendeten Namensräume angegeben. Auf die XML-Syntax wird hier jedoch nicht weiter eingegangen, da dies den Rahmen der Ausarbeitung sprengen würde. Die Nachrichten sind im RPC-Stil gestaltet, verfügen jedoch über keinen Header. In Java-

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions>
  <wsdl:types/>          Datentypen
  <wsdl:message/>        Nachrichten
  <wsdl:portType/>       Schnittstellen
  <wsdl:binding/>        Dienst: Implementierung der Schnittstellen
  <wsdl:service/>        Dienst: Ort
</wsdl:definitions>
```

Abbildung 2.4: Struktur der Beschreibung eines Dienstes mit WSDL

Syntax ausgedrückt, wird folgende Methode aufgerufen: `String BabelFish(String, String)`, wobei hier der erste Parameter als `translationmode`, der zweite als `sourcedata` bezeichnet wird.

2.2 WSDL

Da Dienste plattformunabhängig sind, wird eine plattformunabhängige Methode benötigt, die Schnittstelle der Dienste zu beschreiben. Im Falle von Webservices ist WSDL die beste Wahl. Die Abkürzung steht für *Web Services Description Language*, also die Beschreibungssprache für Webservices. Eine solche maschinenlesbare Beschreibung eines Webservices ermöglicht es, automatische Codeerzeugungs-Werkzeuge zu erstellen und benutzen, die von der SOAP-Kommunikation abstrahieren und dem Programmierer den Webservice als lokale Methode anbieten, welche diesen transparent aufruft.

Eine WSDL-Beschreibung wird als XML-Datei formuliert. Sie besteht aus den Beschreibungen über die vier Bereiche Datentypen, Nachrichten, Schnittstellen und Dienste, welche im Folgenden näher erläutert werden. Abbildung 2.4 gibt eine Übersicht über die XML-Notation.

Um über eine Schnittstelle reden zu können, müssen die verwendeten *Datentypen* (`<wsdl:types/>`) bekannt sein. Diese sollten plattformunabhängig spezifiziert sein. Da sich dies schwierig gestaltet (so hat der Begriff *integer* auf verschiedenen Plattformen sehr unterschiedliche Bedeutungen) wird an dieser Stelle häufig auf die Datentypen der XML-Schema-Spezifikation zurückgegriffen. Da dies in WSDL jedoch nicht vorgeschrieben ist, können aber auch eigene Typen definiert werden.

Eine *Schnittstelle* (`<wsdl:portType/>`) besteht aus einer Anzahl von Operationen. Jede Operation wird über die beim Aufruf versendeten Nachrichten definiert – in der Regel je eine für Anfrage und Antwort.

Zu jeder *Nachricht* (`<wsdl:message/>`) wird angegeben, welcher Datentyp in ihr übertragen wird.

Die konkrete Schnittstelle eines Dienstes wird im Abschnitt `<wsdl:binding/>` angegeben. Hier werden die abstrakten Angaben aus dem Abschnitt `<wsdl:portType/>` konrektisiert, indem der Zugriff auf den Dienst genauer spezifiziert wird. So wird das

Transportprotokoll und die bei der Übertragung verwendete Kodierung angegeben.

Der Abschnitt `<wsdl:service/>` gibt schließlich an, unter welchen Adressen die Implementierung einer konkreten Schnittstelle eines Dienstes zu erreichen ist.

Die vollständige WSDL-Datei zur Beschreibung eines Dienstes wird auf einem Webserver hinterlegt. Dienstnutzer können sich diese Datei nun herunterladen und verwenden. Dabei kann die Adresse der Datei in einem Verzeichnisdienst angemeldet werden (siehe nächsten Abschnitt), damit der Dienst automatisiert gefunden wird, oder aber auf andere Weise publiziert werden, damit Interessenten die Beschreibung manuell herunterladen und in ihr Projekt integrieren können.

2.3 UDDI

Ein wichtiger Aspekt der diensteorientierten Architektur ist das dynamische Auffinden von Diensten mittels eines Verzeichnisdienstes, wie er in Unterabschnitt 1.3.3 beschrieben wurde. Für Webservices gibt es gleich mehrere Standards, die sich dieser Aufgabe annehmen. Hier wird ausschließlich UDDI (*Universal Description, Discovery and Integration*, universelle Beschreibung, Entdeckung und Integration) behandelt. Ein neuerer Standard ist die *Web Service Inspection Language* (*WS-Inspection*).

Der Verzeichnisdienst UDDI besteht aus drei Verzeichnissen: Die *White Pages* (Telefonbuch) beinhalten Beschreibungen zur Organisation und Kontaktinformationen, die *Yellow Pages* (Gelbe Seiten) eine Kategorisierung der Einträge und die *Green Pages* Informationen über die angebotenen Dienste.

UDDI verwaltet die Daten des Verzeichnisses als in XML ausgedrückte Hierarchie. Das Element `businessEntity` beinhaltet Informationen über die Organisation, Kontaktdaten, Branche und eine Liste der angebotenen Dienste. Eine allgemeine Beschreibung eines Dienstes ist im Element `businessService` zu finden, während die technischen Eigenschaften, wie die Adresse des Dienstes und der Verweis auf die Schnittstellenbeschreibung im WSDL-Format im Element `bindingTemplate` liegen. In einem `tModel` wird Kategorisierung und Referenzierung von technischen Informationen vorgenommen.

Der Zugriff auf ein UDDI-Verzeichnis erfolgt mittels SOAP. UDDI selbst ist also ein Webservice, für den eine Schnittstellenbeschreibung in WSDL existiert. Es werden zwei Schnittstellen bereitgestellt: `PublishSOAP` beinhaltet Methoden zur Authentifikation sowie dem Anlegen, Ändern und Löschen von Objekten, während `InquireSOAP` die Suche nach Objekten und Rückgabe von Details eines Objektes bietet. Der Anbieter eines Webservices nutzt also `PublishSOAP` um die Eintragung im UDDI-Verzeichnis zu verwalten. Ein Dienstnutzer wird hingegen `InquireSOAP` verwenden um die dynamische Bindung zu einem Webservice herzustellen.

Literaturverzeichnis

[1] Samudra Gupta. Service oriented architecture. http://javaboutique.internet.com/tutorials/serv_orient/article.html.

[2] Sayed Hashimi. Service-oriented architecture explained. http://www.ondotnet.com/lpt/a/4108.

[3] Wolfgang Herrmann. In zehn Schritten zur SOA. http://www.computerwoche.de/zone/soa/569662/index2.html.

[4] Pavel Kulchenko James Snell, Doug Tidwell. *Webservice-Programmierung mit SOAP*. O'Reilly, 2002.

[5] Jean-Louis Maréchaux. Combining service-oriented architecture and event-driven architecture using an enterprise service bus. http://www-128.ibm.com/developerworks/webservices/library/ws-soa-eda-esb/.

[6] TIBCO. Enabling real-time business through a service-oriented and event-driven architecture.