Thema:

Standards der Wirtschaftsinformatik

Serviceorientierte Standards im Vergleich

Diplomarbeit

Arbeitsgruppe Wirtschaftsinformatik

Themensteller/	Prof. Dr. rer. pol. habil. Hans-Knud Arndt
Betreuer:	Prof. Dr. rer. pol. habil. Hans-Knud Arndt
vorgelegt von:	Robert Schmalz
Abgabetermin:	5. Oktober 2011

Inhaltsverzeichnis

Verzeichnis der Abkürzungen und Akronyme

CDL	Choreography Description Language
CORBA	Common Object Request Broker Architecture
GIOP	General Inter-ORB Protocol
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
NS	Namespace
OMG	Object Management Group
OMG IDL	OMG Interface Definition Language
ORB	Object Request Broker
QName	Qualified Name
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
UDDI API	UDDI Application Programming Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WS	Web-Service
WS-BPEL	Web Service Business Process Execution Language
WSDL	Web Service Description Language
WS-I	Web Service Inspection Language
XML	Extensible Markup Language
XML-RPC	XML Remote Procedure Call

Abbildungsverzeichnis

Tabellenverzeichnis

1 Zusammenfassung & Fazit

Die Standards der Wirtschaftsinformatik bilden einen sehr großen Kreis. Selbst die Einschränkung auf die Serviceorientiert hält eine große Menge an Standards vor. Alle Standards konnten in dieser Arbeit nicht betrachtet und auch nicht in vollem Umfang beschrieben werden.

Die Grundlage der meisten hier dargestellten Standards bildet XML. XML bietet zahlreiche Vorteil. Die hohe Verbreitung und ständige Weiterentwicklung von XML-Standards sind nur einige Vorteile. Darüber hinaus lassen sich so ohne Schwierigkeiten neue Konstrukte bilden und auf einfache Art und Weise bestehende Systeme erweitern.

Die Anwendung von XML-basierten Web-Services wird in Zukunft weiter steigen. Eine generelle Aussage ob UDDI CORBA vorzuziehen ist, lässt sich aber nicht treffen. Da in diesem Zusammenhang auch andere Aspekte, als die technischen Gegebenheiten, eine Rolle spielen. Wenn der Umstieg auf eine neue Technologie, in einem Unternehmen, Ausfälle zur Folge hat, so sollte das bestehende System erhalten bleiben.

2 Einführung

Die Wirtschaftsinformatik bietet eine Vielzahl von Standards an. Alle zu untersuchen würde ein Diplomarbeit bei weitem übersteigen. Allein das W3C bietet mehr als 100 Standards und Entwürfe an. Eine gute Auswahl zu treffen fällt in diesem Zusammenhang nicht gerade leicht. Hinzu kommt das die Spezifikationen einiger Standards einen immensen Umfang haben. Letztendlich habe ich mich dazu entschlossen die serviceorientierten Standards näher zu betrachten.

Die serviceorientierten Standards sind ein ständig aktuelles Thema und können entscheidet zum Erfolg eines Unternehmens beitragen. Ein großer Vorteil für Unternehmen liegt in der Verwendung von XML. XML ist plattformunabhängig und an keinen Hersteller gebunden. Darüber hinaus ist XML kostenlos. Ein weiterer Vorteil liegt in der leichten Zugänglichkeit von XML.

Ziel dieser Arbeit ist es, die Grundlagen von XML für serviceorientierte Standards zu vermitteln und im späteren Verlauf auf diesen Grundlagen aufzubauen, Standards zu beschreiben und Zusammenhänge oder Unterschiede zwischen diesen herauszustellen.

3 Grundlagen

Die Grundlagen für die Standards der Wirtschaftsinformatik bilden zum einen Normen und zum anderen Techniken zur Informationserfassung, -verarbeitung und -verbreitung. Eine Basis für fast alle Standards bildet dabei XML. Im Folgenden werden auf diese eingegangen und grundlegende Begriffe erläutert.

3.1 Extensible Markup Language

Die Extensible Markup Language (XML) ist eine Weiterentwicklung der, in den 70er Jahren entstandenen, Standard Generalized Markup Language (SGML). Diese wurde zum internationalen Standard zur Repräsentation von Daten. Das World Wide Web (WWW) bildet einen weiteren Schritt von SGML zu XML. Zur Veröffentlichung von Dokumenten im WWW war es erforderlich, eine einheitliche Spezifikation zu entwickeln. Durch die Initiative des World Wide Web Consortium (W3C) wurde die Hypertext Markup Language (HTML) im März 1989 als Standard festgelegt. HTML definiert die Formatierung und Darstellung von Dokumenten im Web.[1]

Mit Hilfe von XML lassen sich strukturierte Information, lokal oder über ein Netzwerk, zwischen Programmen, Menschen, Computern und Menschen verbreiten und nutzen. Wie genau ein einfaches XML-Dokument aussieht zeigt Beispiel 3.1.1. Die Ähnlichkeit zu HTML wir hier deutlich. Dennoch gibt es strenge syntaktische Regeln für XML. Werden diese Regeln nicht befolgt können XML-Programme die Dokumente nicht korrekt bearbeiten und liefern eine Fehlermeldung zurück.[2]

Beispiel 3.1.1: XML – Buchstruktur Beispiel

```
<book isbn="123-a-456">
        <title>XML</title>
        <subtitle>Ein Beispiel</subtitle>
        <author>Max Mustermann</author>
        <publisher>Musterverlag</publisher>
        <year>2011</2011>
        <price>29.90</price>
        <description />
</book>
```

[1] Vgl. [Erl2004] S. 18 ff
[2] Vgl. [W3CXMLa]

Die Hauptunterschiede zu HTML sind das alle Elemente eines XML-Dokuments geschlossen sein, als leer markiert und alle Attributwerte in Anführungszeichen stehen müssen. Bei HTML gibt es vordefinierte Elementnamen. Solch eine Vordefinition gibt es nur für XML-Elemente die mit *xml* beginnen, welchen dann eine besondere Bedeutung beigemessen wird. Des Weiteren gibt es in XML fünf Standardzeichensätze <, >, &, " und ' welche den Zeichen <, >, &, " und ' entsprechen. Darüber hinaus lassen sich Unicodezeichen verwenden oder aber eigene Entitäten entwickeln, welche in einer Document Type Definition (DTD) definiert werden. Auf die DTD wird im Folgenden Kapitel näher eingegangen.

Es ergeben sich daraus einige Vorteile gegenüber anderen Formaten. Es wird sicher für einen bestimmten Sachverhalt eine bessere Lösung als XML geben, aber daraus resultiert dann eventuell ein höherer Aufwand beim Umwandeln und Verarbeiten des Formats im Zusammenspiel mit anderen Programmen, welche unter Umständen XML nutzen. Die Vorteile von XML sind im Detail:[3]

- Redundanzen
  Da jede XML-Markierung einen Anfangs- und Endpunkt darstellt, wie das Beispiel 3.1.1 verdeutlicht, kommt es zu einer ausführlichen Kennzeichnung der einzelnen Elemente. Allgemeine Fehler und falsche Verschachtelungen in den Strukturen können so gefunden, gemeldet und behoben werden.

- Selbstbeschreibung
  Die XML-Dokumente werden im textbasiertem Format erstellt. Dies bedeutet die Elemente und Attribute sind für den Menschen lesbar und ermöglichen es das Format in den meisten Fällen sofort zu verstehen. Des Weiteren bietet dieses Format die Möglichkeit ohne große Schwierigkeiten Fehler zu finden.

- Netzwerkeffekt
  Jedes XML-Dokument kann von jedem beliebigen XML-Werkzeug gelesen und verarbeitet werden. Diesem Werkzeug sind zwar unter Umständen spezielle XML-Markierungen nicht bekannt, aber gelesen werden kann es. Dies bedeutet, dass jedes neue XML-Dokument den Wert von jedem anderen XML-Dokument und XML-Werkzeug steigert und umgekehrt verhält es sich genauso.

XML findet in vielen Bereichen Anwendung und bildet die Basis für viele wichtige Standards. Damit diese Standards reibungslos arbeiten können, bedarf es aber mehr als die zuvor beschriebenen Regeln. Es wurden weitere Definitionen und Standards entwickelt um XML-Dokumente für diese Aufgaben zu wappnen. Dazu gehören unter

[3] Vgl. [W3CXMLa]

anderem die DTD, XML Namespaces, XML Schema und XSL. Auf diese wird nun im Folgenden eingegangen.

3.2 Document Type Definition

Die Document Type Definition (DTD) besteht aus formalisierten Definitionen aller Datenelemente die in einem bestimmten XML-Dokument vorkommen. Durch eine gegeben DTD und einem korrespondierendem XML-Dokument lässt sich das Dokument auf Korrektheit untersuchen. Das folgende Beispiel 3.2.1 zeigt wie eine DTD für das Beispiel 3.1.1 aufgebaut sein könnte. Das Beispiel 3.2.2 zeigt die Verknüpfung des XML-Dokuments mit der DTD.

Beispiel 3.2.1: DTD – Buchstruktur

```
<?xml version="1.0"?>
<!--DTD for a simple bookstructure -->
<!ELEMENT book (title, subtitle?, author+, publisher, year,
        price, description*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT subtitle (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ATTLIST book isbn CDATA #REQUIRED>
```

Beispiel 3.1.3: Externe DTD – XML Buchstruktur

```
<?xml version="1.0"?>
<!DOCTYPE book SYSTEM "book.dtd">
<book isbn="123-a-456">
        <title>XML</title>
        <subtitle>Ein Beispiel</subtitle>
        <author>Max Mustermann</author>
        <publisher>Musterverlag</publisher>
        <year>2011</2011>
        <price>29.90</price>
        <description />
</book>
```

Das Beispiel 3.2.2 zeigt wie mit Hilfe von *<!DOCTYPE book SYSTEM "book.dtd">* die im Beispiel 3.2.1 beschriebene DTD mit dem XML-Dokument verknüpft. Das hier gezeigt Beispiel geht davon aus, dass die DTD extern in einer Datei, mit dem Namen *book.dtd*, vorliegt. Eine interne Definition der DTD bildet eine weitere Möglichkeit (siehe Beispiel 3.2.3) DTD und XML-Dokument zu verknüpfen.[4]

Beispiel 3.2.3: Interne DTD – XML Buchstruktur

```
<?xml version="1.0"?>
<!DOCTYPE book [
<!ELEMENT book (title, subtitle?, author+, publisher, year,
        price, description*)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT subtitle (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ATTLIST book isbn CDATA #REQUIRED>
]>
<book isbn="123-a-456">
        <title>XML</title>
        <subtitle>Ein Beispiel</subtitle>
        <author>Max Mustermann</author>
        <publisher>Musterverlag</publisher>
        <year>2011</2011>
        <price>29.90</price>
        <description />
</book>
```

Im Beispiel 3.2.3 wurden Wurzelelement *book* mit mehreren Kindelementen definiert. Die Häufigkeit, mit der ein Element, innerhalb eines Eltern- bzw. Wurzelelements auftreten kann wird mit den Zeichen +, ? und * dargestellt. Wird dem Element kein Zeichen angefügt so muss es einmal auftreten. Das „+“ legt eine Häufigkeit von mindestens eins fest, was im Beispiel mindestens einem oder mehr Autoren entspricht. Das „?“ Zeichen definiert eine Häufigkeit von kein oder einmal. Kein oder mehrmaliges Auftreten des Elements wird durch das Zeichen „*“ deklariert.

[4] Vgl. [W3SCHOOL]

#PCDATA, *EMPTY* und *ANY* sind Möglichkeiten den Inhalt eines Elements zu beschreiben, darüber hinaus kann ein Element auch andere Elemente enthalten. *Parsed Character Data* (*#PCDATA*) erlaubt es dem Element Daten zu beinhalten, sprich ein rein textueller Inhalt. Das Schlüsselwort *Empty* deutet darauf hin, dass es sich um ein leeres Element handelt. Wird dem Element hingegen das Schlüsselwort *ANY* zugewiesen, so kann es eine Kombination von Elementen oder einfachen Informationen enthalten.

Die Deklaration ein Attributs zu einem Element kann über *<!ATTLIST Elementname Attributname Attributtyp Standardwert>* geschehen. Die Attributtypen sind in der folgenden Tabelle dargestellt.

Typ	Beschreibung
CDATA	Wert des Attributs ist beliebige Zeichenkette
(Wert1|Wert2|…)	Wert des Attributs muss einer der Werte aus der Aufzählungsliste sein
ID	Wert ist eine eindeutige ID
IDREF	Wert ist ID eines anderen Elements
IDREFS	Wert ist eine Liste von anderen IDs
NMTOKEN	Wert ist ein gültiger XML-Name
NMTOKENS	Wert ist eine Liste gültiger XML-Namen
ENTITY	Wert ist eine Entität
ENTITIES	Wert ist eine Liste von Entitäten
NOTATION	Wert ist der Name von einer Notation
xml:	Wert ist ein vordefinierte XML-Wert

Tabelle 3.2.1: DTD – Attributtypen

Quelle: [W3SCHOOL]

Der Standardwert von Attributen kann zum einen, entweder einen vordefinierten bzw. festen Wert (*#FIXED*) annehmen oder aber auch festlegen, ob das Attribut benötigt (*#REQUIRED*) bzw. nicht benötigt (*#IMPLIED*) wird.

Die zuvor schon erwähnten Entitäten werden durch die Angabe eines Namens und eines beigeordneten Wertes definiert. Beispiel 3.2.4 zeigt wie eine Entität in einer DTD im Detail definiert wird. Durch diese Definition lässt sich nun, in jedem zugehörigen XML-Dokument, die Entität durch die Verwendung von *&mvl* nutzen. Durch die Verwendung solcher Entitäten lassen sich zum Beispiel häufig verwendete Begriffe in Texten kürzen.

Beispiel 3.2.4: DTD Entität

```
<!ENTITY mvl "Musterverlag">
```

Ersichtlich wird hier, das sich ein großer Teil von Elementen auch als Attribute definieren lassen und umgekehrt. Die Übersichtlichkeit für den Menschen sollte dabei aber gewahrt bleiben. Darüber hinaus können Attribute keine Kind-Elemente enthalten, können nicht so leicht bei zukünftigen Änderungen erweiterte werden und können keine Strukturen beschreiben. Entgegen zu halten ist das Attribute nicht so leicht zu manipulieren sind.[5]

3.3 XML Namespaces

XML ermöglich es, seine eigenen XML-Elemente zu beschreiben, welches in den vorangegangen Abschnitten deutlich geworden ist. Um das Beispiel der Buchstruktur nochmals aufzugreifen, kann ein Buch einen Titel, Untertitel, Autor, Verlag usw. haben. Die gewählten Namen für diese Elemente sind aber sehr allgemein gefasst und können so auch von anderen für bestimmte eigene Beschreibung von Elementen genutzt werden. Die Bedeutung dieser Elemente kann aber eine ganz andere darstellen. Kommt es zu einem Austausch der Dokumente kann es, unter Umständen, schwierig werden, den Elementen die richtige Bedeutung zuzuordnen. Unter Titel könnte zum Beispiel auch der Titel einer bestimmten Person verstanden werden (z.B. Prof. oder Graf von …). Insbesondere wenn die Interpretation der Elemente von Maschinen durchgeführt wird kann es zu gravierenden Problemen kommen. Um die Probleme der Mehrdeutigkeit von Namengebungen in einem XML-Dokument zu lösen, wurde das Konzept der Namespaces (dt. Namensräume) entwickelt.[6]

Ein einfaches Beispiel um das Konzept von Namespaces zu erklären, bietet eine Festplatte an, auf welcher sich in unterschiedlichen Verzeichnissen gleichnamige Dateien befinden. Hier gibt es kein Konflikt zwischen den Dateien, da das Betriebssystem die Dateien über den vollständigen Pfad identifizieren kann. Das gleiche Konzept findet auch Anwendung in XML beim Verwenden von Namespaces. Im Wesentlichen handelt es sich bei Namespaces um qualifizierte Pfade für ein Element in einem bestimmten Dokument. Ein Namespace lässt sich wie in Beispiel 3.3.1 deklarieren, bestehend aus einem lokalen Namen und einem Namespace-Namen.

Beispiel 3.3.1: Namespaces

```
xmlns:book="http://www.example.com/book"
```

[5] Vgl. [W3SCHOOLb]
[6] Vgl. [Sarang] S. 7

Das *xmlns*-Präfix wird für die Deklaration von Namespaces (NS) verwendet. Dieses Präfix wird nur zur Festlegung von Namensraum-Bindungen benutzt und ist laut Definition an den NS-Namen http://www.w3.org/2000/xmlns/ gebunden. Hinter dem Präfix folgt ein Doppelpunkt und dann der Name (im Beispiel *book*), der als Kurzform für die Ressource (*http://www.example.com/book*) verwendet werden soll. Die hier verwendeten Ressourcen werden als Uniform Resource Identifier (URI) bezeichnet.[7]

URI-Verweise, welche Namensräume identifizieren, werden dann verglichen, wenn ein Name zu einem gegebenen Namensraum gehört oder wenn zwei Namen zu dem gleichen Namensraum gehören. Die beiden URIs werden als Zeichenketten behandelt und sie sind nur dann identisch, wenn beide die gleiche Sequenz an Zeichen aufweisen. Beim Vergleich spielt die Groß- und Kleinschreibung der Zeichen eine Rolle. Die im Beispiel 3.3.2 aufgeführten URIs sind somit nicht identisch.8

Beispiel 3.3.2 URI-Vergleich

```
http://www.example.com/book
http://www.example.com/BOOK
```

Der Namensraum-Name sollte die Eigenschaften der Einzigartigkeit und der Persistenz besitzen. Dabei ist es nicht das primäre Ziel, diese direkt zum Abrufen von einem Schema zu benutzen. Anzumerken ist, durch die richtige Verwaltung von einfachen URLs kann das Gleiche erreicht werden.9

Die Deklaration von einer Namensraum-Bindung wurde bereits mit Hilfe des Beispiels 3.3.1 kurz erläutert. Wie das Ganze im Detail aussieht und genau definiert ist wird im Folgenden beschrieben. In Tabelle 3.3.1 und dem Beispiel 3.3.3 wird dies nochmals veranschaulicht.

Eine Namensraum-Bindung wird durch die Verwendung von reservierten Attributen deklariert. Der Attributname muss „*xmlns*“ sein oder aber mit „*xmlns:*“ beginnen. Diese Attribute werden standardmäßig oder direkt zur Verfügung gestellt. Wenn der Attributname mit dem präfigierter Attributnamen (*PrefixedAttName*) übereinstimmt, dann gibt der *NCName* das Namensraumpräfix vor. Der Namensraumpräfix kann dazu verwendet werden Element- und Attributnamen mit dem Namensraum-Namen zu verknüpfen. Identisch verhält es sich mit dem Standard-Attributnamen (*DefaultAttName*).

[7] Vgl. [Sarang] S. 7 f; Vgl. [W3CXML-NS] Abschnitt 3
[8] Vgl. [W3CXML-NS] Abschnitt 2.3
[9] Vgl. [W3CXML-NS] Abschnitt 3

Attributname	Definition
Namespace Attributname (NSAttName)	präfigierter Attributnamen (PrefixedAttName) **oder** Standard-Attributnamen (DefaultAttName)
präfigerter Attributnamen (PrefixedAttName)	“xmlns:“ NCName
Standard-Attributnamen (DefaultAttName)	“xmlns“
NCName	XML-Name, der keine Doppelpunkte enthalten darf

Tabelle 3.3.1: Attributnamen für die Deklaration von Namensräumen

Quelle: [W3CXML-NS] Abschnitt 3

Beispiel 3.3.3 Namespace Attributname

```
<?xml version="1.0"?>
<bk:book xmlns:bk="http://www.example.com/book">
        <bk:title>XML</bk:title>
        <bk:subtitle>Ein Beispiel</bk:subtitle>
        . . .
        <bk:description/>
</bk:book>
```

Darüber hinaus gilt, der Präfix *xml* ist per Definition an den Namensraum-Namen *http://www.w3.org/XML/1998/namespace* gebunden. Es kann, aber muss nicht explizit deklariert, an anderen Namensraum-Namen gebunden, von anderen Präfixen gebunden und als Standard-Namensraum deklariert werden. Der Präfix *xmlns* wird nur benutzt um Namensraum-Bindungen zu deklarieren und ist per Definition an den Namensraum-Namen *http://www.w3.org/2000/xmlns/* gebunden. Auch hier gilt, es muss nicht deklariert, von anderen Präfixen gebunden und als Standard-Namensraum deklariert werden. Elementnamen müssen nicht *xmlns* als Präfix besitzen. Des Weiteren sind alle anderen Präfixe die mit der Sequenz *xml* beginnen reserviert, egal in welcher Reihenfolge die Buchstaben stehen.[10]

Eine weiter Möglichkeit auf einzelne Komponten bzw. Elemente zu referenzieren bietet die Verwendung von qualifizierten Namen (*QName*) an. Tabelle 3.3.2 zeigt wie diese definiert werden. Ein qualifizierter Name besteht aus zwei Bestandteilen, einem Namensraum-Namen (Präfix) und einem lokalem Namen (*LocalPart*). Der Präfix muss demnach mit einer URI durch die Namensraum-Deklaration verknüpft sein.

[10] Vgl. [W3CXML-NS] Abschnitt 3

Name	Definition
QName	präfigierter Name (*PrefixedName*) **oder** nicht präfigierter Name (*unprefixedName*)
präfigierter Name (*PrefixedName*)	Präfix“:“lokaler Teil (*LocalPart*)
nicht präfigierter Name (*unprefixedName*)	lokaler Teil (*LocalPart*)
Präfix	*NCName*
lokaler Teil (*LocalPart*)	*NCName*

Tabelle 3.3.2: Qualifizierte Namen

Quelle: [W3CXML-NS] Abschnitt 4

Wie bereits erwähnt können Namensräume auf bestimmt Schema verweisen, wie diese Schemata aufgebaut sind wird im nächsten Kapitel beschrieben.

3.4 XML Schema

Ein XML Schema stellt eine Alternative zu einer DTD dar. Im Gegensatz zu einer DTD, ist das Schema selbst in XML geschrieben und somit ist es leichter für einen Menschen zu interpretieren.[11]

Ein XML-Dokumente besteht aus Elementen und dessen Attribute, wie diese in einem XML-Dokument angeordnet sind, welchen Namen und Datentypen diese annehmen können wird in einem XML Schema festgelegt.[12] Einen Einblick in eine Schema-Definition eines XML-Dokuments wird im späteren Verlauf dieses Kapitel gegeben. Im Folgenden werden als erstes die Datentypen näher beleuchtet.

Die Datentypen eines XML Schemas lassen sich in primitive und abgeleitete Typen unterteilen. Primitive Typen können nicht durch andere Datentypen ausgedrückt werden, man spricht dann auch von atomaren Datentypen. Der ausgedrückte Wert des Datentyps kann nicht weiter zerlegt werden (Beispiel: die Zahl 1) und existieren von Anfang an. Die abgeleiteten Typen können auch einen atomaren Wert besitzen, müssen es aber nicht (Beispiel: eine Telefonnummer). Abgeleitet Datentypen sind jene, die in Bezug auf andere Datentypen definiert werden. Der Datentyp *double* kann zum Beispiel nicht durch andere Datentypen definiert werden, wohingegen ein *integer* ein spezieller Fall des allgemeinen Datentyps *decimal ist*.[13]

Um den Nutzer die Verwendung von XML Schema zu erleichtern, wurden eine Fülle von Datentypen in die Spezifikation integriert. Ein Datentyp namens *anyType* bildet in

[11] Vgl. [Sarang] S. 20
[12] Vgl. [Moos2008] S. 291
[13] Vgl. [SnTiKu02] S.197; Vgl. [W3CSCHEMAc] Abschnitt 2.5

dieser Datentyphierarchie die Wurzel. Alle anderen Datentypen sind von diesem abgeleitet. Die nächsten Ableitungen von *anyType* bilden zum einen *anySimpleType* und alle komplexen Typen. Eine Übersicht über die einzelnen Typen und von welchen sie abgeleitet sind zeigt die folgende Abbildung 3.4.1.

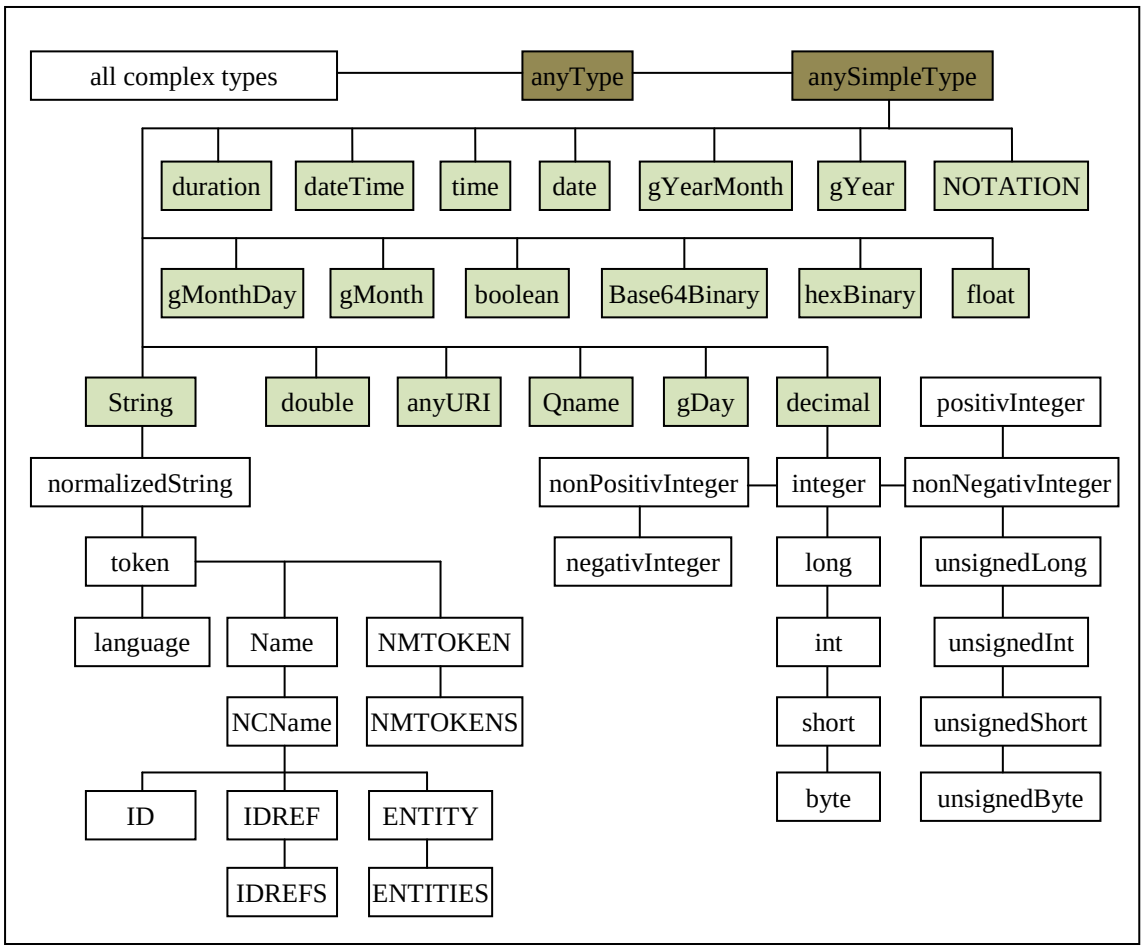

Abbildung 3.4.1: XML Schema – Datentypen

Quelle: Vgl. [W3CSCHEMAc] Abschnitt 3

anyType und *anySimpleType* sind sogenannte Urtypen. Alle abgeleiteten Datentypen von anySimpleType werden als primitive Typen und jene von *string* und *decimal* als abgeleitete Typen bezeichnet. In erster Linie werden alle Datentypen durch Einschränkung abgeleitet. *NMTOKENS*, *IDREFS* und *ENTITIES* werden anhand einer Liste abgeleitet. Die Ausnahme bilden die komplexen Typen die durch Einschränkung oder Erweiterung abgeleitet werden können.

Durch Erweiterung oder Einschränkung lassen sich so neue Datentypen bilden. Werden Datentypen durch Einschränkung neu gebildet, so wird der Wert des Datentyps auf eine bestimmte Art und Weise limitiert. Beispielsweise ist *positivInteger* von *nonNegativInteger* abgleitet, was in dem konkreten Fall bedeutet das der Wertbereich enger gefasst wurde (hier wurde lediglich die 0 von *nonNegativInteger* aus dem

Wertebereich von *positivInteger* entfernt). Handelt es sich um eine Erweiterung des Datentyps, so werden bestimmt Einschränkungen des Datentyps aufgehoben und vorher untersagte Werte können in dem neuen Datentyp verwendet werden. Ein komplexer Datentyp kann zum Beispiel aus mehreren eingeschränkten Datentypen bestehen.[14] Weitere Details zu den Datentypen lassen sich unter [W3CSCHEMAc] Abschnitt 3.2f einsehen.

In der XML Schema Spezifikation werden eine Reihe von Schema Komponenten beschrieben. Es existieren 13 Arten von Komponenten, welche sich in 3 Gruppen aufteilen lassen:[15]

- Primäre Komponenten
    - Einfache Typdefinitionen
    - Komplexe Typdefinitionen
    - Attribut-Deklarationen
    - Element-Deklarationen
- Sekundäre Komponenten
    - Attributgruppen-Definitionen
    - Identitätseinschränkungsdefinitionen
    - Modelgruppen-Definitionen
    - Notationsdeklarationen
- Hilfskomponenten
    - Annotationen
    - Modelgruppen
    - Partikel
    - Wildcards
    - Attributbenutzung

[14] Vgl. [SnTiKu02] S.197 f
[15] Vgl. [W3CSCHEMAb] Abschnitt 2.2

Um die Übersicht zu wahren und die Arbeit nicht ausufern zu lassen, wird an dieser Stelle nur auf die primären Elemente eingegangen. Für den interessierten Leser empfiehlt sich die Spezifikationen des W3C [W3CSCHEMAb] Abschnitt 2.2.

Anhand des Buchbeispiels, aus den vorrangegangen Kapiteln, werden im Folgenden die primären Komponenten eines XML Schemas beschrieben. Unter einer einfachen Typdefinition versteht man eine Reihe von Einschränkung auf Zeichenketten und Informationen über den Wert den diese darstellen. Im Detail wird eine Restriktion auf einen bestimmten Datentyp angewendet. Eine einfache Typdefinition kann auch durch Ableitung eines bereits vorhanden einfachen Datentyps neu deklariert werden.[16]

In Tabelle 3.4.1 werden einige der Einschränkungsmöglichkeiten für einen *simpleType* aufgeführt. Des Weiteren sind noch *minInclusive*, *maxInclusive*, *minExclusive*, *maxExclusive*, *totalDigits* und *fractionDigits* zu nennen, auf die an dieser Stelle, aber nicht weiter eingegangen wird.

Im Beispiel 3.4.1 wird ein Datentyp für den Buchpreis definiert. Die Restriktion auf den Basisdatentyp *string* wird durch *<xsd:pattern value="\d{3},\d{2}"/>* genau definiert. Ein zulässiger Wert wäre z.B. *123,45*. Im Beispiel 3.4.2 wird ein neuer Datentyp durch die Ableitung von *priceType* deklariert und um ein optionales EUR erweitert.[17]

Beispiel 3.4.1: Einfache Typdefinition - Buchpreis

```
<xsd:simpleType name="priceType"/>
        <xsd:restriction base="string">
          <xsd:pattern value="\d{3},\d{2}"/>
        </xsd:restriction>
</xsd:simpleType>
```

Beispiel 3.4.2: Einfache Typdefinition – Buchpreis Erweiterung

```
<xsd:simpleType name="priceTypeEUR"/>
        <xsd:restriction base="priceType">
          <xsd:pattern value="(EUR ){0,1}\d{3},\d{2}"/>
        </xsd:restriction>
</xsd:simpleType>
```

[16] Vgl. [W3CSCHEMAb] Abschnitt 2.2
[17] Vgl. [W3CSCHEMAb] Abschnitt 2.2

<table>
<tr><th>Eigenschaft</th><th>Beschreibung</th></tr>
<tr><td>length</td><td>Gibt die genaue Länge des im simpleType deklarierten Datentypes an. Dieser muss vom Wert nonNegativInteger sein.</td></tr>
<tr><td>maxLength</td><td>Gibt die maximale Länge des im simpleType deklarierten Datentypes an. Dieser muss vom Wert nonNegativInteger sein.</td></tr>
<tr><td>minLength</td><td>Gibt die minimale Länge des im simpleType deklarierten Datentypes an. Dieser muss vom Wert nonNegativInteger sein.</td></tr>
<tr><td>enumeration</td><td>Stellt die Möglichkeit bereit eine Liste von Werten zu erstellen, welche der im simpleType deklarierte Wert annehmen kann.</td></tr>
<tr><td>pattern</td><td>Stellt die Möglichkeit bereit Einschränkungen durch ein vordefiniert Muster für einen Wert zu erstellen. Der Wert von pattern muss ein regulärer Ausdruck sein.</td></tr>
<tr><td>whiteSpace</td><td>Stellt die Möglichkeit bereit abgeleitete Typen von string zu normalisieren. whiteSpace kann preserve, replace oder collapse als Wert annehmen.
<table>
<tr><th>Option</th><th>Beschreibung</th></tr>
<tr><td>preserve</td><td>Keine Normalisierung wurde vorgenommen, der Wert wurde nicht geändert.</td></tr>
<tr><td>replace</td><td>Alle auftretenden Tabs, Zeilenvorschübe und Zeilenumbrüche werden durch ein Leerzeichen ersetzt.</td></tr>
<tr><td>collapse</td><td>Alle zusammenhängenden Leerzeichen werden durch ein einzelnes Leerzeichen ersetzt.</td></tr>
</table></td></tr>
</table>

Tabelle 3.4.1: simpleType-Eigenschaftsdeklaration

Quelle: Vgl. [W3CSCHEMAc] Abschnitt 4.3

Eine komplexe Typdefinition stellt eine Sammlung bzw. Gruppierung von Datentypenelementen dar. Darüber hinaus können auch Attribute deklariert werden. Jede komplexe Typdefinition kann eine Einschränkung eines bereits vorhanden komplexen Datentyps oder eine Erweiterung eines einfachen Datentyps oder einer komplexen Basistypdefinition sein. Das Beispiel 3.4.2 zeigt eine komplexe Typdefinition anhand des komplexen Typs *authorType*. Es werden in diesem Datentyp die Elemente *name* und *forename* für einen Autor eines Buches deklariert. Eine Auswahl möglicher *element*-Attribute wird in Tabelle 3.4.2 gegeben.

Beispiel 3.4.3: Komplexe Typdefinition - Buchautor

```
<xsd:complexType name="authorType"/>
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="forename" type="xsd:string"/>
        </xsd:sequence>
</xsd:complexType>
```

Attribut	Beschreibung
minOccurs	Das Attribut minOccurse eines Elements legt fest, wie häufig ein Element mindestens vorkommen darf.
maxOccurs	Das Attribut maxOccurse eines Elements legt fest, wie häufig ein Element maximal vorkommen darf.
default	Das Attribut *default* eines Elements definiert, ob ein Element einen bestimmten standardmäßigen Wert besitzt oder nicht. Sollte kein *default*-Attribut vorhanden sein, so wird der gegebene Standardwert des Elements verwendet.
fixed	Das Attribut *fixed* eines Elements definiert, ob ein Element einen festen Wert besitzt. Wenn die Eigenschaft in einem XML-Dokument vorkommt, muss der Wert identisch mit dem *fixed*-Wert sein. Sollten die Werte nicht übereinstimmen so wird der vordefinierte fixed-Wert verwendet.

Tabelle 3.4.2: element-Attribute

Quelle: Vgl. [W3CSCHEMAc] Abschnitt 4.3

Es existieren mehrere Möglichkeiten Elemente in einer komplexen Typdefinition in Gruppen zu deklarieren. Das Inhaltsmodell (*content model*) für einen Typ bestimmen die Elemente *sequence, choice* und *all*. Ein Wert des Typs *sequence* muss alle Elemente enthalten, welche durch die *sequence*-Gruppe deklariert wurden. Die Reihenfolge, in welcher die Werte erscheinen, muss mit der Reihenfolge der Deklaration der komplexen Typdefinition übereinstimmen. Wird die komplexe Typdefinition hingegen mit dem Element *choice* deklariert so darf nur ein Wert des Typs *choice* aus der Liste der Elemente den Wert darstellen. Das *all*-Element verlangt, dass die Werte des Typs alle Elemente beinhaltet, keines der enthaltenen Elemente mehr als einmal auftreten darf und die Reihenfolge der Elemente keine Rolle spielt.

Eine weitere Möglichkeit der Gruppierung von Elementen ermöglicht das *group*-Element. Dieses kommt in zwei unterschiedlichen Variationen in der XML Schema Spezifikation vor. Es wird zum einen dazu verwendet einen Namen zu einer eigenständigen *sequence, choice* oder *all*-Gruppenmodell-Definition zuzuweisen oder diese zu repräsentieren. Bei dieser Verwendung kann dem *group*-Element ein Name zugeordnet werden auf den referenziert werden kann. Wenn das *group*-Element aber anderseits innerhalt einer komplexen Typdefinition erscheint, dann referenziert das assoziiert *ref*-Attribut zu dem namentlichen *group*-Element.

Das Beispiel 3.4.4 zeigt den Aufbau eines XML Schema ein wenig ausführlicher. In der komplexen Typdefinition von *bookType* wird auch ein Attribut namens *isbn* mit dem Typ *string* deklariert. Attribute können drei zusätzliche Eigenschaftsdefinitionen aufweisen (siehe Tabelle 3.4.1).

Beispiel 3.4.4: Komplexe Typdefinition - Buch

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:bo="http://www.example.com/book"
        targetNamespace="http://www.example.com/book">

<element name="book" type="bo:bookType"/>

<complexType name="bookType">
        <sequence>
          <element name="title" type="string"/>
          <element name="subtitle" type="string"/>
          <element name="author" type="bo:authorType"/>
          <element name="publisher" type="string"/>
          <!-- etc. -->
        </sequence>
        <attribute name="isbn" type="string">
</complexType>
<complexType name="authorType"/>
        <sequence>
          <element name="name" type="string"/>
          <element name="forename" type="string"/>
        </sequence>
</complexType>
</schema>
```

Eigenschaft	Beschreibung
use	Die Eigenschaft *use* definiert, ob ein Attribut benötigt (*required*), optional (*optional*) oder verboten (*prohibited*) ist. Die Bedeutung dieser Optionen ist vergleichbar mit den Attributen *minOccurs* und *maxOccurs* für die Elementdeklaration.
default	Die Eigenschaft *default* definiert, ob ein Attribut einen bestimmten standardmäßigen Wert besitzt oder nicht. Sollte kein *default* vorhanden sein, so wird der gegebene Standardwert des Attributs verwendet.
fixed	Die Eigenschaft *fixed* definiert, ob ein Attribut einen festen Wert besitzt. Wenn die Eigenschaft in einem XML-Dokument vorkommt, muss der Wert identisch mit dem *fixed*-Wert sein. Sollten die Werte nicht übereinstimmen so wird der vordefinierte fixed-Wert verwendet.

Tabelle 3.4.3: Attributdeklaration

Die Wiederverwendung von vorhandenen XML Schemata spart Zeit und Geld. Es ist unnötig ein neues Schema zu entwickeln, wenn bereits ein passendes Schema für die Anforderungen existiert. Um dies zu ermöglichen werden zwei XML Schema-Elemente durch die Spezifikation angeboten, welche es ermöglichen externe Schemata in ein Dokument einzubinden:

Das *include*-Element besitzt ein *schemaLocation*-Attribut, welches einen URI bereit hält. Das zu inkludierende Schema-Dokument wird mittels dieses URI identifiziert. Zu beachten ist hierbei das der Ziel Namensraum des zu inkludierenden Dokuments identisch mit dem Dokument sein muss, welches inkludiert. Das inkludierte Dokument muss nicht zwingend einen Ziel-Namensraum besitzen. In diesem Fall wird der Ziel-Namensraum vom inkludierenden Dokument übernommen.

Das *import*-Element kann zum einführen von globalen Definitionen eines Schema-Dokuments mit unterschiedlichen Ziel-Namensräumen verwendet werden. Es besitzt, genauso wie das *include*-Element, ein *schemaLocation*-Attribut und eine *namespace*-Attribut. Beide Attribute stellen eine URI-Referenz zur Verfügung. Das *schemaLocation*-Attribut zeigt mittels URI auf die Position des Schema-Dokuments welches importiert werden soll. Das namespace-Attribut identifiziert mittels URI den Ziel-Namensraum von dem importierten Schema-Dokument. Das importierende Dokument muss auf das importierte Dokument mit Hilfe des Namensraum referenzieren.

4 Serviceorientierte Standards

Mit dem steigenden Aufkommen von verteilten Systemen wurden die Probleme bei der Integration immer gravierender. Die Systeme, die vorher eigenständig arbeiteten, mussten im Verlauf dieser Entwicklung immer mehr mit anderen Systemen kommunizieren. Gründe dafür sind z.B. Redundanzen zu minimieren und eine einheitliche Datenhaltung zu schaffen. Um zu ermöglichen dass ein Austausch zwischen z.B. einem Windows- und Linux-System stattfinden konnte, war ein hoher Aufwand beim Entwickeln von spezifischen Protokollen und Formaten nötig.[18] Um diese Anforderungen zu erfüllten ist es nötig, dass der (Web-)Service über ein lokales Netzwerk oder über das Internet verfügbar ist, *standardisierte XML-Nachrichten* verwendet, *unabhängig vom Betriebssystem und der Programmiersprache* ist.[19]

Desweiteren sollten zwei zusätzliche Bestandteile erfüllt sein, welche aber nicht zwingend notwendig sind: [20]

Selbstbeschreibung: Beim Veröffentlichen eines Web-Services, ist es von Vorteil, ein öffentliches Interface hinzuzufügen. Dieses sollte eine, vom Menschen lesbare, Dokumentation und allgemeine XML-Grammatik beinhalten, um es Entwicklern zu vereinfachen den Service zu nutzen. Die XML-Grammatik hilft alle öffentlichen Methoden, Methodenargumente und Rückgabewerte des Service zu identifizieren.

Auffindung: Die Veröffentlichung sollte in einem dezentralem oder zentralem System (Registry) geschehen. Welches das Publizieren und Suchen von Services, für interessierte Nutzer, ermöglicht.

Im Folgenden werden die Bestandteile der einzelnen Service-Architekturen beschrieben. Abschließend werden ihre Zusammenhänge und Unterschiede erläutert und auf zuvor genannte Eigenschaften untersucht.

[18] Vgl. [Cerami02] S. 30
[19] Vgl. [Cerami02] S. 3
[20] Vgl. [Cerami02] S. 4 f

4.1 XML Remote Procedure Call

Die Idee des Remote Procedure Call (RPC) wurde von James E. White 1976 publiziert.[21] Der heute bekannte XML Remote Procedure Call (XML-RPC) wurde von Userland Software entwickelt und im Jahr 1998 veröffentlicht.[22] Der „Fernaufruf“ einer Prozedur im Zusammenhang in einer einfachen Client/Server-Lösung war eine der ersten Techniken einer Anwendungsverteilung bzw. Komponentenverbindung.[23]

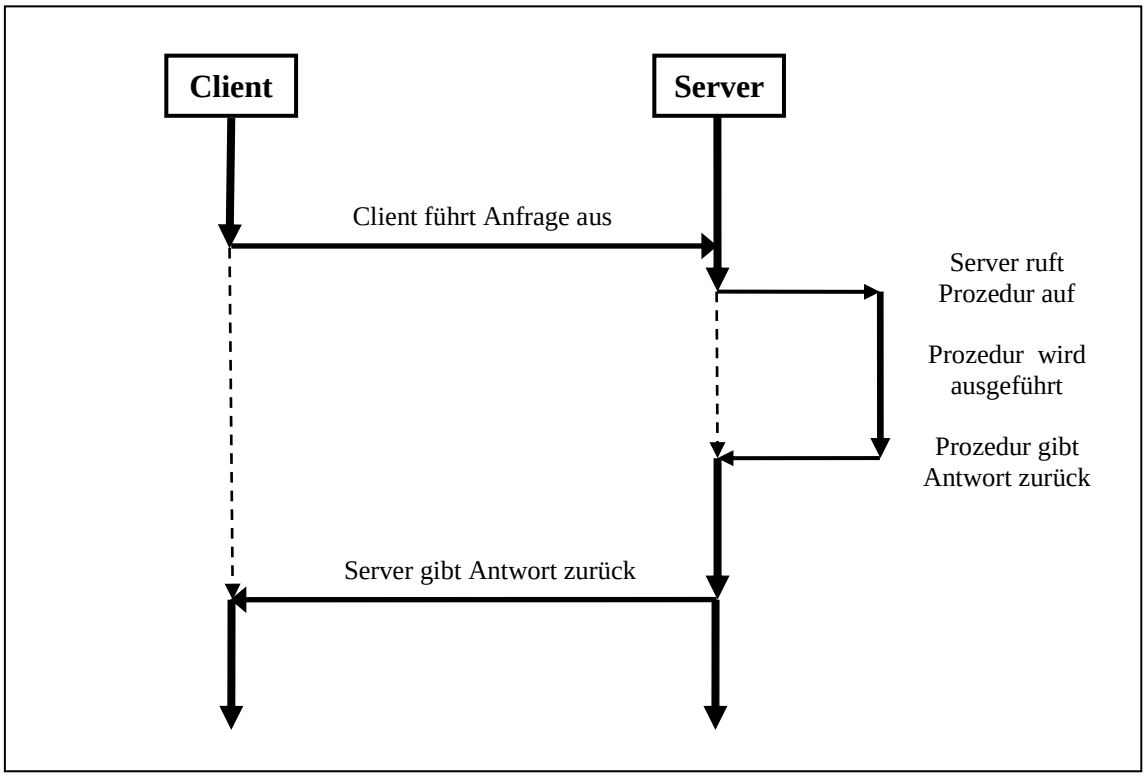

Abbildung 4.1.1: RPC einfache Darstellung

Quelle: Vgl. [Bengel04] S. 138

Abbildung 4.1.1 zeigt die einfache Darstellung des RPC. Der Client schickt eine Anfrage an den Server und dieser ruft die auszuführende Prozedur auf und gibt dann die Antwort der Prozedur an den Client zurück, falls nötig.

Der XML-RPC nutzt den zuvor beschriebenen RPC und erweiterte das vorhandene Modell durch die Verwendung von XML-basierenden Nachrichten und Transferprotokollen (siehe Abb. 4.1.2). Diese XML-Nachrichten können auf spezielle Anwendungen ausgerichtet werden und sind dadurch vom Menschen, mit Hilfe eines Texteditors, lesbar und verständlich (siehe Beispiel 4.1.1).[24]

[21] Vgl. [White76] Abschnitt 13
[22] Vgl. [Cerami02] S. 29
[23] Vgl. [Dumke03] S. 367
[24] Vgl. [Merrick06] Abschnitt 13

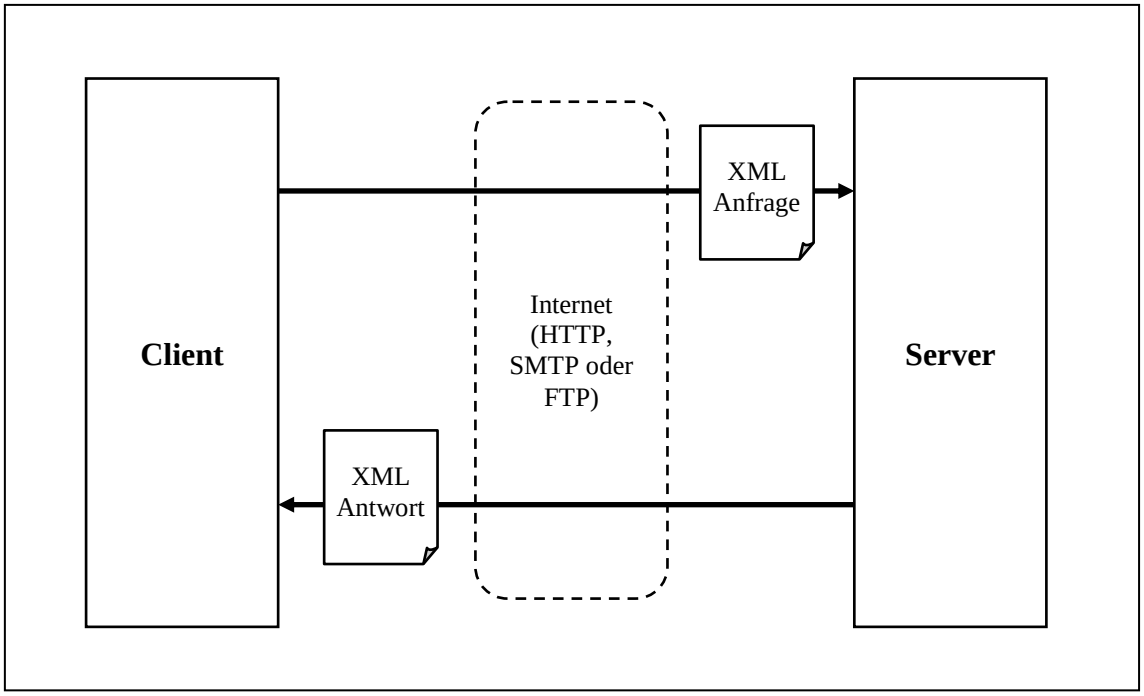

Abbildung 4.1.2: XML-RPC einfache Darstellung

Quelle: Vgl. [Merrick06] Fig. 4

Das Beispiel 4.1.1 zeigt wie eine einfache XML-RPC-Anfrage die Methode isbn.getName die ISBN eines Buches übergibt und als XML-RPC-Antwort den Namen des Buches erhält.

Beispiel 4.1.1: XML-RPC Anfrage-/Antwortbeispiel

```
<?xml version="1.0" coding="ISO-8859-1"?>
<methodCall>
        <methodName>isbn.getName</methodName>
        <params>
          <param>
               <value>0-596-00224-6</value>
          </param>
        </params>
</methodCall>

<?xml version="1.0" coding="ISO-8859-1"?>
<methodResponse>
        <methodName>isbn.getName</methodName>
        <params>
          <param>
               <value>Web Service Essentials</value>
          </param>
        </params>
</methodResponse>
```

Ein detaillierter Ablauf des RPC wird in Abbildung 4.1.3 dargestellt. Es wird der Weg des Aufrufs der Prozedur und dessen Rückgabe der Ergebnisse verdeutlicht. Das Rufen der Prozedur wird vom Anwendungsprogramm initialisiert und blockiert daraufhin. Ein entfernter Server bietet die Implementierung der Prozedur an, welche dem aufführenden Programm bekannt ist.[25]

Der Client-Stub erfüllt folgende Aufgaben:[26]

- Informationen von Anwendungsprogramm empfangen
- Informationen in Nachricht verpacken (Aufrufkodierung)
- Nachricht an Transportkomponente übermitteln
- Antwortet der Transportkomponente auspacken (Ergebnisdekodierung)
- Ergebnis an Anwendungsprogramm übermitteln

Als Informationen werden hier unter anderem die Adresse des Servers und Parameter der Prozedur zusammengefasst. Solange keine Antwort vorliegt, blockiert der Client-Stub. Die Transportkomponente kommuniziert mit dem Server und überträgt den kodierten Aufruf. Des Weiteren wird das Routing, Quittierung und die Fehlerbehandlung in dieser Komponente geregelt.[27]

Die eingehende Nachricht wird von der serverseitigen Transportkomponente an den Server-Stub übergeben. Dieser hat die Aufgaben:[28]

- Nachricht auspacken (Aufrufdekodierung)
- Ermittelt auszuführende Prozedur
- Parameter an Prozedur übergeben und ausführen
- Ergebnis der Prozedur empfangen und verpacken (Ergebniskodierung)
- Nachricht an Transportkomponente übermitteln

Der Server-Stub ist während des ganzen Prozesses in einer Endlosschleife und wartet auf eingehende Nachrichten. Nach dem verpacken und übermitteln der Nachricht kehrt der Server-Stub an den Anfang der Schleife zurück.

---

[25] Vgl. [Bengel04] S.139
[26] Vgl. [Bengel04] S. 139f; Vgl. [Dadam96] S. 292f
[27] Vgl. [Bengel04] S. 140
[28] Vgl. [Bengel04] S. 140; Vgl. [Dadam96] S. 293

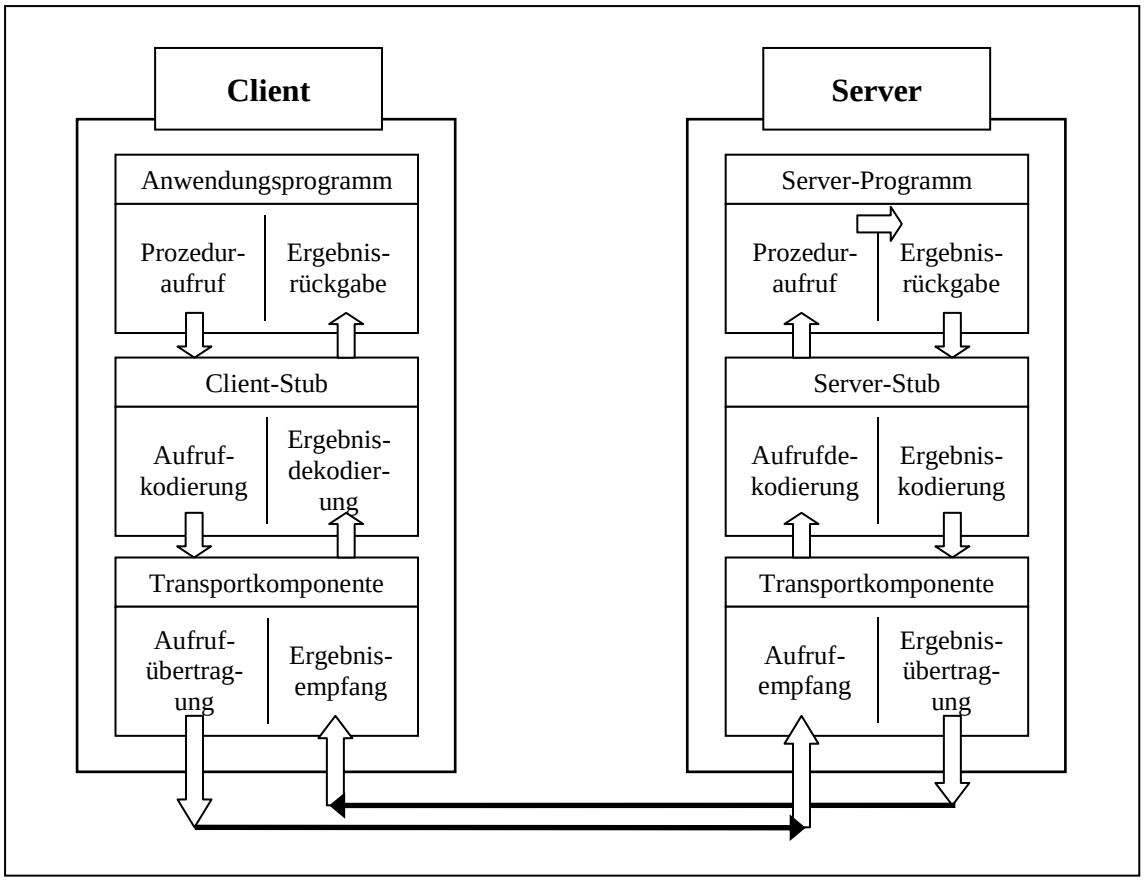

Abbildung 4.1.3: RPC detaillierte Darstellung

Quelle: Vgl. [Bengel04] S. 139

„Obwohl bereits sehr mächtig und elegant, hat der einfache RPC-Mechanismus auch seine Tücken.“[29] Jene „Tücken“ bzw. Probleme können dann entstehen, wenn Fehler beim Übertragen der Nachricht zwischen Client und Server auftreten bzw. die Verbindung ganz unterbrochen wird. Ein weiterer kritischer Punkt ist der Absturz des Servers. Bei beiden Sachverhalten kann es dazu kommt, dass der Client vollständig zum Stillstand kommt, da die Blockierung des Stub und des Anwendungsprogramms nicht aufgehoben wird.[30]

Der hier beschriebene XML-RPC erfüllt somit die Grundanforderungen an einen (Web-)Service, aber ermöglicht es nicht Services zu suchen und liefert keine Beschreibung dieser.

Für den interessierten Leser ist das *U.S. Patent 7028312 B1* des XML-RPC zu empfehlen. Hier werden noch weitere Varianten des RPC ausführlich beschrieben und erklärt.

[29] [Dadam96] S. 293
[30] Vgl. [Dadam96] S. 293

4.2 Simple Object Access Protocol

Simple Object Access Protocol (SOAP) Version 1.2 ist ein auf XML-basierendes Protokoll des World Wide Web Consortium (W3C) für den Austausch von strukturierten Information (Dokumenten-Stil) zwischen verteilten Systemen und darüber hinaus bietet es die Funktionalität eines Methodenaufrufs (RPC-Stil).[31]

SOAP-Nachrichten unterliegen einer bestimmten XML-Struktur, welche sich aus dem Hauptelement *Envelope*, einem optionalen *Header-Element* und dem obligatorisch *Body-Element* zusammensetzt (siehe Abbildung 4.2.1). Obwohl das Header-Element nur optional ist enthält es wichtige Bestandteile. Es handelt sich dabei um eine Erweiterung der SOAP-Nachricht zum Transport zusätzlicher Informationen. Diese Informationen, in sogenannten Headerblöcken (Header Blocks) bereitgestellt, enthalten Anweisungen wie die Nachricht im Body verarbeitet oder weitergeleitet werden sollen.

Das Pflichtelement Body kann aus mehreren Unterelementen (Body sub-elements) bestehen und beinhaltet die eigentlichen Information vom Sender für den endgültigen Empfänger.[32]

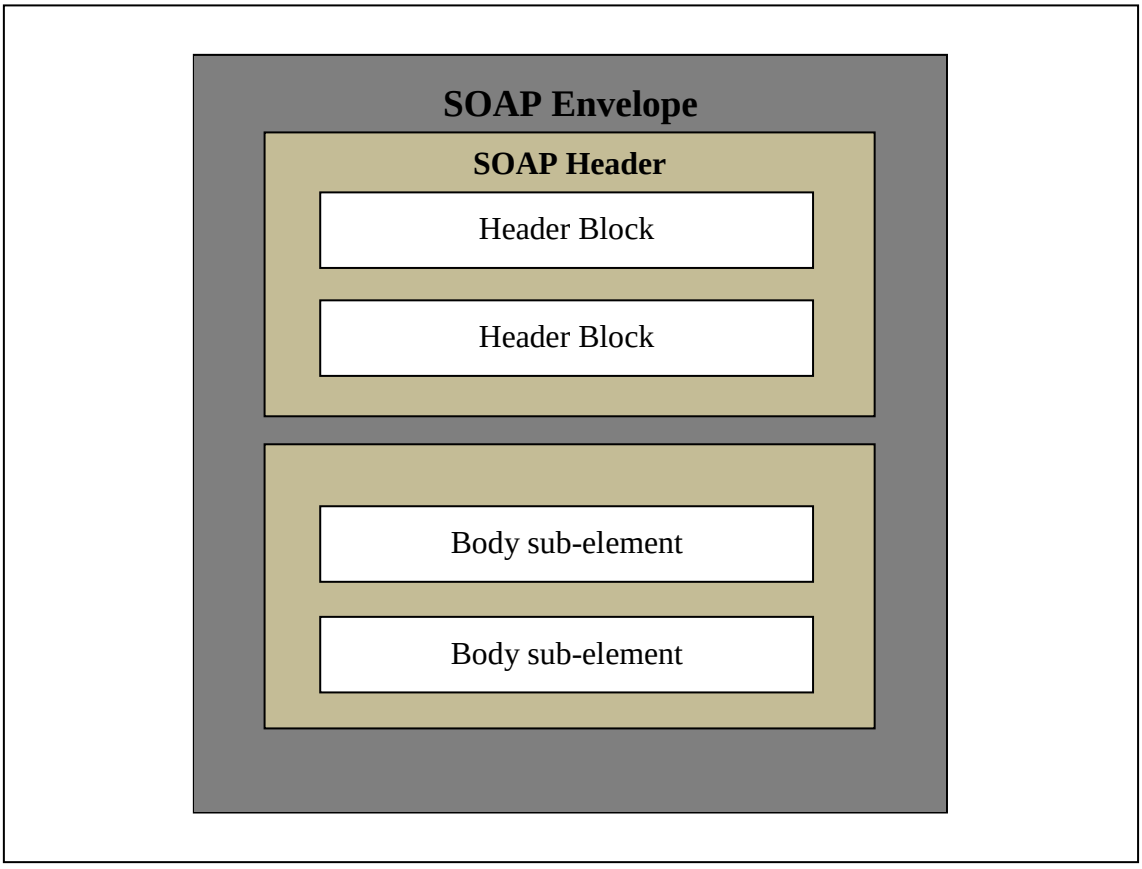

Abbildung 4.2.1: SOAP-Nachrichtenstruktur

Quelle: Vgl. [W3C2011] Abschnitt 2.1

[31] Vgl. [SnTiKu02] S. 13 f
[32] Vgl. [W3C2011] Abschnitt 2.1

Das Beispiel 4.2.1 zeigt eine einfache SOAP-Nachricht im Dokumenten-Stil, die im Header die Transaktionsnummer 65545 und im Body die Bestellinformationen bereitstellt. Der Header enthält darüber hinaus noch ein optionales *mustUnderstand*-Attribut. Dieses soll sicherstellen, dass der Empfänger den Headerblock der Nachricht versteht und diese zu verarbeiten weiß. Sollte dies nicht der Fall sein, wird sie zurückgewiesen. Jenes bedeutet aber nicht, dass die eigentliche Nachricht im Body nicht ordnungsgemäß verstanden und verarbeitet werden kann. In diesem Beispiel soll *transaction* als *mustUnderstand* sichergestellt werden.[33]

Beispiel 4.2.1: SOAP-Nachricht im Dokuementen-Stil – Bestellung

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-
        envelope">
        <soap:Header>
          <m:transaction xmlns:m="soap-transaction"
          soap:mustUnderstand ="true">
               <transactionID>65545</transactionID>
          </m:transaction>
        </soap:Header>
        <soap:Body>
          <n:purchaseOrder xmlns:n="urn:OrderService">
               <from><person>Max Mustermann</person></from>
               <to><person>Maxi Mustermann</person></to>
               <order>
                    <quantity>10</quantity>
                    <item>Blume</item>
               </order>
          </n:purchaseOrder>
        </soap:Body>
</soap:Envelope>
```

Bei einer SOAP-Nachricht handelt es sich in erster Linie um eine "one-way"-Übertragung zwischen einem SOAP-Sender zu einem SOAP-Empfänger. Es ist dennoch möglich Anwendungen so zu gestalten, dass SOAP-Nachrichten komplexe Interaktionen zwischen den einzelnen Anwendungen ermöglichen. Ein einfaches Anwendungsbeispiel, für eine solche Interaktion, ist mit Hilfe einer SOAP-Nachricht einen Remote Procedure Call zu realisiert (siehe Beispiel 4.2.2 und 4.2.3).

[33] Vgl. [SnTiKu02] S. 18 f

Beispiel 4.2.2: SOAP-Nachricht im RPC-Stil – Aktienkurs Anfrage

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-
        envelope">
        <soap:Body>
          <n:getQuote xmlns:n="urn:QuoteService">
               <name xsi:type="xsd:string">
                    Microsoft
               </name>
          </n:getQuote>
        </soap:Body>
</soap:Envelope>
```

Beispiel 4.2.3: SOAP-Nachricht im RPC-Stil – Aktienkurs Antwort

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-
        envelope">
        <soap:Body>
          <n:getQuoteResponse xmlns:n="urn:QuoteService">
               <value xsi:type="xsd:float">
                    29.32
               </value>
          </n:getQuoteResponse>
        </soap:Body>
</soap:Envelope>
```

Die SOAP-Nachricht im RPC-Stil unterliegt bestimmten Regeln für das Verpacken einer RPC-Anfrage. Jeder Ein- und Ausgabeparameter muss als Feld in einer XML-Struktur modelliert werden. Die einzelnen Parameter in dieser Struktur müssen mit der Bezeichnung und der Reihenfolge der auszuführenden Methode übereinstimmen. Die Struktur der Antwort unterliegt keinen zwingenden Regeln. Dennoch hat sich durchgesetzt die Antwort in einer ähnlichen Struktur wie Anfrage zu verpacken. Dazu wird das Antwortstrukturelement der Methodenname mit einem angehängten Response versehen (siehe Beispiel 4.2.2 und 4.2.3).[34]

Des Weiteren unterstützt SOAP eine Versionsverwaltung und eine ausführliche Fehlerbehandlung auf diese hier aber nicht weiter eingegangen werden soll. Für weitere Informationen und Details zu SOAP empfehlen sich die Dokumente des W3C, welche unter http://www.w3.org/standards/techs/soap zu finden sind.

[34] Vgl. [SnTiKu02] S. 28 f

4.3 Web Service Description Language

Web Services Description Language Version 2.0 (WSDL 2.0) unterstützt ein Model und ein XML-Format zur Beschreibung von Web-Services. WSDL 2.0 ermöglicht es die Beschreibung der abstrakten Funktionalität eines angeboten Service von der detaillierten Beschreibung, wie dem „wie“ und „wo“ die Funktionalität angeboten wird, zu trennen.

WSDL 2.0 lässt sich somit in zwei grundlegende Bereiche einteilen: einen abstrakten und einen konkreten Bereich. Innerhalb jedes Teilbereichs verwendet die Beschreibung eine Reihe von Strukturen, um die Wiederverwendbarkeit der Beschreibung zu fördern.

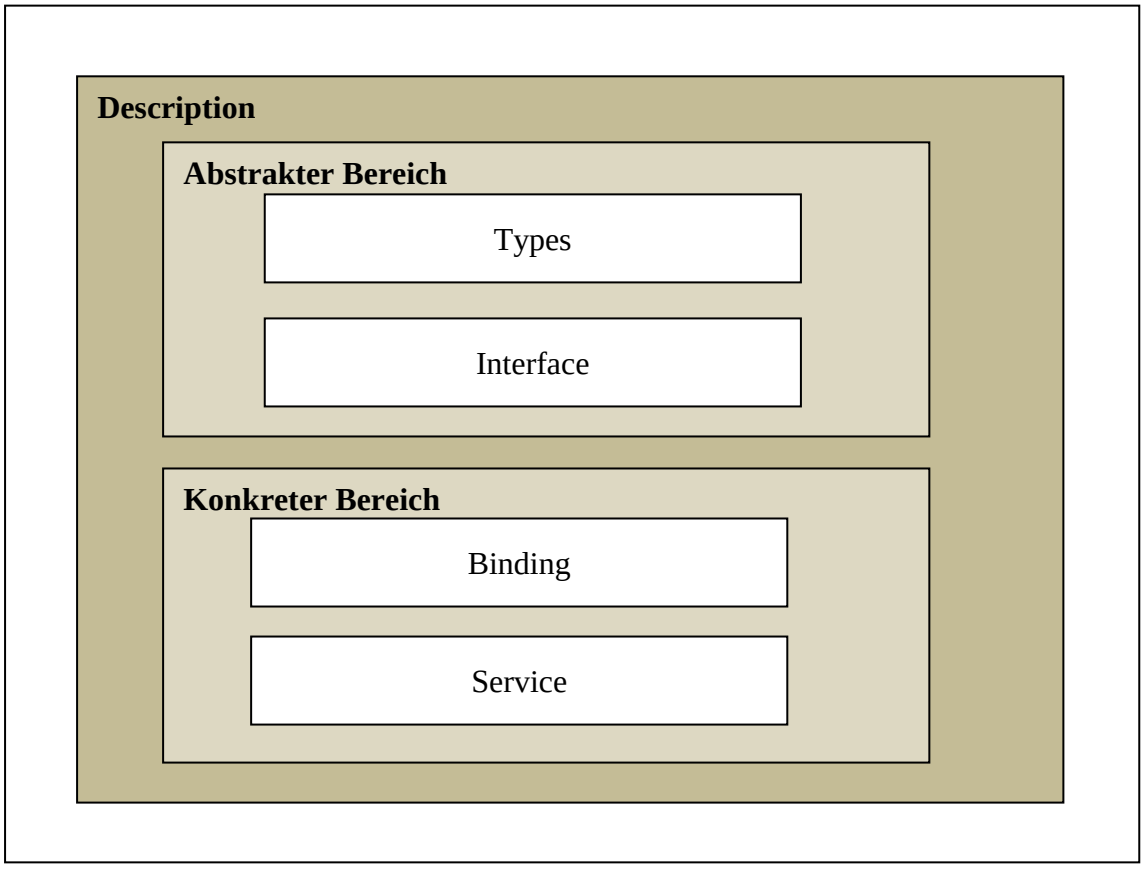

Abbildung 4.3.1: WSDL-Struktur – Abstrakter und konkreter Bereich

Quelle: Vgl. [SnTiKu02] S. 16

In dem abstrakten Bereich (Funktionalität) beschreibt WSDL 2.0 einen Web-Service in Bezug auf das Senden und Empfangen von Nachrichten und dessen Namen. Diese werden unabhängig von einem bestimmten Format und Kommunikationsprotokollen mit Hilfe von einem XML-Schema beschrieben.

Der konkrete Bereich (technische Details) beinhaltet Bindings, welche den Transport- und Kommunikationsdetails für eine oder mehrere Schnittstellen spezifizieren. Ein

Endpunkt verbindet eine Netzwerk-Adresse mit einem *Binding*. Ein *Service* fasst die Endpunkte zusammen, die eine gemeinsame Schnittstelle implementieren.[35]

Somit beinhaltet die WSDL eines Web-Services die Beschreibung von Nachrichten, die zwischen dem Server (Anbieter) und dem Client (Nutzer) ausgetauscht werden. Diese Nachricht wird abstrakt durch eine Ansammlung von Datenelementen beschrieben. Um den Austausch einer Nachricht zu ermöglichen, muss diese an ein Transportprotokoll und an Nachrichtenformat gebunden werden (Binding). Dies wird in WSDL als Operation bezeichnet. Mehrere Operationen werden zu einem *Interface* zusammengefasst. Ein *Interface* wird wiederum an ein Protokoll und an ein Nachrichtenformat gebunden, somit ist das *Interface* über einen Endpunkt zugänglich, wobei jeder Endpunkt eine eigene URI besitzt. Die Endpunkte eines Web-Services, die an dasselbe *Interface* gebunden sind, werden zum Schluss im *Service* zusammengefasst. Wie ein WSDL-Dokument im Detail aufgebaut ist zeigen die folgenden Beschreibungen der einzelnen XML-Elemente. Beispiel 4.3.1 gibt einen Überblick über die WSDL-Struktur.

Beispiel 4.3.1: WSDL-Struktur – Übersicht[36]

```
<description
        targetNamespace="xs:anyURI" >
  <documentation />*
  [ <import /> | <include /> ]*
  <types />?
  [ <interface /> | <binding /> | <service /> ]*
</description>
```

Das Root-Element einer WSDL–Beschreibung ist das *description*-Element. Alle nötigen anderen Elemente sind diesem untergeordnet. Es muss ein *targetNamespace*-Attribut besitzen mit dem der Namensraum dieses Web-Services festgelegt wird, dieser wird an alle anderen Elemente im Dokument vererbt. Des Weiteren ist es möglich zusätzliche Namespace-Attribute hinzuzufügen, die auf die verwendeten Bindungen verweisen.[37]

Das *documentation*-Element (siehe Beispiel 4.3.2) kann optional oder mehrfach auftreten und ist in jedem anderen Element innerhalb der WSDL 2.0 erlaubt. Es enthält Dokumentationen, die für den Menschen lesbare oder verarbeitbar für den Rechner sind. Wie genau die Struktur innerhalb des Elements gestaltet ist, wird nicht näher spezifiziert, muss sich aber an die XML-Schema Strukturen halten. Darüber hinaus darf

[35] Vgl. [W3C2011b] Abschnitt 1
[36] [W3C2011b] Abschnitt 2.1.2
[37] Vgl. [W3C2011b] Abschnitt 1 f

ein xml:lang-Attribut verwendet werden um die Sprache des Inhalts der Dokumentation anzugeben.[38]

Beispiel 4.3.2: WSDL-Struktur – *documentation*-Element[39]

```
<documentation>
  [extension elements]*
</documentation>
```

4.3.1 Import

Das *import*-Element (siehe Beispiel 4.3.2) ermöglicht es WSDL-Komponenten aus fremden Namensräumen zu importieren und zu identifizieren. Wenn ein *import*-Element in einer WSDL-Struktur auftritt, wird damit verdeutlicht, dass das Dokument Verweise auf fremde Komponenten enthalten darf. Somit fungiert es auch als vorgelagerte Deklaration für fremde Namensräume. Somit ist die Verwendung des *import*-Elements eine notwendige Bedingung um fremde Komponenten einem WSDL-Dokument zugänglich zu machen.[40]

Beispiel 4.3.2: WSDL-Struktur – *import*-Element[41]

```
<description>
  <import
        namespace="xs:anyURI" location="xs:anyURI"? >
    <documentation />*
  </import>
</description>
```

Wenn mehrere Elemente des Typs *import* im Dokument vorhanden sein sollten, welche das gleiche *namespace*-Attribut verwenden, so müssen diese unterschiedliche Werte für das *location*-Attribut aufweisen. Die mehrfache Aufführung von *import*-Elementen, mit dem gleichen Namenraum, muss daher alternative Standorte für einen gegeben Namensraum liefern. Ferner ist es, in dieser Spezifikation, nicht zwingend erforderlich diese *location*-Attribute zu dereferenzieren. Sollte es nicht dereferenzierbar sein, so kann es dazu führen, dass keine Informationen, über den zu importierenden Namensraum durch das *import*-Element, zur Verfügung gestellt werden. Dadurch ist es möglich, dass in anderen Teilen des WSDL-Dokuments Fehler auftreten. Diese Fehler

[38] Vgl. [W3C2011b] Abschnitt 5
[39] [W3C2011b] Abschnitt 5
[40] Vgl. [W3C2011b] Abschnitt 4.2
[41] [W3C2011b] Abschnitt 4.2

resultieren durch einen fehlerhaften Verweis. Diese Verweise bzw. Referenzierungen werden *QName* genannt. Was sich dahinter verbirgt wird in Kapitel 3.3 – XML Namespaces erläutert.[42]

4.3.2 Include

Das *include*-Element erlaubt die Aufteilung der unterschiedlichen Komponenten einer Servicedefinition in unabhängige WSDL-Dokumente, welche zum gleichen Ziel-Namensraum gehören. Insbesondere kann es benutzt werden um Komponenten aus einer WSDL Beschreibung hinzuzufügen, welche den gleichen Ziel-Namensraum wie die inkludierende Beschreibung besitzt. Das angegebene *location*-Attribut bezieht sich auf ein *targetNamespace*-Attribut eines *description*-Elements und identifiziert somit das WSDL-Dokument.[43]

Beispiel 4.3.4: WSDL-Struktur – *include*-Element[44]

```
<description>
  <include
        location="xs:anyURI" >
    <documentation />*
  </include>
</description>
```

Ein wechselseitiges Einschließen (engl. *mutual include*) ist das direkte inkludieren eines WSDL-Dokument von einem anderen WSDL-Dokument, welches das erste Dokument bereits inkludiert hat (Beispiel: A inkludiert B, und B inkludiert A). Ein kreisförmiges Einschließen (engl. *circular include*) erzielt den gleichen Effekt nur mit einer größeren Umleitung (Beispiel: A inkludiert B, B inkludiert C, und C inkludiert A). Mehrfaches Einschließen (engl. multiple inclusion) von einem einzelnen WSDL-Dokument löst sich so auf als wäre das Dokument nur einmalig inkludiert worden. Das wechselseitige, kreisförmige und mehrfache Einschließen ist explizit gestattet und stellt keine Neudefinition der gleichen Komponenten dar.[45]

[42] Vgl. [W3C2011b] Abschnitt 4.2
[43] Vgl. [Melzer2010] S. 125
[44] [W3C2011b] Abschnitt 4.1
[45] Vgl. [W3C2011b] Abschnitt 4.1

4.3.3 Types

Das *types*-Element ermöglicht es Datentypen zentral zu definieren. Die Nachricht eines Web-Service kann aus einfachen, aber auch komplexen Datentypen bestehen. Um die Wiederverwendbarkeit von komplexen Datentypen zu fördern, sollte diese auch entsprechend definiert werden, da diese bei mehreren Operationen oder gar Schnittstellenspezifikationen von Bedeutung sein können. Es empfiehlt sich die vom XML-Schema bereitgestellten Strukturen von Elementen und Typen zu nutzen, um eine Datentypdefinition zu erstellen.[46]

Beispiel 4.3.5: WSDL-Struktur – *types*-Element[47]

```
<description>
  <types>
    <documentation />*
    [ <xs:import namespace="xs:anyURI"
        schemaLocation="xs:anyURI"? /> |
      <xs:schema targetNamespace="xs:anyURI"? /> |
      <other extension elements/> ]*
  </types>
</description>
```

4.3.4 Interface

Das *interface*-Element beschreibt Sequenzen von Nachrichten die ein Web-Service sendet bzw. empfängt (siehe Beispiel 4.3.6). Ähnliche Nachrichten werden gruppiert und als *operations* behandelt. Ein *operation*-Element enthält eine Abfolge von Input- und Output-Nachrichten (siehe Beispiel 4.3.7). Es ist möglich, dass ein *interface*-Element aus mehreren *operation*-Elementen bestehen kann.

Über das *extends*-Attribut kann das bestehende *interface*-Element um weitere Schnittstellen (*interfaces*) erweitern werden. Um kreisförmige Definitionen zu vermeiden darf sich das *interface*-Element weder direkt, noch indirekt um sich selbst erweitern. Die Menge der verfügbaren Nachrichten (*operations*) in einem interface-Element hängt von den direkt definierten *operation*-Elementen und den indirekt, über das *extends*-Attribut, definierten Schnittstellen ab.[48]

---

[46] Vgl. [Melzer2010] S. 119
[47] [W3C2011b] Abschnitt 3
[48] Vgl. [W3C2011b] Abschnitt 2.2

Beispiel 4.3.6: WSDL-Struktur – *interface*-Element[49]

```
<description>
  <interface
        name="xs:NCName"
        extends="list of xs:QName"?
        styleDefault="list of xs:anyURI"? >
    <documentation />*
    [ <fault /> | <operation /> ]*
  </interface>
</description>
```

Sollte ein Fehler bei der Verarbeitung eines *interface*-Elements auftreten so wird dies, mit Hilfe des *fault*-Elements, den Beteiligten eine Fehlermeldung übermittelt. Eine Fehlermeldung kann zur Kommunikation abseits des normalen Nachrichtenaustauschs gesendet und empfangen werden. Diese Informationen beschreiben keine allgemeinen Systemfehler wie Netzwerkfehler, Festplattenfehler oder ein ungültiges Nachrichtenformat. Die Fehler beziehen sich mehr auf das Niveau der Applikationssemantik. Zum Beispiel: Eine Interface-Komponente soll eine Kreditkartennummer akzeptieren. Fehler könnten in diesem Beispiel sein, dass die Kreditkartennummer falsch ist, gestohlen gemeldet oder abgelaufen ist.[50]

Beispiel 4.3.7 WSDL-Struktur-Interface – *operation*-Element[51]

```
<description>
  <interface>
    <operation
          name="xs:NCName"
          pattern="xs:anyURI"?
          style="list of xs:anyURI"? >
        <documentation />*
        [ <input /> | <output /> ]*
        [ <infault /> | <outfault/> ]*
    </operation>
  </interface>
</description>
```

[49] [W3C2011b] Abschnitt 2.2.2
[50] Vgl. [W3C2011b] Abschnitt 2.3
[51] [W3C2011b] Abschnitt 2.4.2

4.3.5 Binding

Ein konkretes Nachrichtenformat und Übertragungsprotokoll definiert einen Endpunkt und wird über das *binding*-Element beschrieben. Das *binding*-Element definiert die nötigen Implementationsdetails innerhalb des *operation*-Elements um auf den Service zuzugreifen. Fehler werden mit Hilfe des *fault*-Elements näher beschreiben. Zudem können Informationen für jedes oder ein spezielles Interface und dessen Operationen beschrieben werden um diese wiederzuverwenden. Zwingend erforderlich sind ein Name und die Angabe des Typs für die Komponente (siehe Beispiel 4.3.8).[52]

Beispiel 4.3.8: WSDL-Struktur – *binding*-Element[53]

```
<description>
  <binding
        name="xs:NCName"
        interface="xs:QName"?
        type="xs:anyURI" >
    [<documentation /> | <fault /> | <operation /> ]*
  </binding>
</description>
```

Die Komponente *operation* im *binding*-Element beschreibt das konkrete Nachrichtenformat und Austauschprotokolle in Verbindung mit einer bestimmten Interface Operation des gegebenen Endpunktes.[54]

Beispiel 4.3.9: WSDL-Struktur-Binding – *operation*-Element[55]

```
<description>
  <binding>
    <operation
          ref="xs:QName" >
        <documentation />*
        [ <input /> | <output /> ]*
        [ <infault /> | <outfault /> ]*
    </operation>
  </binding>
</description>
```

[52] Vgl. [W3C2011b] Abschnitt 2.7
[53] [W3C2011b] Abschnitt 2.7.2
[54] Vgl. [W3C2011b] Abschnitt 2.9.1
[55] [W3C2011b] Abschnitt 2.9.2

4.3.6 Service

Das *service*-Element beschreibt eine Menge von Endpunkten (*endpoints*), an dem eine bestimmte Implementation ausgeführt und der Service bereitgestellt wird. Die Endpunkte bilden somit unterschiedliche Orte, an denen ein Service angeboten wird.[56]

Beispiel 4.3.10: WSDL-Struktur – *service*-Element[57]

```
<description>
  <service
        name="xs:NCName"
        interface="xs:QName" >
    <documentation />*
    <endpoint />+
  </service>
</description>
```

Die Attribute *name* und *interface* sind zwingend erforderlich (siehe Beispiel 4.3.10). Das *name*-Attribut stellt den Namen der Komponente dar. Über das zwingend erforderliche *interface*-Attribut wird die Schnittstelle angegeben, die der Service instanziiert. Darüber hinaus beinhaltet der Service mindestens ein *endpoint*-Element und keine oder mehere Dokumentationen.[58]

Ein *endpoint*-Element definiert die Angaben eines bestimmten Endpunktes. Des Weiteren stellt ein Endpunkt eine konkrete Software-Komponente auf einem System dar, welche den angeforderten Dienst unter einer bestimmten Netzwerkadresse zur Verfügung stellt.

Das Angeben von einem *name*-Attribut und einem *binding*-Attribut, welche sich innerhalb des *endpoint*-Elements befinden, sind zwingend erforderlich. Die Angabe einer Netzwerkadresse, mit Hilfe des address-Attributs, kann optional geschehen muss aber eine absolute URI verwenden, die die Netzwerkadresse des Service unter Einbezug des Bindings darstellt. Dokumentationen können auch hier mehrfach oder gar nicht angegeben werden (siehe Beispiel 4.3.11).[59]

[56] Vgl. [W3C2011b] Abschnitt 2.12
[57] [W3C2011b] Abschnitt 2.12.2
[58] Vgl. [W3C2011b] Abschnitt 2.12
[59] Vgl. [W3C2011b] Abschnitt 2.13

Beispiel 4.3.11: WSDL-Struktur-Service – *endpoint*-Element[60]

```
<description>
  <service>
    <endpoint
          name="xs:NCName"
          binding="xs:QName"
          address="xs:anyURI"? >
      <documentation />*
    </endpoint>+
  </service>
</description>
```

Abbildung 4.3.2 zeigt nochmals alle Komponenten bzw. Elemente mit den einzelnen Attributen im Überblick.

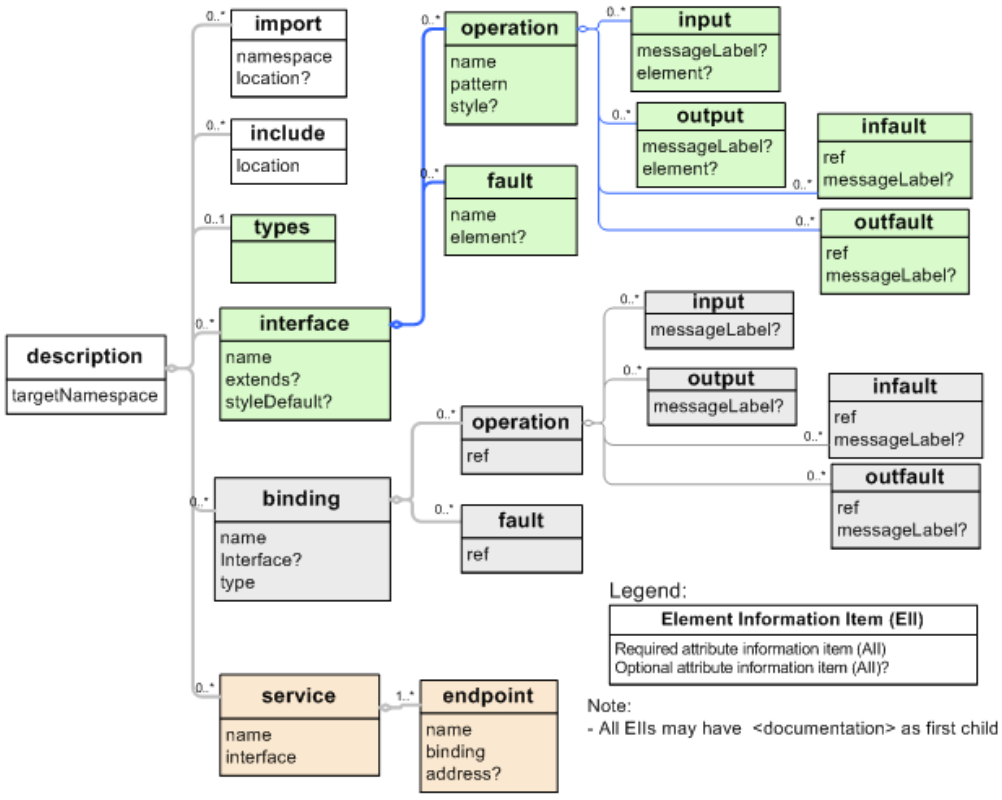

Abbildung 4.3.2 WSDL-Struktur – Übersicht

Quelle: [W3C2011c] Abschnitt 2.2

[60] [W3C2011b] Abschnitt

Für den interessierten Leser gibt es ein ausführliches Beispiel eines WSDL-Dokuments vom W3C. Zu finden ist dies unter http://www.w3.org/TR/wsdl20-primer/#basic-example. Es wird dort nochmals auf alle Bestandteile eines solchen Dokuments eingegangen und anhand eines Hotelreservierungs-Service beschrieben.

4.4 Universal Description, Discovery and Integration

Das Universal Description, Discovery and Integration (UDDI) Konzept wurde von IBM, Ariba und Microsoft entwickelt und soll es ermöglichen Web-Services zu veröffentlichen, zu finden und zu nutzen. Um dies zu ermöglichen veröffentlicht ein Anbieter eines Web-Services die WSDL-Beschreibung seines Services in ein zentrales UDDI Registry (UDDI-Verzeichnis). Der Nutzer muss nun einen Service nach technischen Gesichtspunkten auswählen, um sicherzustellen, dass dieser auch verwendet werden kann. Darüber hinaus können nicht nur die technischen Details eine Rolle für den Nutzer spielen. Wichtig könnten auch die Reputation oder die geographische Lage des Unternehmens sein. Das UDDI-Verzeichnis bietet dem Nutzer folgende Informationen an:[61]

- Unternehmensinformationen – *White Pages*
- Brancheninformationen – *Yellow Pages*
- Beschreibungen von Web-Services – *Green Pages*
- Schnittstellen der Web-Services – *Service Type Registration*

4.4.1 UDDI Pages

White Pages

Die *White Pages* lassen sich mit einem normalen „Telefonbuch“ vergleich. Unternehmen welche einen Web-Service in einem UDDI-Verzeichnis bereitstellen, stellen somit auch gleichzeitig Informationen über sich selbst zur Verfügung. Dadurch wird dem Nutzer ermöglicht, ein Unternehmen genau zu identifizieren und er kann sich entscheiden, ob er den Service des Unternehmens nutzen möchte oder nicht. Ein Beispiel für *White Pages* bitte die Webseite www.whitepages.com.

Yellow Pages

Die *Yellow Pages* sind vergleichbar mit den bekannten „Gelben Seiten“. Der Name von einem Unternehmen, welche eventuell den gesucht Service anbietet, ist unter Umständen nicht bekannt. Der Nutzer weiß aber, in welche Kategorie bzw. Branche der gesuchte Service fällt. Mit Hilfe der *Yellow Pages* kann so ein möglicher Anbieter für einen gewünschten Service ermittelt werden. Ein Beispiel für *Yellow Pages* bitte die Webseite www.gelbeseiten.de.

[61] Vgl. [Zimmermann/Tomlinson/Peuser] S. 131

Green Pages

Die *Green Pages* bilden ein weiteres Konzept zur Service suche. Es ist mit Hilfe der Green Pages möglich für jeden Service eine Beschreibung zu hinterlegen. Weiß ein Nutzer also was für einen Service er benötigt, aber ihm kein Anbieter oder die Branche gänzlich unbekannt ist, so kann durch die Nutzung von *Green Pages* die Liste der Services durchsucht werden.

Service Type Registration

Die *Service Type Registration* vervollständigt das Konzept. Da die Green Pages in erster Linie für die Verwendung durch menschliche Nutzer gedacht sind, wurden so ist die *Service Type Registration* speziell in einer maschinenlesbaren Form. Green Pages und die *Service Type Registration* verweisen aufeinander.

Menschen und Anwendungen sind die Nutzer eines UDDI-Verzeichnis, welches aus den zuvor beschriebenen *Pages* und der *Service Type Registration* hervorgeht. Der Mensch seinerseits, versucht Informationen über einen Web-Service oder über ein Unternehmen, welches Web-Services anbietet, zu finden. Wohingegen eine Anwendung eine bestimmte erforderliche Funktionalität aus einem UDDI-Verzeichnis sucht. Um den Service nutzen zu können, werden die benötigten technischen Informationen zur Verwendung des Web-Services, mit Hilfe eines WSDL-Verweises auf ein bestimmtes Dokument bereitgestellt.

Im Idealfall könnte es möglich sein, das eine Anwendung, falls es zum Absturz, Verbindungsfehler oder sonstigen auftretenden Problemen kommt, mit Hilfe des UDDI-Verzeichnisses einen oder mehrere Web-Services sucht und findet, die den gleichen Funktionsumfang bieten wie der zuvor gewählte Web-Service.

Zur Realisierung des menschlichen und maschinellen Zugriffs auf das UDDI-Verzeichnis werden im Normalfall zwei Schnittstellen für diese Zugriffe angeboten. Ein webbasierter Zugang ist für den menschlichen Nutzer gedacht. Die nötigen Informationen über die Schnittstellen des Web-Services für eine Anwendung werden über die WSDL-Datei bereitgestellt, welche durch die Anwendung über einen speziellen Zugang gesucht werden können.

Eine Fülle von standardisierten Web-Services zur automatisierten Suche für Anwendung wird daher durch ein UDDI-Verzeichnis angeboten. Auch wenn die Datenstruktur von UDDI standardisiert ist, kann sich die Darstellung je nach

verwendetem Tool unterscheiden. Zudem ist die Struktur der UDDI-Inhalte als XML-Dokument definiert und damit flexibel erweiterbar.

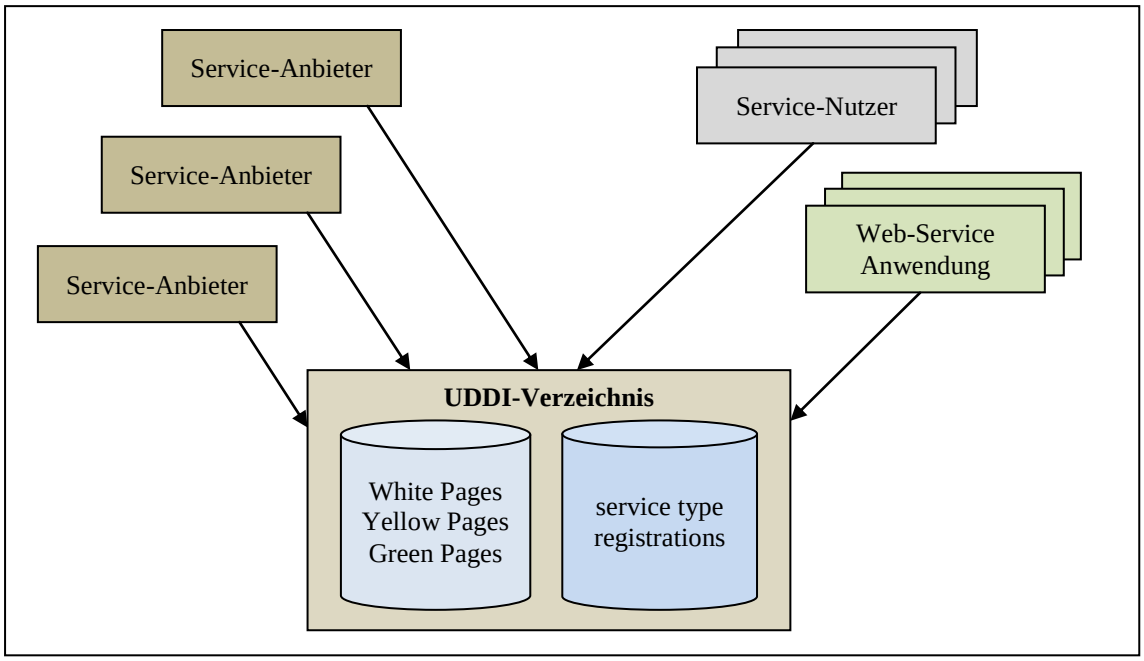

Abbildung 4.4.1: UDDI-Übersicht

Quelle: Vgl. [Melzer2010] S. 143

4.4.2 UDDI technischer Überblick

Der UDDI-Standard ist seit 2005 in der Version 3 verfügbar. Der technische Überlick über UDDI wird sich daher auf die Version 3 beziehen. UDDI steht wie bereits eingangs beschrieben für Universial Description, Discovery and Integration. Die drei Kernbegriffe (Beschreibung, Entdeckung und Integration) beschreiben den Funktionsumfang von UDDI ziemlich genau. Da alles was sich in einem XML-Dokumente darstellen lässt, auch in einem UDDI-Verzeichnis veröffentlichen und suchen lässt, wird er Begriff der Universalität auch erfüllt. UDDI selbst besteht aus einem UDDI XML Schema und der UDDI-API (UDDI Application Programming Interface):[62]

- Das Datenmodell von UDDI wird durch UDDI XML Schema definiert und stellt folgende Datenstrukturen bereit: *businessEntity*, *businessService*, *bindingTemplate*, *tModel*, *operationalInfo* und *publisherAssertion*.

- Die UDDI-API verwendet selbst Web-Services und nutzt das SOAP-Protokoll des W3C. Die Schnittstellen ermöglichen die Suche, Veröffentlichung oder Verwaltung von Daten im Verzeichnis. Veröffentlicht oder sucht ein Nutzer Information, kommt es zum Austausch von SOAP-Nachrichten mit dem Server.

[62] Vgl. [Melzer2010] S. 147f

Diese zwei Komponenten von UDDI werden in den folgenden Abschnitten im Detail beschrieben.

4.4.2.1 UDDI XML Schema

Das Suchen, Verwalten, Katalogisierung und Kategorisierung von Web-Services in einem UDDI-Verzeichnis wird mit Hilfe eines speziellen Datenmodells realisiert. Dieses Datenmodell bietet, neben der technischen Beschreibung eines Web-Service, auch die Möglichkeit Informationen über den Anbieter der Web-Services zu erfassen. Neben dieser Funktionalität spielt die Kategorisierung von Anbietern und Diensten eine wichtige Rolle. Insbesondere dann wenn in größeren Verzeichnissen gesucht werden soll. Oder aber der Nutzer kennt den Anbieter beziehungsweise den konkreten Namen eines Dienstes nicht oder möchte nach einer bestimmten Kategorie von Diensten suchen.[63]

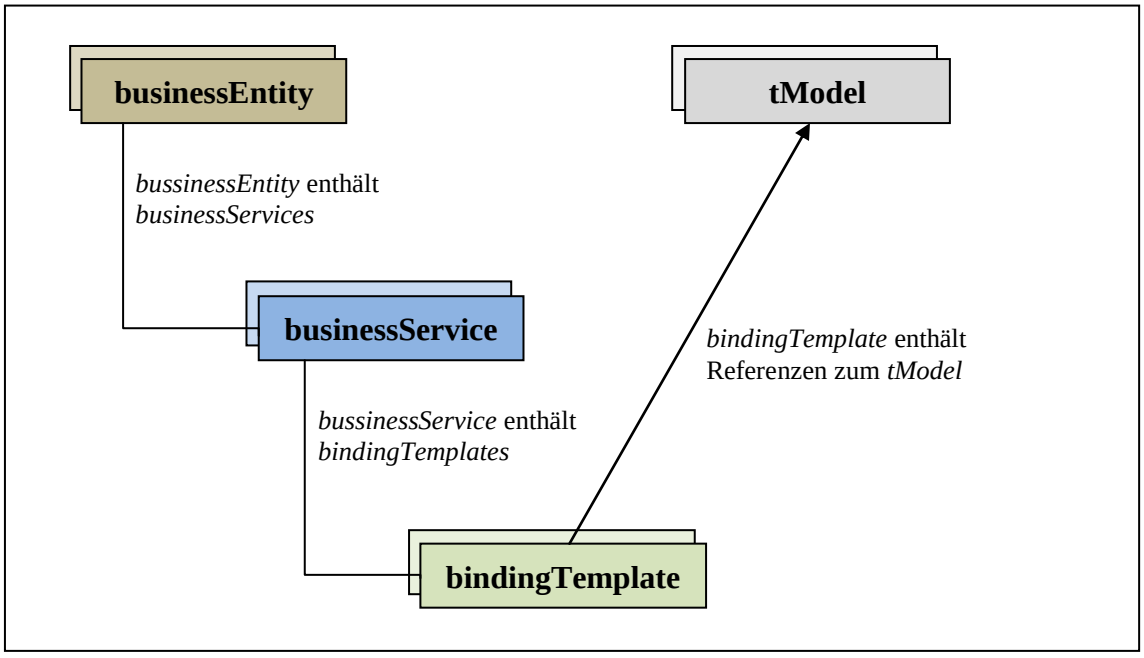

Abbildung 4.4.2: UDDI Datenstruktur

Quelle: Vgl. [OASISUDDI] Abschnitt 3

Die Abbildung 4.4.2 zeigt die Datenstruktur von UDDI, die vier Kerndatentypen businessEntity, businessService, bindingTemplate und das tModel stehen miteinander in Beziehung.[64]

- Der *businessEntity*-Datentyp stellt Information von Gruppen bereit, welche Informationen über einen Service veröffentlichen. Bei den Gruppen handelt es

[63] Vgl. [Melzer2010] S. 148
[64] Vgl. [OASISUDDI] Abschnitt 3.2

sich zum Beispiel um Softwareunternehmen, Organisationen oder auch einzelne Personen.[65]

- Der *businessService*-Datentyp stellt beschreibende Informationen über eine bestimmte Branche von technischen Dienstleistungen zur Verfügung.[66]
- Der *bindingTemplate*-Datentyp stellt technische Informationen über eine Serviceeintrag und Implementationspezifikationen bereit.
- Der tModel-Datentyp liefert die Beschreibungen der Spezifikationen für Dienstleistungen oder Wertemengen. Des Weiteren bildet es die Basis für den technischen Fingerabdruck.

Im Folgenden werden nun die einzelnen Datentypen näher beleuchtet.

businessEntity

Wie zuvor beschrieben, enthält jede *businessEntity* die Beschreibung über ein Unternehmen oder eine Organisation. Durch den zugehörigen businessService können Informationen über die Dienstleistungen dieser Unternehmen abgerufen werden. Aus XML-Sicht bildet *businessEntity* das Wurzelelemt in der Datenstruktur, welches alle Informationen beinhaltet.[67]

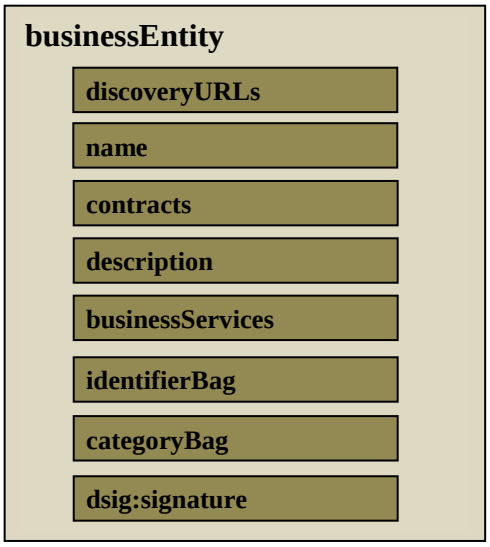

Abbildung 4.4.3: UDDI businessEntity-Datenstruktur

Quelle: Vgl. [OASISUDDI] Abschnitt 3.3.1

[65] Vgl. [IBMUDDI]; Vgl. [OASISUDDI] Abschnitt 3.1
[66] Vgl. [OASISUDDI] Abschnitt 3.1
[67] Vgl. [OASISUDDI] Abschnitt 3.3f

Abbildung 4.4.3 zeigt die *businessEntity*-Datenstruktur und die darin enthaltenen Elemente, welche im Folgenden beschrieben werden.[68] Das Beispiel 4.4.1 veranschaulicht nochmal die Datenstruktur.

- Das *businessKey*-Attribut (in Abbildung 4.4.3 nicht enthaltene) dient der eindeutigen Identifizierung einer *businessEntity*. Der Wert ist im gesamten UDDI-Verzeichnis einzigartig. Wenn eine *businessEntity* in einem UDDI-Verzeichnis veröffentlicht wird, muss der *businessKey* anfangs entfallen um einen Schlüssel durch das Verzeichnis generieren zu lassen.

- Das optionale *discoveryURLs*-Element ist eine Liste von URLs die auf Alternativen verweisen.

- Das *name*-Element definiert den Namen und kann mehr als einmal in der *businessEntity* vorkommen. Um zum Beispiel gesetzliche Namen oder Abkürzungen zu nutzen, ist die Möglichkeit mehrere Namen anzugeben, sinnvoll. Das *name*-Element hat ein optionales Attribut *xml:lang* um die verwendetet Sprachen festzulegen.

- Das *description*-Element kann mehrere Beschreibungen der *businessEntity* beinhaltet. Das *description*-Element besitzt ein optionales Attribut *xml:lang* um die Sprache der Beschreibung festzulegen.

- Das *Contacts*-Element beinhaltet eine Kontaktdatenstruktur für Informationen zu einer oder mehreren Personen. Dies dient dem Zweck der Kontaktaufnahme. Diese Informationen bestehen aus einer Beschreibung, Personennamen, Telefonnummer, eMail und Addressinformationen.

- Das *businessServices*-Element wird verwendet um eine Liste von *businessService*-Datenstrukturen anzubieten. Weitere Details zum *businessService* werden im nächsten Abschnitt beschrieben.

- Das *identifierBag*-Element ist optional und enthält eine Liste von einem oder mehreren *keyedReference*-Elementen. Eine keyedReference repräsentiert einen Kennung oder ein spezielles Kennungssystem (zum Beispiel eine Steuernummer).

- Das *categoryBag*-Element erlaubt es *businessEntity* nach dem veröffentlichten Kategorisierungssystem eingeordnet zu werden. Die Zuordnung des Service kann zum Beispiel nach einem geografischen Gebiet oder einer Branche erfolgen.

[68] Vgl. [OASISUDDI] Abschnitt 3.3.1ff

- Das *dsig:Signature*-Element ermöglicht es dem *businessEntity*-Element eine digitale Signatur zu geben.

Beispiel 4.4.1: businessEntity XML-Beispiel[69]

```
<businessEntity businessKey="1B51FFEA-9101-43D0-BAB9-...">
        <discoveryURLs>
          <discoveryURL useType="businessEntity">
          http://uddi.ibm.com/ubr/uddiget?businessKey=...
          </discoveryURL>
        </discoveryURLs>
        <name>IBM UDDI Business Registry Node</name>
        <description>
          This provider represents the IBM node of the
          UDDI Business Registry
        </description>
        <contacts>
          <contact useType="UDDI Support">
               <personName>UDDI Support</personName>
               <email
               useType="help">uddisupt@us.ibm.com</email>
          </contact>
        </contacts>
        <businessServices>...</businessServices>
        <categoryBag>
        <keyedReference
          tModelKey="UUID:327A56F0-3299-4461-BC23-
          5CD513E95C55"
          keyName="IBM UDDI Node" keyValue="node"/>
        </categoryBag>
</businessEntity>
```

businessService

Die *businessService*-Struktur repräsentiert einen Dienst in einem logischen und betriebswirtschaftlichen Sinn und beinhaltet beschreibende Information. Das *businessService*-Element ist das Kindelement von einer *businessEntity.*[70]

[69] [Melzer2010] S. 151

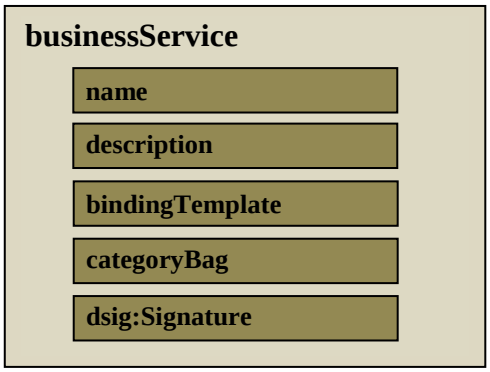

Abbildung 4.4.4: UDDI businessService-Datenstruktur

Quelle: Vgl. [OASISUDDI] Abschnitt 3.4.1

Die technischen Informationen über eine *businessService*-Entität können innerhalb des *bindingTemplate*-Elements gefunden werden. Unternehmen würden in einigen Fällen gern Dienste mit anderen teilen oder diese wiederverwenden. Dies kann dadurch erreicht werden, wenn eine *businessService*-Struktur auf einen veröffentlichten *businessService* projiziert wird.[71]

Eine *businessService*-Datenstruktur wird durch das optionale *serviceKey*-Attribut eindeutig identifiziert. Es verhält sich mit *serviceKey*-Attribut genauso wie mit dem zuvor beschriebenen *businessKey*-Attribut des *businessEntity*-Elements. Darüber hinaus existiert noch ein weiteres optionales einzigartiges *businessKey*-Attribut, welches das *businessEntity*-Element identifiziert das vom *businessService* angeboten wird.[72]

Beispiel 4.4.2: businessService XML-Beispiel[73]

```
<businessService serviceKey="14949B50-4507-11D7-BC51-...">
        <name>Publisch to IBM UDDI ...</name>
        <description>Publish to IBM UDDI ...</description>
        <bindingTemplates/>
          <bindingTemplate bindingKey="14B9...">
               ...
          </bindingTemplate>
        </bindingTemplates>
</businessService>
```

Abbildung 4.4.4 zeigt die *businessService*-Datenstruktur die untergeordneten Elemente. Die Definition der Elemente *name* und *description* sind mit den gleichnamigen

[70] Vgl. [OASISUDDI] Abschnitt 3.4
[71] Vgl. [OASISUDDI] Abschnitt 3.4
[72] Vgl. [OASISUDDI] Abschnitt 3.4.2
[73] Vgl. [Melzer2010] S. 152

Elementen der *businessEntity*-Datenstruktur identisch. Das *bindingTemplate*-Element wird im nächsten Abschnitt näher beschrieben. Die Elemente *categoryBag* und *disg:Signature* sind genauso definiert, wie die gleichnamigen Elemente der *businessEntity*. Das Beispiel 4.4.2 veranschaulicht nochmals die businessService-Datenstruktur.

bindingTemplate

Die *bindingTemplate*-Datenstruktur stellt die notwendigen technischen Details für einen Web-Service zur Verfügung. Jedes *bindingTemplate* beschreibt eine Web-Services und bietet eine bestimmte Netzwerkadresse an, in der Regel in Form einer URL. Des Weiteren wird beschrieben wie ein Web-Service unter der Verwendung von Referenzen zu einem tModel, anwendungsspezifischen Parametern und Einstellungen angeboten wird. Jedes *bindingTemplate* ist in einem *businessService* enthalten.[74]

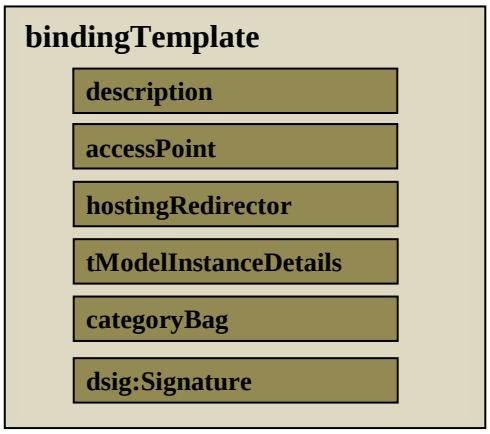

Abbildung 4.4.5: UDDI bindingTemplate-Datenstruktur

Quelle: Vgl. [OASISUDDI] Abschnitt 3.5.1

Abbildung 4.4.5. zeigt die bindingTemplate-Datenstruktur und die untergeordneten Elemente. Darüber hinaus besitzt das *bindingTemplate* ein optionales *bindingKey*-Attribut und *serviceKey*-Attribut. Das *bindingKey*-Attribut ist einzigartig und erlaubt eine eindeutige Identifizierung des *bindingTemplates*. Die Elemente *description*, *categoryBag* und *dsig:Signature* besitzen die gleichen Eigenschaften, wie schon zuvor beschrieben, aber in Bezug auf das *bindingTemplate*. Im Folgendenn werden die noch ungenannten Elemente der bindingTemplate-Datenstruktur beschrieben.[75]

- Das *accessPoint*-Element ist eine Zeichenkette, welche eine Netzwerkadresse darstellt, um einen Web-Service aufzurufen. Normalerweise handelt es sich dabei um

[74] Vgl. [OASISUDDI] Abschnitt 3.5
[75] Vgl. [OASISUDDI] Abschnitt 3.5.2

eine URL, kann aber auch eMail-Adresse oder gar eine Telefonnummer sein. Das optionale *useType*-Attribut kann zum Beispiel folgende Werte annehmen, auf die aber an dieser Stelle nur genannt werden sollen. Der interessierten Leser kann sich unter [OASISUDDI] Abschnitt 3.5.2.1 informieren.

  - endPoint
  - bindingTemplate
  - hostingRedirctor
  - wsdlDeployment

- Das *hostingRedirector*-Element findet keine Anwendung ab der UDDI Version 3, da das *accessPoint*-Element diese Funktionalität übernommen hat. Für die Abwärtskompatibilität kann es noch verwendet werden, aber dies wird nicht empfohlen.

- Das *tModelInstanceDetails*-Element ist eine Liste mit einem oder mehreren *tModelInstanceInfo*-Element und sollte bei der Veröffentlichung eines *bindingTemplates* enthalten sein. Das *tModelInstanceInfo*-Element beinhaltet einen oder mehrere *tModel*-Referenzen. Diese willkürlich geordnete Sammlung von Referenzen nennt man den „technischen Fingerabdruck“ eines Web-Service. Web-Services können so über die tModelKeys identifiziert werden. Mittels Untersuchung lassen sich so bindingTemplates mit einem bestimmten Fingerabdruck erkennen.

Beispiel 4.4.3: bindingTemplate XML-Beispiel[76]

```
<bindingTemplate bindingKey="14B7B3B0-4507-11D7-BC51-...">
        <description>
          Publish to the IBM UDDI BR
        </description>
        <accessPoint URLType="https">
          https://uddi.ibm.com/ubr/publishapi
        </accessPoint>
        <tModelInstanceDetails>
          <tModelInstanceInfo tModelKey="UUID:A2F36B65-
          2D66-408..."/>
        </tModelInstanceDetails>
</bindingTemplate>
```

[76] [Melzer2010] S. 154

tModel

Die tModel-Datenstruktur soll es einfacher machen, Web-Services in einer aussagekräftigen Art zu beschreiben, um einen Nutzen während der Suche zu haben. Dies ist ein wichtiges Ziel von UDDI. Ein weiteres ist es, diese Beschreibungen so zu gestalten, dass Menschen und Anwendungen auf einfache Art und Weise herausfinden wie unbekannte Web-Services funktionieren und mit diesen zu interagieren ist. Um dies zu erreichen müssen Spezifikationen, Konzepte und Designs eingehalten werden, die *tModel*-Struktur erfüllt diese Rolle.

„Es stellt also eine Struktur dar, die nicht nur Wiederverwendung und dadurch eine Standardisierung in einem Software-System erlaubt, sondern darüber hinaus die Möglichkeit schafft, Web Services auf technischer Ebene miteinander zu vergleichen und automatisiert zu prüfen, wie ein Web Service angesprochen werden muss und welche Spezifikationen und Konstrukte er verwendet.“[77]

Ein gegebenes *tModel* ist einzigartig und mittel dessen *tModelKey*-Attribut zu identifizieren. Das *deleted*-Attribut gibt an, ob ein *tModel* logisch gelöscht ist. Die beiden zulässigen Werte sind „*true*“ und „*false*“.

Abbildung 4.4.6 zeigt die *tModel*-Datenstruktur. Im Gegensatz zu den anderen Datenstrukturen muss das *name*-Element beim *tModel* genau einmal vorhanden sein und darf nicht leer sein. Darüber hinaus sollte der Name als URI formatiert werden und das Attribut *xml:lang* sollte nicht verwendet werden. Ein tModel kann null oder mehrere *description*-Elemente enthalten.

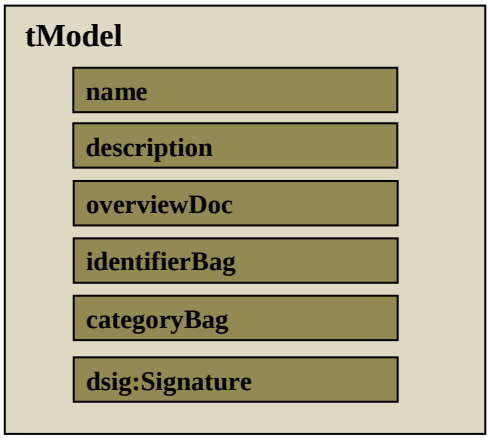

Abbildung 4.4.6: UDDI tModel-Datenstruktur

Quelle: Vgl. [OASISUDDI] Abschnitt 3.6.2

[77] [Melzer2010] S. 154

Das *overviewDoc*-Element ist ein optionales sich wiederholendes Element, zur Unterbringung von Verweisen auf extern vorliegende Dokumente die sich auf das tModel beziehen.

Beispiel 4.4.4: tModel XML Beispiel[78]

```
<tModel tModelKey="UUID:A2F36B65-2D66-4088-ABC7">
        <name>uddi-org:publication_v2</name>
        <description>
          UDDI Publication API Version 2 - Core
          Specification
        </description>
        <overviewDoc>
          <description>
               This tModel defines the publication V2
               API calls for interacting with a V2 UDDI
               node.
          </description>
          <overviewURL>
               http://www.uddi.org/wsdl/publish_v2.wsdl
          </overviewURL>
        </overviewDoc>
        <categoryBag>
          <keyedReference tModelKey="UUID:C1ACF26D-9672-
          4404" keyName="types" keyValue="specification"/>
          <keyedReference tModelKey="UUID:C1ACF26D-9672-
          4404" keyName="types" keyValue="xmlSpec"/>
          <keyedReference tModelKey="UUID:C1ACF26D-9672-
          4404" keyName="types" keyValue="soapSpec"/>
          <keyedReference tModelKey="UUID:C1ACF26D-9672-
          4404" keyName="types" keyValue="wsdlSpec"/>
        </categoryBag>
</tModel>
```

Publisher Assertion

Viele Unternehmen und Organisationen können nicht effektiv durch eine einzige *businessEntity* beschrieben werden, sondern benötigen die Repräsentation durch

[78] [Melzer2010] S. 156

mehrere *businessEntity*. Dies kann geschehen wenn zum Beispiel ein Unternehmen eine Vielzahl von Tochtergesellschaften aufweist. Um dieses Problem zu lösen, liegt es nahe einfach mehrere *businessEntity* zu veröffentlichen. Die Verbindungen zwischen den einzelnen *businessEntity* sind in diesem Zusammenhang für Außenstehende meist nicht zu erkennen, da keine sichtbaren Beziehungen zwischen den einzelnen Entitäten im UDDI-Verzeichnis hinterlegt sind. Eine Lösung bietet die *publisherAssertion*-Struktur an, um somit die Verbindung zwischen zwei Parteien sichtbar zu machen. Um zu vermeiden, dass ein Anbieter eine ungewollte Beziehung zu einem anderen Anbieter eingeht, müssen beide dieser Beziehung zustimmen, ansonsten wird diese Beziehung nicht sichtbar.[79]

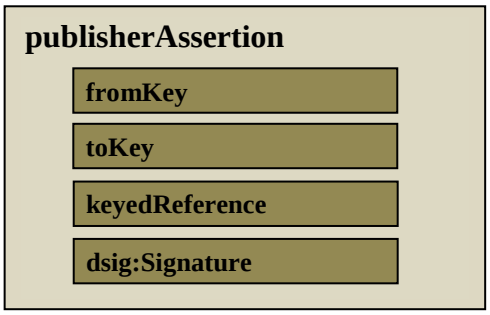

Abbildung 4.4.7: UDDI publisherAssertion-Datenstruktur

Quelle: Vgl. [OASISUDDI] Abschnitt 3.7.1

Die Abbildung 4.4.7 zeigt die *publisherAssertion*-Datenstruktur. Die beiden *businessEntity* zwischen denen eine Beziehung sichtbar gemacht werden soll, sind über die einzigartigen benötigten Elemente *fromKey* und *toKey* zu identifizieren. Das *keyedReference*-Element beschreibt die Beziehung zwischen den *businessEntity*-Elementen, welche über *fromKey* und *toKey* identifiziert werden.[80]

operationalInfo

Wird eine UDDI Kerndatenstruktur veröffentlicht, dann werden Informationen über diese Operation aufgezeichnet. Diese Informationen umfassen das Datum und die Zeit des Anlegens, wann und an welchem UDDI-Knoten Änderungen vorgenommen wurde und die Identität des Herausgebers.[81]

Die Abbildung 4.4.8 zeigt die *operationalInfo*-Datenstruktur. Das *operationalInfo*-Element besitzt ein benötigtes einzigartiges *entityKey*-Attribut. Das *created*-Element dokumentiert, wann eine Entität das erst mal in einem UDDI-Verzeichnis auftaucht.

[79] Vgl. [OASISUDDI] Abschnitt 3.7
[80] Vgl. [OASISUDDI] Abschnitt 3.7.2
[81] Vgl. [OASISUDDI] Abschnitt 3.8

Das *modified*-Element dokumentiert, wann genau eine Entität das letzte Mal geändert wurde. Das *nodeID*-Element identifiziert einen Knoten innerhalb eines UDDI-Verzeichnisses.[82]

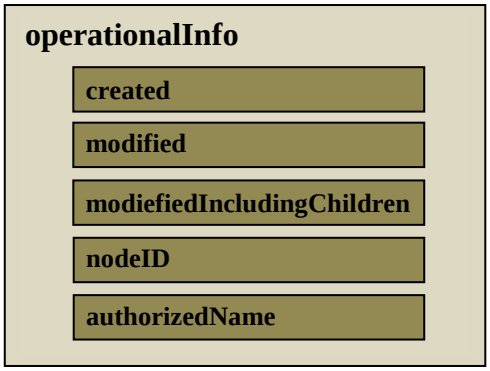

Abbildung 4.4.8: UDDI operationalInfo-Datenstruktur

Quelle: Vgl. [OASISUDDI] Abschnitt 3.8.1

4.4.2.2 UDDI API

Um zwischen dem UDDI Client und UDDI Server eine Kommunikation zu realisieren wird die UDDI API verwendet. Um den Einsatz auf möglichst vielen Systemen zu ermöglichen, wird die Kommunikation zwischen Client und Server durch XML-Dokumente realisiert. Unter diesen Voraussetzungen kann jedes System mit dem UDDI-Verzeichnis kommunizieren. Es existieren für den Server und Client unterschiedliche UDDI APIs.

Die Server-Funktionen, auch Nodes genannt, sind das Suchen, Veröffentlichen, Replizieren, Abonnieren und Verwalten. Im Folgenden werde diese kurz beschrieben.

Server-Funktion	Beschreibung
UDDI Inquiry	Stellt Funktionen zum Auffinden von Entitäten bereit. Es sind Interfaces verfügbar, um in einem UDDI-Verzeichnis eine Suche durchführen oder detaillierte Informationen über Entitäten abrufen zu können.
UDDI Publication	Stellt Funktionen für die Publizierung, Aktualisierung und Löschung von Entitäten in einem UDDI-Verzeichnis mit Hilfe von Interfaces zur Verfügung.

Tabelle 4.4.1: UDDI API – Server-Funktionen (1)

[82] Vgl. [OASISUDDI] Abschnitt 3.8.2

Server-Funktion	Beschreibung
UDDI Security	Um bestimmte Operationen ausführen zu können, bedarf es unter Umständen einer Authentifizierung. Daher kann Authentifizierungstoken zusammen mit dem Operationsaufruf gesendet werden. Die Operationen im UDDI Security API Set bieten Funktionen zum Holen oder Verwerfen solcher Tokens.
UDDI Custody Transfer	Stellt Operationen zur Übertragung von Verantwortlichkeiten und Besitzrechten für Entitäten innerhalb eines UDDI-Verzeichnisses von einem Nutzer auf einen anderen (oder zwischen verschiedenen Verzeichnissen).
UDDI Subscription	Stellt dem Nutzer die Funktionalität zur Verfügung bestimmte Entitäten zu abonnieren. Um festzulegen, für welche Entitäten dieses Abonnement erfolgen soll, werden die Funktionen des Inquiry API verwendet. Ein Abonnement kann beispielsweise alle Abfragen verwenden, die mit dem UDDI Inquiry API konform sind. So sind alle Entitäten, die dieser Abfrage entsprechen, für dieses Abonnement relevant. Ändert sich das Abfrage-Ergebnis, dann werden die Entitäten, die von dem Abonnement betroffen sind, ebenfalls geändert. Dem Abonnenten stehen zwei Möglichkeiten zur Verfügung, die passenden Entitäten zu überwachen: Asynchrone Benachrichtigung und synchrone Änderungsverfolgung. *Asynchrone Benachrichtigung* Ein Abonnent wird benachrichtigt, wenn sich eine Entität ändert, die der angegebenen Query entspricht. Somit kann der Abonnent eine Implementierung des UDDI Subscription Listener API bereitstellen. Diese wird dann aufgerufen. *Synchrone Änderungsverfolgung* Die Verzeichnisse können eine Operation bereitstellen, die dann jederzeit vom Abonnenten aufgerufen werden kann. Sie stellt Informationen darüber zur Verfügung, inwiefern sich die überwachten Entitäten geändert haben.
UDDI Replication	Das Replication API Set bietet Interfaces, die für die Replikation von Daten zwischen verschiedenen Verzeichnissen benutzt werden.

Tabelle 4.4.2: UDDI API – Server-Funktionen (2)

Die Client-Funktionen sind Abonnement-Push-Dienste und die Validierung von Referenzen.

Client-Funktionen	Beschreibung
UDDI Subscription Listener	Ändert sich der Status eines Dokuments so wird diese Operation jedes Mal von dem UDDI-Verzeichnis aufgerufen (siehe UDDI Subscription).
UDDI Value Set	In UDDI können Value Sets zum Speichern von Kategorisierungen verwenden. Wird eine bestimmte Kategorie beim Speichern oder Ändern einer Entität in dem Verzeichnis benutzt, ist das Verzeichnis in der Lage, eine externe Operation aufzurufen, um die Gültigkeit der Werte zu überprüfen.

Tabelle 4.4.3: UDDI API – Client-Funktionen

4.5 Web Service Inspection Language

UDDI stellt den bekanntesten Mechanismus zum Entdecken von Diensten dar. Dies bedeutet aber nicht das UDDI die einzige Möglich ist diesem Zweck gerecht zu werden. Die Komplexität und er Anspruch von UDDI ist in vielen Fällen zu hoch, insbesondere wenn nur ein einfacher Verweis auf ein WSDL-Dokument oder eine URL benötigt wird. Darüber hinaus geht ist die Ausrichtung von UDDI mehr in Richtung von Unternehmen. Aus diesem Grund haben IBM und Microsoft eine neue Untersuchungssprache für Web-Services entwickelt. Die Web Service Inspection Language (kurz WS-Inspection oder WS-I) kann zur Erzeugung von einfachen Verzeichnissen zur Servicebeschreibung verwendet werden.[83]

Beispiel 4.5.1: WS-Inspection einfaches Beispiel[84]

```
<?xml version="1.0"?>
          <inspection
            xmlns=http://schemas.xmlsoap.org/ws/2001/10/inspection/
            xmlns:uddi="http://schemas.xmlsoap.org/ws/2001/10/inspectio
            n/uddi/">
          <service>
            <abstract>Hallo Welt</abstract>
          <description
            referencedNamespace="http://schemas.xmlsoap.org/wsdl/"
            location="http://example.com/Hallo.wsdl"/>

            <description referencedNamespace="urn:uddi-org:api">
                  <uddi:serviceDescription
                  location="http://www.example.com/uddi/inquiryapi">
                  <uddi:serviceKey>4FA28580-5C39-…</uddi:serviceKey>
            </uddi:serviceDescription>
            </description>
          </service>
          <link
          referencedNamespace="http://schemas.xmlsoap.org/ws/2001/10/in
          spection/"
          location="http://example.com/moreservices.wsil"/>
</inspection>
```

[83] Vgl. [SnTiKu02] S. 126; Vgl. [IBMWSI] S. 2f
[84] [SnTiKu02] S.126

Das Beispiel 4.5.1 zeigt ein WS-Inspection-Dokument. Es wird ein Verweis auf einen Service dargestellt. Des Weiteren enthält das Beispiel zwei Beschreibungen. Eine WSDL-Beschreibung im oberen und eine UDDI-Beschreibung im unteren Teil des Beispiels.

Wenn WS-Inspection-Dokumente erzeugt wurden, sollten diese an einer gut zugänglichen beziehungsweise leicht zu finden Stelle hinterlegt werden. Normalerweise handelt es sich dabei um die Root-Verzeichnis eines Servers.[85]

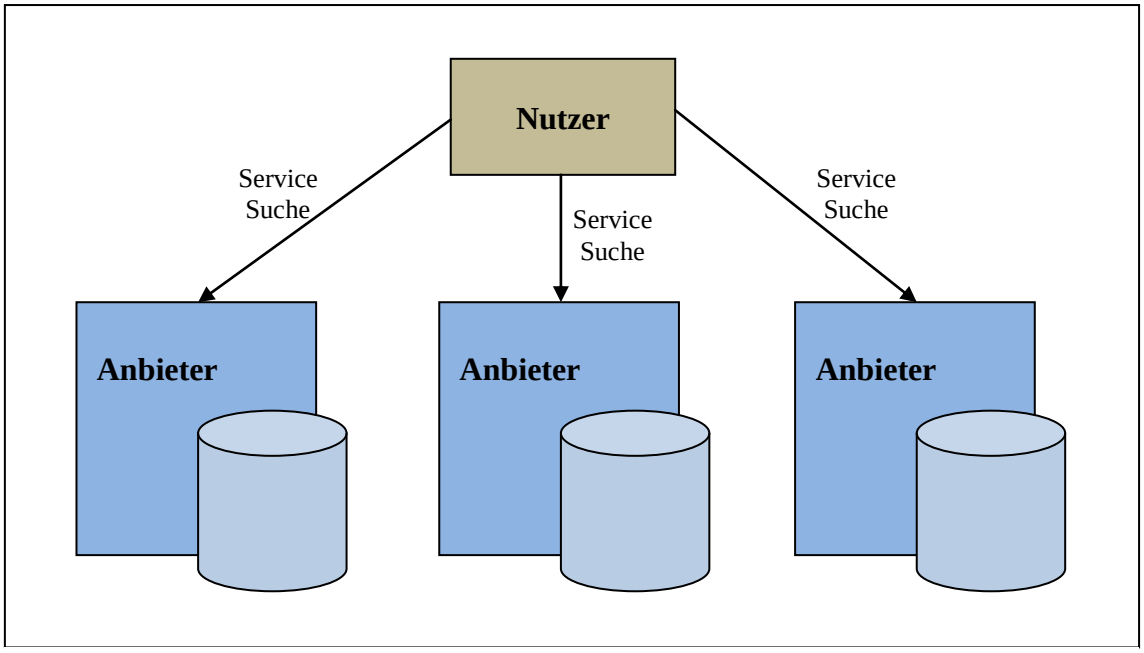

Abbildung 4.5.1: WS-Inspection – Service suche

Quelle: Vgl. [Melzer2010] S. 143

[85] Vgl. [SnTiKu02] S. 127

4.6 Web Service Business Process Execution Language

Die Web Service Business Process Execution Language (WS-BPEL) ermöglicht es Web-Services zu orchestrieren. Unter der Orchestrierung von Web-Services versteht man die Verknüpfung mehrerer Web-Services zu einem Ganzen. Abbildung 4.6.1 veranschaulicht das Prinzip der Orchestrierung und zeigt gleichzeitig die Funktionsweise von Choreographie auf.[86]

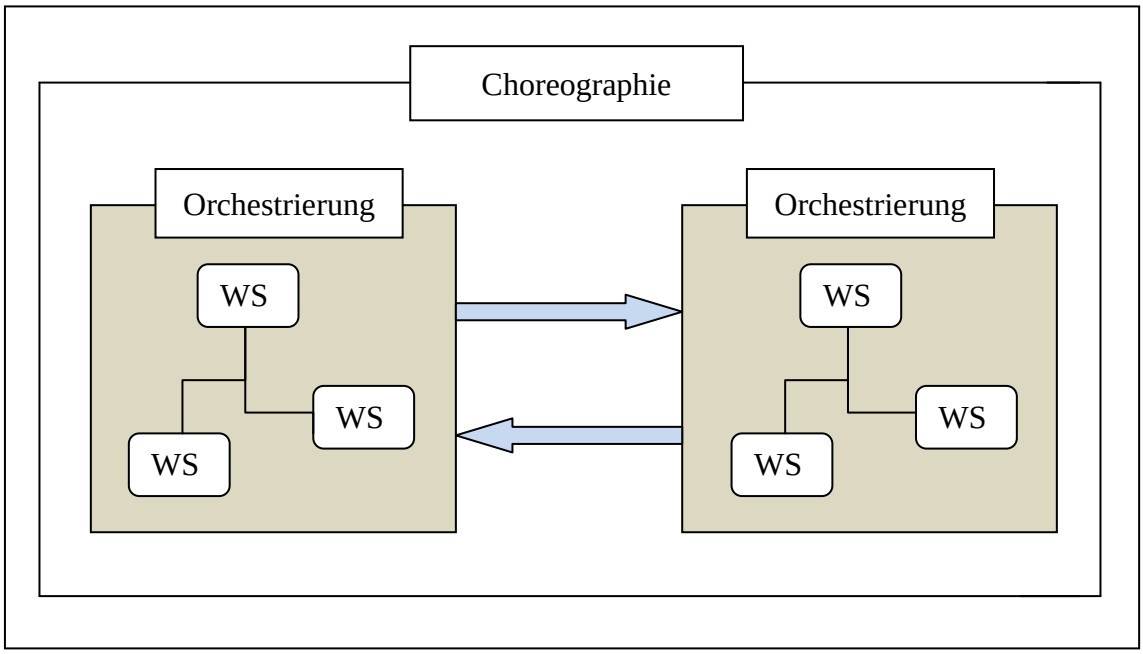

Abbildung 4.6.1: Choreographie und Orchestrierung

Quelle: Vgl. [FinZep09] S. 71

Die in Abbildung 4.6.1 dargestellten Orchestrierungen können miteinander Kommunizieren und Daten austauschen. Es lassen sich mit Hilfe der Orchestrierung Geschäftsprozesse abbilden. WS-Choreographie Description Language (WS-CDL) ist eine Beschreibungssprache für die Choreographie, auf die aber nicht weiter eingegangen wird (eine Spezifikation ist beim W3C zu finden).

Ein ausführliches XML-Dokument zu WS-BPEL würde an dieser Stelle etwas ausufern, daher verweise ich auf die Spezifikation [OASISBPEL]. Auf einige wichtige Elemente wird im Folgenden kurz eingegangen.

Die Aktivitäten eines WS-BPEL-Prozesses lassen sich in grundlegende (*basic activities*) und strukturierte (*structured activities*) unterteilen. Zu den grundlegenden Aktivitäten zählen unter anderem *receive*, *reply*, *invoke*, *assign*, *compensate*, *wait* und *exit*.

[86] Vgl. [OASISBPEL]; Vgl. [FinZep09] S. 71f

Aktivität	Beschreibung
receive	Startpunkt für BPEL-Prozess, Initialisierung erfolgt durch eine Anfrage
reply	Liefert Antwort an anfragenden Service und beendet den Prozess
invoke	Beteiligt andere Services an dem Prozess
assign	Zuweisen von Werten an andere Variable
compensate	Führt bei einem Fehler einen Rollback aus
wait	Prozess wartet für einen bestimmten Zeitraum
exit	Prozess wird sofort beendet

Tabelle 4.6.1: WS-BPEL – grundlegende Aktivitäten

Quelle: Vgl. [OASISBPEL] Abschnitt 10

Die strukturierten Aktivitäten befassen sich mit der Ablaufsteuerung von Prozessen. Wichtig sind hierbei *sequence, flow, pick, scope, while, foreach* und *if.*

Aktivität	Beschreibung
sequence	Aktivitäten werden der Reihe nach abgearbeitet
flow	Aktivitäten werden parallel abgearbeitet
pick	Prozess wartet auf ankommende Meldungen
scope	Zerlegt den Prozess in verschiedene Teile
while	Wiederholungschleife
foreach	*For*-Schleife
if	*If*-Anweisung

Tabelle 4.6.2: WS-BPEL – strukturierte Aktivitäten

Quelle: Vgl. [OASISBPEL] Abschnitt 11

Abschließend wird noch auf die Behandlung von bestimmt Ereignissen eingegangen. Die wichtigen BPEL-Handler sind eventHandler, faultHandler, compensationHandler und terminationHandler.

Handler	Beschreibung
eventHandler	Reagiert auf allgemeine Ereignisse
faultHandler	Definierte Fehlerbehandlung wird ausgeführt
compensationHandler	Wird durch *compensate* aufgerufen
terminationHandler	Meldet einen beendeten Prozess

Tabelle 4.6.3: WS-BPEL – Handler

Quelle: Vgl. [OASISBPEL] Abschnitt 12

4.7 Common Object Request Broker Architecture

Die Common Object Request Broker Architecture (CORBA) wurde erstmals im August 1991 von der Object Management Group (OMG) spezifiziert. CORBA ist eine offene, herstellerunabhängige Architektur und Infrastruktur, welche Anwendungen gestattet über Netzwerke zusammenzuarbeiten. Mit Hilfe des Object Request Broker (ORB) können CORBA-basierte Programme mit jedem anderen CORBA-basierenden Programm zusammenarbeiten, unabhängig von Anbieter, Programmiersprache, Netzwerk und Betriebssystem. Durch diese Eigenschaften ist CORBA vielseitig einsetzbar. Das primäre Einsatzgebiet liegt bei Systemen, die eine hohe Zuverlässigkeit, viele Zugriffe und Benutzer aufweisen (z.B. Webseiten).[87]

Der Object Request Broker (ORB) stellt alle Mechanismen, die für das Auffinden, Vorbereiten und Empfangen von Objekt-Implementationen zur Bearbeitung der Anfrage, bereit. Das Interface, welche der Client sieht, ist unabhängig von der Programmiersprache, dem Ort oder jeder anderen Beziehung die nicht in dem Interface dargestellt wird.[88]

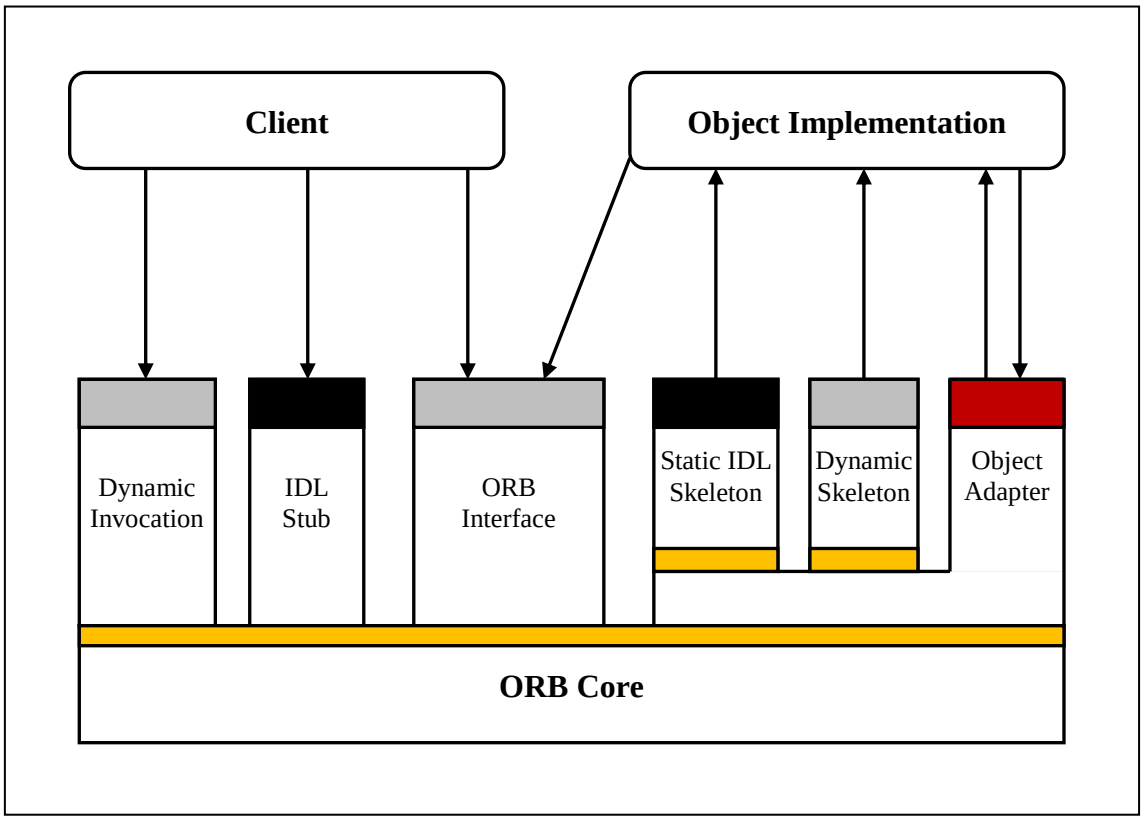

Abbildung 4.7.1: CORBA Übersicht

Quelle: Vgl. [OMG11b] S. 14

[87] Vgl. [OMG11a]
[88] Vgl. [OMG11b] S.13

Abbildung 4.2.1 zeigt die Struktur eines einzelnen ORB und den Weg die eine Anfrage vom Client nehmen kann. Die farbigen Bereiche stellen die Schnittstellen des ORB dar.

- *Dynamic Invocation*, *ORB Interface* und *Dynamic Skeleton* besitzen identische Schnittstellen über alle ORB-Implementationen hinweg (**grau**)
- *IDL Stub* und *Static IDL Skeleton* existieren für jeden Objekttyp (**schwarz**)
- *Object Adapter* können mehrfach auftreten (**rot**)
- ORB-abhängige Schnittstellen existieren zwischen allen Komponenten (**gelb**)

Um eine Anfrage zu starten, kann der Client das Dynamic Invocation Interface nutzen (unabhängig vom Zielobjekt) oder einen OMG Interface Definition Language Stub (OMG IDL Stub; den spezifischen Stub des Zielobjekts). Der Client hat Zugriff auf eine Objektreferenz, kennt die Typen der Objekte für gewünschte Operation und kann für einige Funktionen auch direkt mit dem ORB interagieren. Der ORB seinerseits ermittelt den Implementationscode, transferiert die Parameter und überträgt die Kontrolle zur Object Implementation, entweder über ein IDL Skeleton oder ein Dynamic Skeleton. Skeletons besitzen eine spezifisches Interface und einen Object Adapter. Die Object Implementation kann zum Ausführen einer Anfrage, über einen ausgewählten Object Adapter, den ORB Core rufen um benötigte Services auszuführen (während der Bearbeitung der Anfrage oder zu einem späteren Zeitpunkt). Wenn die Anfrage abgearbeitet ist, werden die Kontrolle und der Rückgabewert an den Client zurückgegeben.[89]

Die Definition eines Interface zu einem Objekt kann auf zwei Arten festgelegt werden.[90]

- Interfaces können statisch in einer Interface Definition Language (IDL) definiert werden (hier OMG IDL). Die Typen und Parameter der Objekte, welche zur Ausführung der Operation nötig sind werden darin festgelegt.
- Interfaces können alternativ oder zusätzlich zu einem Interface Repository Service hinzugefügt werden. Die Komponenten eines Interface präsentiert der Service als Objekte.

Wie das Interface und die Implementationsinformationen dem Client und der Object Implementation zur Verfügung gestellt werden zeigt die Abbildung 4.2.2. Das Interface wird in den IDL Definitions bzw. dem Interface Repository definiert. Die Definition

[89] Vgl. [OMG11b] S. 14 ff
[90] Vgl. [OMG11b] S. 15

wird zur Generierung des Stubs und des Skeletons verwendet. Die Implementationsinformationen eines Objekts werden zum Zeitpunkt der Installation bereitgestellt und im Implementation Repository abgespeichert umso für eine Anfrage bereitzustehen.

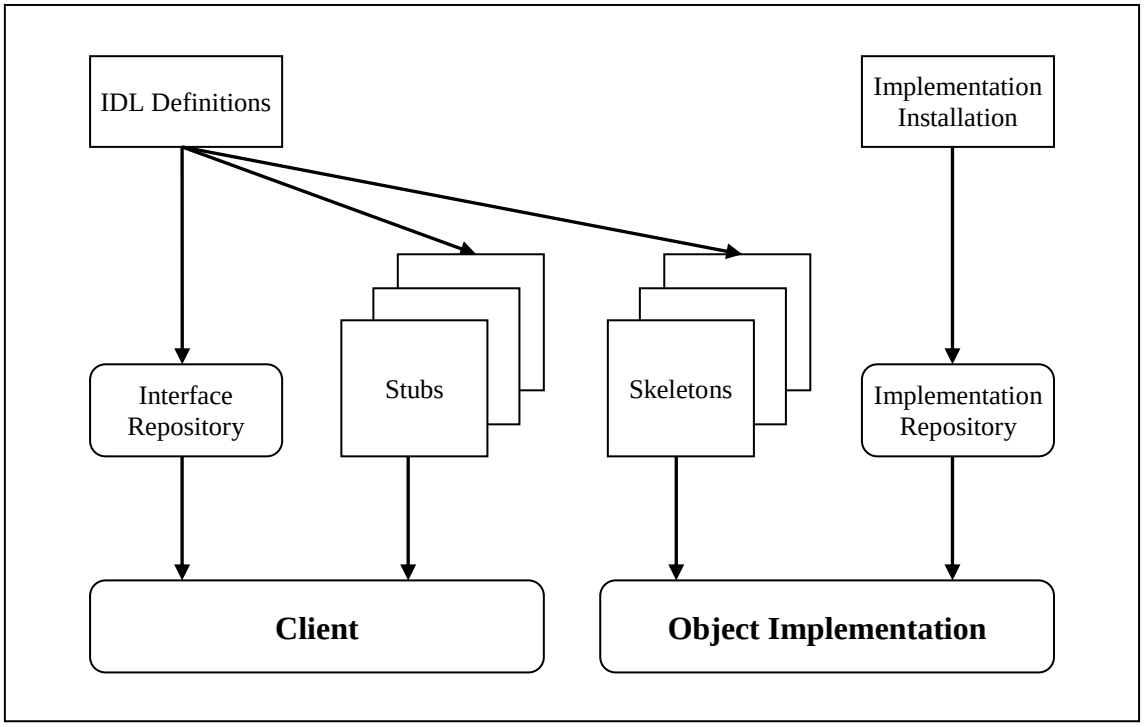

Abbildung 4.7.2: CORBA Interface und Implementation Repositories

Quelle: Vgl. [OMG11b] S. 17

Das *General Inter-ORB Protocol* (GIOP) wurde mit der CORBA Version 2.0 definiert. GIOP ist ein abstraktes Protokoll zur Kommunikation.

Die Kommunikation inner- und außerhalb von CORBA kann auch über das *Internet Inter-ORB Protocol* (IIOP) realisiert werden, welches eine Weiterentwicklung von GIOP ist. Dieses Protokoll ermöglicht es das Programme auf unterschiedlichen Systemen und mit unterschiedlicher Programmiersprache miteinander über das Internet kommunizieren können.

4.8 Auswertung und Vergleich der serviceorientierten Standards

Die zuvor beschrieben Standards haben fast alle dasselbe Ziel und zwar den Ablauf und die Verwendung von Web-Services zu vereinfachen oder eine größere Funktionalität anzubieten.

Die Abbildung 4.8.1 ordnet die beschriebenen Standards in den Rahmen der serviceorientierten Architektur ein.

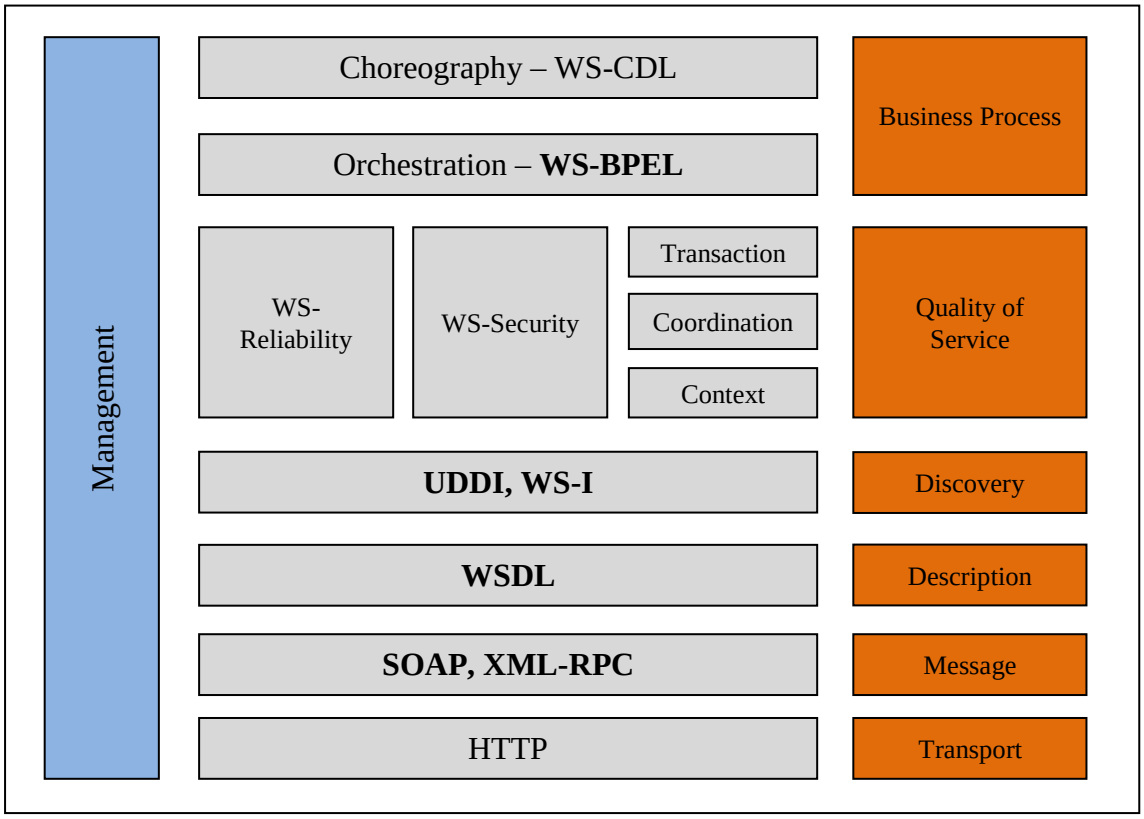

Abbildung 4.8.1: Einordnung von serviceorientierten Standards

Quelle: Vgl. [FinZep09] S. 69

CORBA nimmt hier eine Sonderstellung ein, da es zum einen nicht auf XML basiert und zum anderen kann es, dadurch dass es sich in mehrere Bestandteile zerlegen lässt, unterschiedlichen Stellen zuordnen werden. CORBA wird im Folgenden noch näher betrachtet.

SOAP und der XML-RPC sind in erster Linie für den Austausch und das verpacken von Nachrichten zuständig. SOAP bietet durch die Verwendung von Header und Body die Möglichkeit zusätzliche Informationen zu transportieren. Wohingegen der XML-RPC durch die einfache Datenstruktur einen geringes Datenaufkommen besitzt. Eine Servicebeschreibung lässt sich ableiten, wenn die XML Dokumente betrachtet werden. Tabelle 4.8.1 auf der nächsten Seite zeigt eine Gegenüberstellung von SOAP und RPC.

Eigenschaft	SOAP	XML-RPC
XML	Ja	Ja
Funktionalität	Gering	Sehr Gering
Fehlerbehandlung	Ja	Nein
Datenaufkommen	Moderat	Gering
Servicesuche	Nein	Nein
Servicebeschreibung	(Nein)	(Nein)

Tabelle 4.8.1: SOAP vs. XML-RPC

UDDI bietet im Gegensatz zu WS-I ein zentrales Verzeichnis an. Die Suche in diesem Verzeichnis wird mit hoher Wahrscheinlichkeit ein zufriedenstellendes Ergebnis liefern. WS-I hingegen nutzt viele kleiner dezentrale Verzeichnisse, welche meist direkt beim Anbieter vorliegen. Vorteil der kleinen Verzeichnisse ist eine einfache Pflege und ein Nutzer der positive Erfahrungen mit dem Anbieter gemacht hat, wird mit hoher Wahrscheinlichkeit den Anbieter erneut aufsuchen.[91] Die folgende Tabelle 4.8.2 zeigt eine Gegenüberstellung des UDDI Standards und der Web Service Inspection Language.

Eigenschaft	UDDI	WS-Inspection
XML	Ja	Ja
Funktionalität	Hoch	Moderat
Fehlerbehandlung	Ja	Ja
Datenaufkommen	Sehr hoch	Moderat/Hoch
Servicesuche	Ja	Ja
Servicebeschreibung	Ja	Ja

Tabelle 4.8.2: UDDI vs. WS-Inspection

Bei dem Vergleich von Web-Services und CORBA wird deutlich das beide aus einer Menge von Standards bestehen. CORBA besitzt einige Vorteile im Bereich der Sicherheit. Die Web-Services holen aber in diesem Bereich immer mehr auf, siehe Web Service Security. Durch das einfache erweitern von XML-Dokumenten können ohne größere Schwierigkeiten neue Komponenten entwickelt werden. Die Tabelle 4.8.3 auf der nächsten Seite zeigt eine Gegenüberstellung von Web-Services und CORBA.

[91] Vgl. [Melzer2010] S. 142f

	Web-Service	**CORBA**
Protokoll	SOAP, http, XML Schema	GIOP, IIOP
Standortsbestimmung	URLs, URI	URLs
Interfacespezifikation	WSDL	OMG IDL
Benennung & Verzeichnisse	UDDI, WS-I	Interface Repository, Naming Service
XML	Ja	Nein
Funktionalität	Hoch	Hoch
Fehlerbehandlung	Ja	Ja
Datenaufkommen	Hoch	Gering/Moderat
Servicesuche	Ja	Ja
Servicebeschreibung	Ja	Ja

Tabelle 4.8.3: Web-Service vs. CORBA

Egal für welchen Standard man sich am Ende entscheidet, mit Abstrichen muss man bei der einen oder anderen Variante immer rechnen.

Literaturverzeichnis

[Bengel04] Bengel, Günther: Verteilte Systeme: Grundlagen und Praxis des Client-Server-Computing. Friedrich Vieweg & Sohn Verlag: 3. Auflage, Wiesbaden, 2004

[Cerami02] Cerami, Ethan: Web Service Essentials. O'Reilly & Associates, Inc.: Sebastopol, 2002

[Dadam96] Dadam, Peter: Verteilte Datenbanken und Client/Server-Systeme: Grundlagen, Konzepte und Realsierungsformen, Springer-Verlag: Berlin, Heidelberg, 1996

[Dumke03] Dumke, Reiner: Software Engineering: Eine Einführung für Informatiker und Ingenieure. Friedrich Vieweg & Sohn Verlag: 4. Auflage, Wiesbaden, 2003

[Erl2004] Erl, Thomas: Service-oriented architecture: a field guide to integrating XML and Web services. Prentice Hall International: 2004

[FinZep09] Finger, Patrick; Zeppenfeld, Klaus: SOA und WebServices. Springer-Verlag: Berlin, Heidelberg, 2009

[IBMUDDI] IBM: Understanding UDDI http://www.ibm.com/developerworks/webservices/library/ws-featuddi/. 17.Juli 2011

[IBMWSI] IBM: Web Services Inspection Language (WS-Inspection) 1.0 http://public.dhe.ibm.com/software/dw/specs/ws-wsilspec/ws-wsilspec.pdf. 25. Juli 2011

[Melzer2010] Melzer, Ingo: Service-orientierte Architekturen mit Web Services: Konzepte – Standards – Praxis. Spektrum Akademischer Verlag: 4. Auflage, Heidelberg, 2010

[Moos2008] Moos, Alfred: XQuery und SQL/XML in DB2-Datenbanken: Verwaltung und Erzeugung von XML-Dokumenten in DB2. Vieweg + Teubner, GWV Fachverlage GmbH: 1 Auflage, Wiesbaden, 2008

[OASISBPEL] OASIS: Web Services Business Process Execution Language http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. 14. August 2011

[OASISUDDI] OASIS: UDDI Version 3.0.2 http://www.uddi.org/pubs/uddi_v3.htm. 17. Juli 2011.

[OMG11a] OMG: CORBA FAQ (2011) – CORBA BASICS http://www.omg.org/gettingstarted/corbafaq.htm. 21. Mai 2011.

[OMG11b] OMG: CORBA 3.1 – CORBA- Part 1: Interfaces, v3.1 http://www.omg.org/spec/CORBA/3.1/Interfaces/PDF. 21. Mai 2011.

[Sarang] Sarang, Poornachandra: Pro Apache XML. Springer-Verlag: New York, 2006

[SnTiKu02] Snell, James; Tidwell, Doug; Kulchenko, Pavel: Programming Web Service with SOAP. O'Reilly & Associates, Inc.: Sebastopol, 2002

[W3C2011a] W3C: SOAP Version 1.2 Part 0: Primer (Second Edition) http://www.w3.org/TR/2007/REC-soap12-part0-20070427/. 30. Mai 2011.

[W3C2011b] W3C: Web Service Description Language (WSDL) Version 2.0 Part 1: Core Language
http://www.w3.org/TR/wsdl20/. 1. Juni 2011.

[W3C2011b] W3C: Web Service Description Language (WSDL) Version 2.0 Part 0: Primer
http://www.w3.org/TR/wsdl20-primer/. 3. Juni 2011.

[W3CSCHEMAc] W3C:XML Schema Part 2:Datatypes Second Edition
http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/. 6. Juni 2011.

[W3CSCHEMAb] W3C: XML Schema Part 1: Structures Second Edition
http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/. 5. Juni 2011.

[W3CXMLa] W3C: XML Essentials
http://www.w3.org/standards/xml/core. 17. Mai 2011.

[W3CXML-NS] W3C: Namcespaces in XML 1.0 (Third Edition)
http://www.w3.org/TR/REC-xml-names/. 29. Mai 2011.

[W3SCHOOL] Refsnes Data: DTD Tutorial
http://www.w3schools.com/dtd/. 18. Mai 2011.

[W3SCHOOLb] Refsnes Data: XML Elements vs. Attributes
http://www.w3schools.com/dtd/dtd_el_vs_attr.asp. 18. Mai 2011

[White76] White, James E. (1976): RFC707 - A High-level framework for network-based resource sharing.
http://tools.ietf.org/html/rfc707. 20. Mai 2011.

[Zimmermann/Tomlinson/Peuser] Zimmermann, Olaf; Tomlinson, Mark; Peuser, Stefan: Perspectives on Web Services: Applying SOAP, WSDL and UDDI to Real-World Projects. Springer-Verlag: Berlin; Heidelberg, 2003

Abschließende Erklärung

Ich versichere hiermit, dass ich die vorliegende Diplomarbeit selbständig, ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Berlin, den 5. Oktober 2011