
ulm university universität uulm
Universität Ulm
Institut für Datenbanken und Informationssysteme

JAX-WS
Analyse, Einordnung und Bewertung der Technologie anhand praxisnaher Beispiele

Diplomarbeit im Studienfach Informatik

vorgelegt von
Ralf Klein Heßling

WS 2007/2008

1. Gutachter: Prof. Dr. Manfred Reichert
2. Gutachter: Prof. Dr. Peter Dadam

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Abkürzungsverzeichnis

bspw.	beispielsweise.
bzw.	beziehungsweise.
d.h.	das heißt.
DOM	Document Object Model.
EDI	Electronic Data Interchange.
EJB	Enterprise JavaBeans.
etc.	et cetera.
HTTP	Hypertext Transfer Protocol.
IDL	Interface Definition Language.
IIOP	Internet Inter-ORB Protocol.
inkl.	inklusive.
IT	Informationstechnik.
JAR	Java Archive.
Java EE	Java Platform, Enterprise Edition.
Java SE	Java Platform, Standard Edition.
JAX-RPC	Representational State Transfer.
JAX-WS	Java API for XML - Web Services.
MIME	Multipurpose Internet Mail Extensions.
NASSL	Network Application Service Specification Language.
PFS	Person Finder Service.
PISys	Person Info System.
REST	Representational State Transfer.
RMI	Remote Method Invocation.
RPC	Remote Procedure Call.
SAX	Simple API for XML.

SDL	Service Description Language.
SEI	Service Endpoint Interface.
SI	Standard Implementation.
SIB	Service Implementation Bean.
SOA	Serviceorientierte Architektur.
StAX	Streaming API for XML.
u.a.	unter anderem.
UDDI	Universal Description, Discovery and Integration.
usw.	und so weiter.
WAR	Web Application Archive.
WSDL	Web Service Description Language.
WWW	World Wide Web.
XML	Extensible Markup Language.
XPath	XML Path Language.
XSD	XML Schema.
z.B.	zum Beispiel.

1 Zusammenfassung

Ziel dieser Diplomarbeit war die Analyse, Einordnung und Bewertung der Java Webservice Technologie JAX-WS anhand verschiedenartiger Implementierungen. Angefangen wurde zunächst mit der Einordnung dieser Technologie anhand der Definition der Begriffe SOA und Webservice. Hier wurde gezeigt, dass eine SOA nachträglich in die bestehende IT-Struktur eines Unternehmens eingebracht wird um die Kommunikation mit anderen Unternehmen zu ermöglichen. Diese Eigenschaft entspricht dem in dieser Diplomarbeit beschriebenen „Start von WSDL & Java“ Szenario bei der Entwicklung eines Webservices. JAX-WS ist, als Instanz eines Webservices, jedoch auch für die möglichen Szenarien „Start von WSDL“ und „Start von Java“ ausgerichtet.
Da es wenige zu JAX-WS vergleichbare Technologien in der Java Welt gibt, bezog sich die Bewertung von JAX-WS neben dessen Nutzen in den obigen Szenarien auch auf einen Vergleich mit der Vorgängerversion von JAX-WS, der JAX-RPC, sowie einer Analyse inwiefern die von einer WSDL spezifizierten Möglichkeiten in JAX-WS umgesetzt werden können. Letztgenannter Punkt bezieht sich hauptsächlich auf die Fähigkeiten von JAXB, einer zwar mittlerweile unabhängigen Spezifikation für das Binding zwischen XML und Java, die jedoch einen wichtigen Stützpfeiler für die JAX-WS Architektur bildet. Die sehr eingeschränkten Möglichkeiten des Bindings zwischen XML und Java waren, neben der umständlichen Erstellung der Konfigurationsdateien, eines der hauptsächlichen Kritikpunkte von JAX-RPC. Beide Punkte wurden in JAX-WS (bzw. JAXB) über die XML/Java Annotationen stark vereinfacht und verbessert.
Auch in Bezug auf die Möglichkeiten der Implementierung bietet JAX-WS ein sehr reichhaltiges Repertoire dieser an. So wurden viele mögliche Implementierungen exemplarisch analysiert und deren Einsatz in Bezug auf die obigen drei Szenarien bewertet. Als mögliche Implementierungen sind dabei insbesondere die Entwicklung eines JAX-WS Servers unter der Java SE, sowie die clientseitigen Möglichkeiten der asynchronen Kommunikation zu nennen, welche von JAX-RPC noch nicht realisiert wurden.
Zusammenfassend stellt sich heraus, dass die Stärken von JAX-WS insbesondere in den für die Einführung einer SOA untypischen Szenarien „Start von WSDL“ und „Start von Java“ liegen. Dies liegt zum einen an den stark eingeschränkten Mapping-Fähigkeiten von JAXB, zum anderen gibt es keine trivialen Lösungen für das „Start von WSDL & Java“ Szenario. Dennoch ist JAX-WS eine Spezifikation an der es nur wenige Kritikpunkte gibt und welche die Komplexität bei der Entwicklung eines Webservices auf ein absolutes Minimum reduziert.

2 Einleitung

Dieses Kapitel beginnt mit einer kurzen Einführung in das Themengebiet Webservices. Dabei wird vor allem der Nutzen und das typische Einsatzgebiet eines Webservices aufgezeigt. Anhand dieser Motivation wird im darauf folgenden Abschnitt auf die Zielsetzung dieser Diplomarbeit eingegangen. Anschließend folgen Erläuterungen zum theoretischen sowie praktischen Teil dieser Diplomarbeit. Dabei wird vor allem verdeutlicht, inwiefern diese aufeinander aufbauen und miteinander verstrikt sind. Die dabei verwendeten Konventionen werden am Ende dieses Kapitels erläutert.

2.1 Motivation

Im Zuge der Internationalisierung der Märkte und der verstärkten Entwicklung des tertiären Sektors sind Unternehmen oftmals auf eine schnelle Bereitstellung von Dienstleistungen und Gütern anderer Unternehmen angewiesen. Darunter fallen z.B. Unternehmen aus der Automobilindustrie, die mit unterschiedlichen Zulieferern effizient kooperieren und kommunizieren müssen. Aber auch ein Reisebüro ist auf eine schnelle Bearbeitung der Flugbuchungen seitens der Fluggesellschaften angewiesen.
Nun verwirklicht jedoch jedes Unternehmen seine eigene IT-Struktur und diese sind unternehmensübergreifend oftmals nicht kompatibel. Diese Situation war der Ausgangspunkt für die Entwicklung einer standardisierten Schnittstelle und eines standardisierten Nachrichtenformats, welche für die Kommunikation im Business-to-Business (B2B) Bereich ausgelegt sind. Also für die Entwicklung der Webservice Technologie.
Mit der Einführung der Webservice Technologie entstand jedoch ein neues Problem, welches die Motivation für diese Diplomarbeit liefert – die gebräuchliche Webservice Spezifikation ist zu komplex um sie mit einfachen Mitteln in die IT-Struktur eines Unternehmens einzuführen. Sun erkannte dieses Problem und hat im Jahr 2006 die Java API for XML - Web Services (JAX-WS) veröffentlicht, welche die Entwicklung eines Java-basierten Webservices erheblich vereinfachen soll.

2.2 Zielsetzung

Zielsetzung dieser Diplomarbeit ist die Analyse und Bewertung der Möglichkeiten, die JAX-WS für die Erstellung eines Webservices anbietet. Für die Analyse der von JAX-WS angebotenen Implementierungsmöglichkeiten werden insbesondere Vergleiche mit den Eigenschaften der Webservice Description Language (WSDL) durchgeführt und gezeigt, wie diese in JAX-WS realisiert werden. Die Bewertung erfolgt nach dem Nutzen und der Güte der einzelnen JAX-WS Implementierungen in den für die Entwicklung eines Webservices typischen Szenarien „Start von WSDL & Java“, „Start von WSDL“ und „Start von Java“. Die dabei durchgeführten prototypischen Implementierungen werden überdies mit den Möglichkeiten der Vorgängerversion von JAX-WS, der Java API for XML-Based RPC (JAX-RPC) verglichen, wobei dieser Vergleich ein Teil der Bewertung darstellt.
Letzendlich soll gezeigt werden, wie gut die obigen Szenarien in JAX-WS umgesetzt werden können. Dabei sollen zum einen die Probleme einer jeden Implementierung aufgezeigt, zum anderen aber auch die positiven Aspekte hervorgehoben werden.

2.3 Aufbau des praktischen Teils

Alle in dieser Diplomarbeit aufgezeigten prototypischen Implementierungen eines JAX-WS Webservices werden in einem Projekt namens *Person Info System (PISys)* realisiert. PISys ist ein exemplarisches Beispielprojekt und demonstriert die praktische Umsetzbarkeit der in späteren Kapiteln beschriebenen Möglichkeiten und Eigenschaften der Java Architecture for XML Binding (JAXB) und JAX-WS. Abbildung 2.1 zeigt den Aufbau des Person Info Systems, welches aus Gründen der Einfachheit sowohl clientseitig als auch serverseitig in Java SE realisiert ist. Dies schränkt die Möglichkeiten von JAXB und JAXWS, ausgenommen der Dependency Injection (s. Kapitel 7.1.1), in keinster Weise ein. Der im PISys realisierte Webservice trägt den Namen *Person Finder Service (PFS)* und dient der Anzeige von Daten von im Person Info System eingetragenen Personen. Dazu implementiert der Person Finder Service einen Port namens *Person Finder Port*, welcher die Operation `<pfs:getPerson>` mit dem Parameter `name` vom Typ `xs:string` anbietet. Rückgabewert dieser Methode sind Daten über den im Parameter `name` angegebenen Namen der Person. Diese Daten beinhalten unter anderem die Postleitzahl vom Wohnort, die Interessen, den Familienstand und das Alter der gesuchten Person. Diese Daten werden in einem Element vom Typ `<pfs:getPersonResponse>` zum Aufrufer zurückgeliefert. In Kapitel 4 wird genauer auf die vom Person Finder Service verwendete WSDL eingegangen, welche diese Operation definiert.
Auf der Implementierungsseite besteht der Person Finder Service aus drei serverseitigen und sieben clientseitigen Implementierungen. Alle diese Implemen-

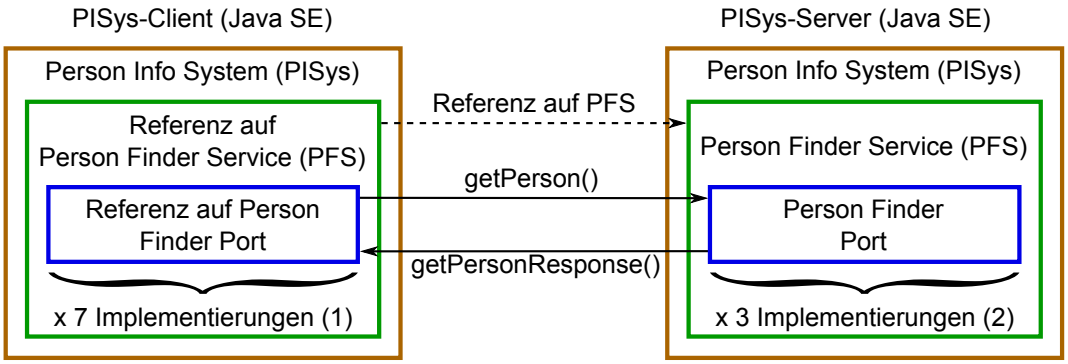

Abbildung 2.1: Aufbau des Person Info Systems
Quelle: Eigene Darstellung

tierungen liefern äquivalente Ausgaben bzw. stellen äquivalente Anfragen. Die clientseitigen Implementierungen sind im PISys-Client Projekt in der Klasse `de.pisys.pfs.client.gui.ClientGui` realisiert und werden in folgender Auflistung vorgestellt:

- **1 – Implementierungen des PISys-Client**
- `runSyncGen()` – Synchrone Implementierung eines generierten PISys-Clients.
- `runSyncManLowLevel()` – Synchrone Implementierung eines PISys-Clients über die Low-Level XML API (Source).
- `runSyncManJAXB()` – Synchrone Implementierung eines PISys-Clients über eine benutzerdefinierte JAXB-Klasse.
- `runAsyncManLowLevelPolling()` – Asynchrone Implementierung eines PISys-Clients mit dem Polling-Modell über die Low-Level XML API (Source).
- `runAsyncGenPolling()` – Asynchrone Implementierung eines generierten PISys-Clients mit dem Polling-Modell.
- `runAsyncManLowLevelCallback()` – Asynchrone Implementierung eines PISys-Clients mit dem Callback-Modell über die Low-Level XML API (Source).
- `runAsyncGenCallback()` – Asynchrone Implementierung eines generierten PISys-Clients mit dem Callback-Modell.

Auf die genaue Bedeutung dieser Implementierungen wird im Kapitel zum JAX-WS Client (s. Kapitel 7) genauer eingegangen. Dort werden insbesondere die Unterschiede dieser Implementierungen aufgezeigt, wie diese in JAX-WS intern realisiert werden und welche dieser Implementierungen in welchem der für die

Entwicklung eines Webservices typischen Szenarios besser geeignet ist.
Serverseitig werden drei mögliche Implementierungen des Person Finder Services erläutert, welche im PISys-Server Projekt in drei unterschiedlichen Paketen untergebracht sind. Die folgende Auflistung gibt einen Überblick, welche Implementierungen in welchen Paketen zu finden sind.

- **2 – Implementierungen des PISys-Servers**
- `de.pisys.pfs.server.gen` – Generierten Implementierung des PISys-Servers.
- `de.pisys.pfs.server.jaxb` – Implementierung des PISys-Servers über eine benutzerdefinierte JAXB-Klasse.
- `de.pisys.pfs.server.lowlevel` – Implementierung des PISys-Servers über die Low-Level XML API (SOAPMessage).

Die Vorstellung, Analyse und Bewertung dieser serverseitigen Implementierungen finden sich dabei im Kapitel zum JAX-WS Server (s. Kapitel 8) wieder.

2.4 Aufbau der theoretischen Teils

Die Diplomarbeit ist in mehrere Kapitel unterteilt, deren Inhalt und Zusammenhang durch folgende Auflistung verdeutlicht wird. Wie in dieser Auflistung zu erkennen ist, stellt die Entwicklung des im letzten Abschnitt vorgestellten Person Info System den „roten Faden“ dieser Diplomarbeit dar.

- *Kapitel 3:* Das Thema Webservices grundiert aufgrund seiner verteilten und komplexen Architektur auf einer Vielzahl von Standards und Definitionen, die für Verwirrung sorgen können. Aus diesem Grund wird in Kapitel 3 eine genaue Abgrenzung und Erläuterung aller verwendeten Begriffe vorgenommen. Dies erleichtert das Verständnis für die Thematik und beseitigt eventuelle Mehrdeutigkeiten und Unschärfen. Auf der anderen Seite spiegelt diese Abgrenzung der Webservice Technologie das wieder, was in JAX-WS möglich sein sollte.
- *Kapitel 4:* Die WSDL ist die Grundlage einer jeden clientseitigen Webservice Implementierung und liefert die genaue Beschreibung der von einem Webservice angebotenen Operationen. Da JAX-WS zu großen Teilen auf der WSDL beruht, wird in diesem Kapitel relativ detailliert auf den Aufbau und die unterschiedlichen Styles einer WSDL eingegangen. Als exemplarisches Beispiel einer WSDL wird dabei die WSDL des Person Finder Services vorgestellt.

- *Kapitel 5:* Dieses Kapitel widmet sich dem Vorgänger von JAX-WS, der JAX-RPC Technologie. Inhaltlich wird in diesem Kapitel der Versuch unternommen, einige Aspekte vom Person Finder Port in JAX-RPC nach zu implementieren. Dabei werden vor allem die Defizite dieser Implementierung herausgearbeitet und in späteren Kapiteln über JAX-WS gezeigt, wie diese dort gelöst wurden.

- *Kapitel 6:* In JAX-WS wurde der komplette Part, welcher das Binding zwischen XML und Java betrifft, in eine unabhängige Spezifikation namens JAXB ausgelagert. JAX-WS macht dabei regen Gebrauch von JAXB, was sich auch in vielen Implementierungen des Person Finder Services niederschlägt. Aus diesem Grund wird in diesem Kapitel sehr detailliert auf die JAXB Technologie eingegangen und gezeigt, mit welcher Güte JAXB das Binding der Datentypdefinitionen der WSDL des Person Finder Services durchführt.

- *Kapitel 7:* Dies ist das erste Kapitel welches sich dem Kern der JAX-WS Spezifikation, der Implementierung eines JAX-WS Webservices, widmet. Dabei werden alle relevanten clientseitigen Implementierungsmöglichkeiten des Person Finder Services exemplarisch behandelt und beurteilt.

- *Kapitel 8:* Im Gegensatz zum vorigen Kapitel behandelt dieses Kapitel die serverseitigen Implementierungen des Person Finder Services. Auch hier werden die verschiedenen Implementierungsmöglichkeiten analysiert und eine Bewertung dieser gefällt.

In fast allen dieser Kapitel werden die behandelten Technologien durch sog. Listings veranschaulicht. In diesen Listings werden zum einen XML-Ausschnitte zum anderen aber auch Java-Quelltexte dargestellt, die sich ebenfalls im Fließtext dieser Diplomarbeit niederschlagen. Dadurch entstehen eine Reihe von Ungereimtheiten bezüglich der Zusammenführung dieser Sprachen, welche im folgenden Abschnitt durch die Einführung von Konventionen aufgelöst werden.

2.5 Konventionen

Diese Diplomarbeit verwendet eine Reihe von Konventionen, welche eventuelle Mehrdeutigkeiten zwischen den verwendeten Sprachen und Technologien klären, sowie für eine bessere Lesbarkeit dieser dienen sollen. Diese Konventionen werden in folgender Auflistung vorgestellt.

- Ein Serviceanbieter bezeichnet die serverseitige Implementierung eines Webservices. Dieser Webservice muss nicht zwangsläufig in JAX-WS implementiert sein, es ist auch jede andere Technologie möglich, die sich an den Webservice Standard hält. Gleiches gilt für den Servicekonsumenten. Dieser

bezeichnet allgemein die clientseitige Implementierung eines Webservices. Wenn die Implementierung auf JAX-WS ausgelegt ist, so wird diese durch den Zusatz JAX-WS (z.B. clientseitige Implementierung unter JAX-WS) gekennzeichnet. Auch werden die Begriffe JAX-WS Client und JAX-WS Server verwendet, welche als Synonyme der oben definierten Begriffe zu verstehen sind.

- Innerhalb dieser Diplomarbeit existieren eine Reihe von Listings. Diese Listings dienen der exemplarischen Behandlung der beschriebenen Technologien. Generell sind bei diesen Listings aus Gründen der Übersichtlichkeit nur die jeweils relevanten Codeausschnitte aus den entsprechenden Dateien dargestellt. Importe (bei Java Klassen) sowie alle Header und Namensraum-informationen (bei XML-Dokumenten) werden, sofern im entsprechenden Kontext nicht relevant, ebenfalls ausgeblendet.

- Um eine Unterscheidung zwischen den Java-Variablen und XML-Attributen sowie XML-Attributen und XML-Elemente aus unterschiedlichen Namensräumen zu fällen, werden in dieser Diplomarbeit alle XML-Elemente sowie XML-Attribute über einen vorangestellten Präfix einem Namensraum zugeordnet. Ist dieser Präfix nicht vorhanden, so kommt das Codewort aus der Java-Welt. Tabelle 2.1 liefert eine Übersicht über alle verwendeten Namensräume und deren Präfixe.

Präfix	Namensraum
pfs	http://pisys.de/pfs/
xs	http://www.w3.org/2001/XMLSchema
wsdl	http://schemas.xmlsoap.org/wsdl/
soap	http://schemas.xmlsoap.org/wsdl/soap/
jaxws	http://java.sun.com/xml/ns/jaxws
jaxb	http://java.sun.com/xml/ns/jaxb
wsu	http://docs.oasis-open.org/wss/2004/01/oasis-200401 -wss-wssecurity-utility-1.0.xsd
wsp	http://schemas.xmlsoap.org/ws/2004/09/policy

Tabelle 2.1: Verwendete XML Namensräume und deren Präfixe

2.6 Zusammenfassung

Dieses Kapitel stellte eine Einführung in die Thematik dieser Diplomarbeit dar. Neben einer einleitenden Beschreibung des Person Info Systems (PISys), wurde insbesondere auf den Aufbau dieser Diplomarbeit eingegangen, welcher maßgeblich von der Entwicklung des Person Finder Services (PFS), dem Webservice

innerhalb des Person Info Systems, beeinflusst wird.
Das nun folgende Kapitel behandelt alle wichtigen Grundlagen, auf denen die in dieser Diplomarbeit aufgeführten Technologien beruhen.

3 Grundlagen

Dieses Kapitel erläutert alle wichtigen Grundlagen, die für das Verständnis der in den folgenden Kapiteln vorgestellten Technologien wichtig sind. Da diese Grundlagen per Definition nicht an eine konkrete Technologie gebunden sind, werden diese in einer abstrakten Sichtweise dargestellt. Obwohl die konkrete Implementierung der vorgestellten Grundlagen erst in späteren Kapiteln erfolgt, finden sich bereits einige Hinweise auf Möglichkeiten wieder, wie diese in JAX-WS umgesetzt werden.

3.1 Serviceorientierte Architektur

Der Begriff *Serviceorientierte Architektur (SOA)* wurde erstmals im September 2000 von IBM eingeführt und beschreibt die Architektur einer losen Kopplung wiederverwendbarer Softwarebausteine (Services), die spezifische Aufgaben in einem bestimmten Geschäftsbereich (engl. Business Sector) übernehmen. Eine genaue und umfassende Definition von SOA lässt sich nur schwer finden, da SOA kein mathematisches/technisches Konzept ist. In seinem Buch beschreibt Josuttis eine SOA als:

> SOA is a paradigm. SOA is a way of thinking. SOA is a value system for architecture and design.[1]

Somit adressiert SOA eher die Managementebene eines Unternehmens, die sich mit IT-Konzepten weniger beschäftigt aber dennoch die Globalisierung ihrer IT-Struktur anstrebt. Aus diesem Grund stellen viele Artikel den Hype und die Vorteile um SOA in den Vordergrund und geben wenig Aufschluss über die Details der konkreten Implementierung. Dabei ist zu beachten, dass eine serviceorientierte Architektur nicht die bisherige IT-Landschaft eines Unternehmen ersetzen soll („from scratch“[2]), sondern sich vielmehr in diese eingliedern soll. Dies wird auch in einem Artikel über SOA von Liebhart deutlich:

> SOA ist die einzige Standardarchitektur, die explizit die Integration bestehender Systeme vorsieht.[3]

[1] S. [Jos07], Seite 2.

[2] Die weniger gebräuchliche deutsche Übersetzung für „from scratch“ lautet: „von Grund auf“ oder „von Anfang an“. Im Bereich Informatik hat sich jedoch der englischsprachige Begriff „from scratch“ durchgesetzt.

[3] S. [Lie07], Seite 3.

Nach Josuttis[4] besteht eine SOA aus den folgenden Elementen:

- *Services*, welche technologie- und plattformunabhängige Implementierungen eigenständiger Geschäftseinheiten darstellen, die eine in sich abgeschlossene Aufgabe übernehmen. Im Prinzip sind Services eine Erweiterung des Modulkonzepts, wie es schon seit Urzeiten der Informatik zum Einsatz kommt. Services können Teil (Arbeitsschritt) eines Geschäftsprozesses sein.

- *Einer spezifischen Infrastruktur*, die es erlaubt die Services in irgendeiner Form miteinander zu verbinden.

- *Regeln und Prozesse*, die eine heterogene Umgebung berücksichtigen und somit die Kombination der vorherigen Punkte ermöglichen. Auf diesen Punkt hat die Managementebene großen Einfluss.

Im Folgenden wird innerhalb dieser Diplomarbeit nicht mehr auf die SOA eingegangen, sondern vielmehr auf die konkrete Umsetzung einer SOA über Webservices. Letztere werden im folgenden Abschnitt beschrieben.

3.2 Webservice

Webservice war das IT-Modewort der letzten Jahre und führte zu einer Unmenge von Begrifflichkeiten, Definitionen und Technologien rund um das Thema der serviceorientierten Architektur (SOA). Gute Informationen zu diesem Thema waren lange Zeit nur spärlich vorhanden und blendeten die technischen Aspekte oftmals geschickt aus, um den allgemeinen Hype in den Vordergrund zu stellen. Dabei basieren Webservices auf dem einfachen Prinzip der elektronischen Übertragung von strukturierten Informationen. Was einen Webservice so besonders macht, ist allein die Art der Übertragung der Informationen und eng damit verbunden, sein Einsatzgebiet. Eine kurze und treffende Definition eines Webservices liefert Cerami mit:

> A web service is any service that is available over the Internet, uses a standardized XML messaging system, and is not tied to any one operating system or programming language.[5]

Historisch gesehen ist ein Webservice eine moderne Spezialisierung des *elektronischen Datenaustauschs (EDI)*, wobei der Begriff EDI dem Webservice weichen musste um den Bezug zu allen neuen Technologien der letzten Jahre wie SOAP, UDDI und WSDL herzustellen. Das Bedürfnis nach Webservices entstand aus der zunehmenden Globalisierung der Märkte (siehe 2.1) und der Zunahme des

[4] Vgl. [Jos07], Seite 2.
[5] S. [Cer02], Seite 6.

tertiären-Sektors (Dienstleistungen) im IT-Bereich. Der Wunsch vieler Unternehmen ist es, ihre Dienstleistungen in einer standardisierten und elektronischen Form weltweit anbieten zu können. Ein erster Schritt in diese Richtung war die Entwicklung des *World Wide Web (WWW)* von Tim Berners-Lee im Jahr 1989. Typischerweise wird das WWW von Unternehmen dazu genutzt, um einem Endkunden über eine Webseite (neben den Unternehmensdaten) bestimmte Dienste anzubieten. Daher handelt es sich hierbei um ein sog. *Business-To-Consumer (BTC)* Modell. In der Realität sind jedoch viele Unternehmen von anderen Unternehmen abhängig (z.B. Automobilhersteller an Zulieferer, Pauschalreiseanbieter an Fluganbieter, etc.) und möchten effizient auf deren Dienste zugreifen. Diese unternehmensübergreifende Abhängigkeit wird als *Business-To-Business (B2B)* Modell bezeichnet und ist der typische Einsatzort eines Webservices. Anstelle der manuellen Bedienung der Webseiten nur Nutzung der von einem Unternehmen angebotenen Dienste, regelt nun ein automatisierter Prozess über standardisierte Protokolle die Kommunikation der Unternehmen und nutzt deren angebotenen Dienste. Wie in Abbildung 3.1 dargestellt ist, findet diese Kommunikation

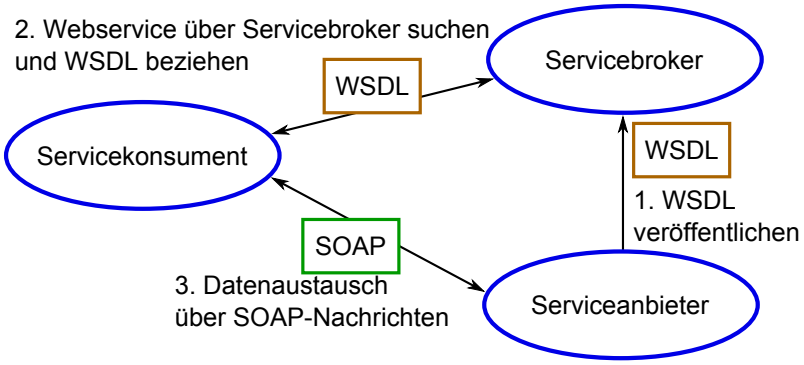

Abbildung 3.1: Typischer Aufbau eines Webservices
Quelle: Eigene Darstellung basierend auf der gebräuchlichen Konvention

in einem Webservice über verschiedene Akteure statt, die in den Formen Servicekonsument, Serviceanbieter und Servicebroker auftreten. Der grundlegende Ablauf der Kommunikation stellt sich folgendermaßen dar: Der Serviceanbieter veröffentlicht eine Beschreibung seines Services in einem global verfügbaren und bekannten Verzeichnis (bei einem Servicebroker). Möchte der Servicekonsument nun einen bestimmten Service konsumieren (nutzen), so sucht er diesen in diesem Verzeichnis. Hat er einen geeigneten Webservice gefunden, so findet eine dynamische Bindung zwischen Servicekonsument und Serviceanbieter statt. Der Servicekonsument kann nun synchron oder asynchron den bereitgestellten Service nutzen.
Diese Form der Kommunikation beruht auf mehreren XML-basierten Technologien, die sich in den vergangenen Jahren etabliert haben und mittlerweile zum

Webservice Standard gehören. Diese Technologien gliedern sich in verschiedene Ebenen des Webservice Protokoll Stacks und werden im Folgenden erläutert.

3.2.1 Auffindung

Die *Universal Description, Discovery and Integration (UDDI)* Spezifikation wurde im Jahr 2000 von Microsoft, IBM und Ariba veröffentlicht. Ziel war es, einen plattformunabhängigen und auf SOAP basierten Verzeichnisdienst zu schaffen, dessen primäres Ziel das Veröffentlichen und Finden von Webservices ist. Dabei ist der Verzeichnisdienst selber wieder ein Webservice.
Ein Serviceanbieter kann nun die Nutzung seines Webservices anbieten, indem er diesen in dem Verzeichnisdienst bereitstellt. Der Servicekonsument kann seinerseits auf den Verzeichnisdienst zugreifen, sich mit dem Serviceanbieter verbinden und den angebotenen Webservice nutzen. Um die Analogie mit einem typischen Telefonbuch herzustellen, werden die vom Serviceanbieter bereitgestellten Informationen in drei Kategorien unterteilt.

- *White Pages* – Generelle Informationen des Serviceanbieters, wie Name und Kontaktinformationen.
- *Yellow Pages* – Entspricht einem Branchenverzeichnis, wobei die Suche über spezielle Taxonomien (wie der Dienstart) erfolgt.
- *Green pages* – Technische Informationen zu dem angebotenen Webservice und seine Adresse.

Da die von IBM und Microsoft betriebene globale UDDI Business Registry abgeschaltet wurde, ist die UDDI nur noch von lokaler (etwa zur Lastenverteilung der Webservice-Aufrufe innerhalb eines Unternehmens/Konsortiums) Bedeutung und liegt somit außerhalb des Zielgebietes von SOA. Auf der anderen Seite sind in den noch betriebenen UDDIs überwiegend kleine Serviceanbieter zu finden, die zu Test- und Demonstrationszwecken erstellt wurden und somit die Qualität und den Nutzen einer UDDI in Frage stellen. Aufgrund dieser Tatsachen und der fehlenden Unterstützung durch JAX-WS wird auf die UDDI nicht weiter eingegangen.

3.2.2 Beschreibung

Damit ein Servicekonsument weiß, wie er mit einem in der UDDI eingetragenen Webservice kommunizieren kann, ist mindestens eine abstrakte Beschreibung des Interfaces vom angebotenen Webservice nötig. Diese Beschreibung trägt allgemein den Namen *Interface Definition Language (IDL)*. Ohne eine standardisierte IDL würde zumindest keine dynamische Bindung an einen Webservice möglich sein, was aber das Prinzip eines Webservices verletzen würde.

Als IDL hat sich im Bereich Webservices die *Web Services Description Language (WSDL)* durchgesetzt. Diese entstand aus der Kombination der *Service Description Language (SDL)* von Microsoft sowie der *Network Application Service Specification Language (NASSL)* von IBM und wurde im September 2000 umgesetzt. Die WSDL Spezifikation beschreibt die WSDL als:

> WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.[6]

Die WSDL ist also ein auf XML basierendes Format, welches der Beschreibung eines Webservices dient. Dabei ist ein Webservice aus der Sicht der WSDL eine Menge von Webservice-Endpunkten, die wiederum eine Ansammlung von konkreten Operationen sind. Ferner definiert die WSDL das Protokoll- und Nachrichtenformat des Webservice-Endpunktes.
Auf der anderen Seite zeigt die WSDL Spezifikation natürlich auch das auf, was in einer konkreten Implementierung eines Webservices alles realisierbar sein muss. Daher wird in Kapitel 4 relativ detailliert auf die WSDL eingegangen und gezeigt, inwiefern die Feinheiten der WSDL Spezifikation in JAX-WS umgesetzt werden können.

3.2.3 XML-Nachrichtenaustausch

Die eigentliche Kommunikation zwischen Serviceanbieter und Servicekonsument basiert auf standardisierten XML-Nachrichten. Diese Wahl fiel relativ leicht, da XML den Vorteil der Plattformunabhängigkeit und der hohen Verbreitung (in Form von Tools und als ein Standardprotokoll im Internet) bietet. Dennoch gibt es unterschiedliche Kommunikationsmodelle für Webservices, die jeweils in einer anderen Struktur der XML-Nachrichten resultieren. Da JAX-WS von Haus aus alle Kommunikationsmodelle unterstützt, werden diese im Folgenden vorgestellt.

XML-RPC

Das *XML-Remote Procedure Call (XML-RPC)* ermöglicht das Aufrufen von Funktionen oder Methoden auf entfernten Rechnern mit XML-Nachrichten über HTTP. XML-RPC bedient sich dabei der gewöhnlichen HTTP-Request-Methoden. So werden Anfragen an einen Serviceanbieter über HTTP POST Methoden geschickt, wobei die Antworten in dem Body der entsprechenden HTTP-Antwort zum Servicekonsumenten zurück geliefert werden. Für die Übertragung der Parameter beschränkt man sich auf einige wenige Datentypen, die in jeder gängigen Programmiersprache abgebildet werden können. Benutzerdefinierte Datentypen gibt es nicht. Aufgrund der geringen Komplexität der XML-RPC Spezifikation

[6] S. [WSDL1.1].

und der damit verbundenen schnellen Einarbeitung und leichten Implementierung wird XML-RPC noch oft eingesetzt.[7] Wesentliche Nachteile von XML-RPC liegen vor allem in der Beschränkung auf HTTP als einziges Übertragungsprotokoll und RPC als einziges Kommunikationsmodell, sowie in der nicht möglichen Typisierung. Auch fehlt die Möglichkeit, den angebotenen Webservice zu beschreiben (WSDL gibt es nur in Verbindung mit SOAP), so dass eine dynamische Bindung an einen vorher unbekannten Webservice unmöglich ist. Aufgrund dieser Nachteile und da XML-RPC vollständig durch SOAP realisiert werden kann, wird auf XML-RPC in dieser Diplomarbeit nicht weiter eingegangen. Allein das Fehlen einer IDL ist nach der üblichen Webservice-Konvention ein Ausschlusskriterium dieser Technologie.

REST

Unter dem *Representational State Transfer (REST)* versteht man eine Softwarearchitektur dessen Kernpunkt die Minimierung der Schnittstellen zwischen kommunizierenden Systemen ist. So definiert REST für die Kommunikation mit anderen Systemen nur die Befehle Create, Read, Update und Delete. Typischerweise wird die REST-Architektur mit HTTP umgesetzt (REST entstand aus der Analyse von HTTP), wo die REST-Befehle entsprechend auf die HTTP-Methoden PUT, GET, POST und DELETE gemappt werden. Da REST auf HTTP basiert, ist REST komplett zustandslos. Jede Anfrage an den Serviceanbieter muss also alle benötigten Informationen für die Verarbeitung dieser Anfrage beinhalten, es kann von keinem bekanntem Kontext ausgegangen werden. Zustandsinformationen können jedoch außerhalb von REST (z.B. in Cookies, SSL Session IDs oder durch URL Rewriting) gespeichert werden. Bei der Realisierung von Webservices mit der REST Architektur spricht man auch oft von RESTful-Webservices, welche den Vorteil der Einfachheit gegenüber SOAP-basierten Webservices genießen. JAX-WS bietet native Unterstützung für die REST-Architektur an. Dieser Ansatz bietet sich vor allem dann an, wenn der Entwickler direkten Zugriff auf die XML-Nachrichten haben will und einen schlanken Webservice (ohne den für SOAP typischen Overhead) realisieren will. Dennoch wird REST, auch aufgrund der fehlenden IDL, in dieser Diplomarbeit nicht weiter behandelt.

SOAP

Wenn man von Webservices spricht, so fällt meist auch das Wort *SOAP*[8]. SOAP ist die direkte Weiterentwicklung von XML-RPC und kann seinen Vorgänger vollständig ersetzen, bietet aber noch weitaus mehr. So ist es insbesondere möglich, auch dokumentenorientierte Nachrichten zu verschicken und eigene Datentypen

[7] Vgl. [Win99].

[8] SOAP ist seit der Version 1.2 ein eigenständiger Begriff. Früher war SOAP die Abkürzung für *Simple Object Access Protocol* bzw. *Service Oriented Architecture Protocol*.

zu definieren. Überdies ist SOAP nicht an HTTP als einziges Übertragungsprotokoll gebunden und es können Fehlermeldungen in Form von SOAP-Fault Elementen verschickt werden. Ein weiterer wesentlicher Vorteil von SOAP ist die dynamische Bindung an einen Webservice über eine definierte IDL, der WSDL. In Abbildung 3.2 ist der interne Aufbau einer SOAP-Nachricht abgebildet. Diese

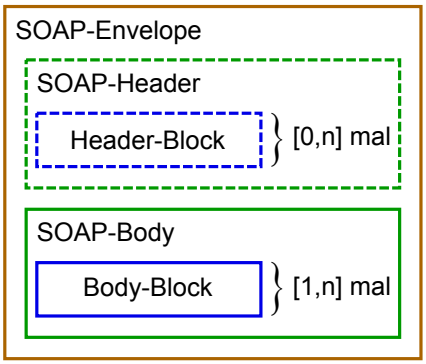

Abbildung 3.2: Aufbau einer SOAP-Nachricht
Quelle: Eigene Darstellung basierend auf der gebräuchlichen Konvention

besteht demnach aus einem SOAP-Envelope, welcher einen optionalen SOAP-Header (für Metainformationen wie die verwendete „Quality of Service“, Verschlüsselung und Routing) sowie einen SOAP-Body enthält. Letzteres enthält die eigentlichen Nutzdaten der übertragenden SOAP-Nachricht und variiert maßgeblich nach dem Style der verwendeten WSDL. So ist es insbesondere möglich, dass der SOAP-Body mehrere Kindelemente enthält (entspricht dem Wrapped-Style der WSDL), oder dass das einzige Kindelement eine Instanz des XML-Schemas aus der `<wsdl:types>` Sektion der WSDL ist (entspricht dem Document-Style der WSDL). Dieser Zusammenhang zwischen der WSDL und den daraus resultierenden SOAP-Nachrichten wird in Kapitel 4 vertieft.

3.3 Zusammenfassung und Fazit

Dieses Kapitel hat alle notwendigen Grundlagen erläutert, die für das Verständnis der nun folgenden Kapitel wichtig sind. Auf die UDDI und SOA wird in der Diplomarbeit nicht weiter eingegangen. Die Relevanz der UDDI ist derzeit fraglich und SOA wird im Kontext der Webservices abgehandelt. An dieser Stelle stellt sich die Frage, wo der eigentliche Unterschied zwischen Webservices und SOA liegt. Viele Artikel sehen den Unterschied in der modernen Definition eines Webservices, welcher auf standardisierten XML-Nachrichten basiert. Nach dieser Auslegung stellen Webservices nur einen Teilbereich dessen dar, was eine serviceorientierte Architektur ausmacht und legen sich auf eine konkrete XML-lastige Implementierung fest. Fasst man den Begriff Webservice weitreichender,

so lässt sich folgende Aussage treffen: „SOA ist das, was das Management will und Webservice ist das, was die IT dann schließlich umsetzt“. Auf der anderen Seite beschränkt sich SOA nur auf die Integration bestehender Systeme. Ein Webservice kann jedoch auch „from scratch“ realisiert werden.
Wichtig sind in diesem Kapitel vor allem die Unterschiede in den verschiedenen Kommunikationsmodellen XML-RPC, REST und SOAP. Da JAX-WS von Haus aus Unterstützung für alle diese Kommunikationsmodelle anbietet, werden diese in Tabelle 3.1 anhand wichtiger Kriterien miteinander verglichen. Hierbei fällt

	XML-RPC	REST	SOAP
HTTP Unterstützung	√	√	√
Andere Protokolle			√
Zustandsinformationen speichern	√		√
Quality of Service			√
IDL			√
Eigene Datentypen			√
Ausnahmebehandlung			√

Tabelle 3.1: Vergleich der Kommunikationsmodelle

insbesondere auf, dass SOAP alle aufgeführten Kriterien unterstützt. Dies erklärt die Komplexität von SOAP, welche zugleich der größte Nachteil dieses Protokolls darstellt. Dennoch, wenn XML-RPC oder REST alle die von SOAP unterstützten Eigenschaften besitzen würden, dann wären diese Kommunikationsmodelle in der Komplexität sicherlich mit der des SOAP Protokolls vergleichbar.

4 Webservice Description Language

In der abstrakt gehaltenen Einleitung zur WSDL in Kapitel 3.2.2 wurde insbesondere auf die Wichtigkeit einer standardisierten IDL (Interface Definition Language) in Bezug auf SOA eingegangen. Diese IDL stellt nach dem heutigen Stand die *Webservice Description Language (WSDL)* Version 1.1[1] dar, deren Aufbau und Eigenschaften in diesem Kapitel erläutert werden.[2]
In Bezug auf den Kern von JAX-WS, dem Fokus dieser Diplomarbeit, scheint dieses Kapitel in direkter Sichtweise eher überflüssig zu sein, ist es aber nicht. Zum einen grenzt die WSDL die durch einen Webservice realisierbaren Möglichkeiten ab. Dadurch stellt die WSDL die Anforderungen auf, welche durch JAX-WS umgesetzt werden müssen. Zum anderen spezifiziert JAX-WS einen WSDL zu Java Generator, welcher die WSDL auf Java-Artefakte (Klassen und Interfaces) abbildet. Die Bedeutung der WSDL wird in der JAX-WS Spezifikation mit

> WSDL can be considered the de-facto service description language for XML Web Services.[3]

hervorgehoben. Zur Veranschaulichung dieses Kapitels dient die WSDL des Person Finder Services. Diese wird im Folgenden mit PFS-WSDL abgekürzt und ist in Abschnitt 4.2 dargestellt. Zunächst wird allerdings der allgemeine Aufbau einer WSDL abgehandelt.

4.1 Aufbau

Wie bereits geschildert, dient die WSDL der Beschreibung eines Webservices. Möchte ein Servicekonsument diesen Webservice nutzen, so müssen drei Fragestellungen geklärt werden:

- *Was* für Webservice-Endpunkte mitsamt Operationen werden von dem Webservice angeboten?
- *Wo* sind diese Webservice-Endpunkte zu finden?

[1] Vgl. [WSDL1.1].
[2] JAX-WS bietet zwar auch Unterstützung für die Version 2.0 der WSDL an, letztere ist jedoch vom W3C noch nicht freigegeben.
[3] S. [JSR224], Seite 1.

- *Wie* sind die Operationen dieser Webservice-Endpunkte realisiert (betrifft maßgeblich den Style der WSDL)?

Diese „was“, „wo“ und „wie“ Fragen werden in einer WSDL durch einzelne Elemente definiert, welche in Abbildung 4.1 aufgeführt sind. Diese Elemente stehen

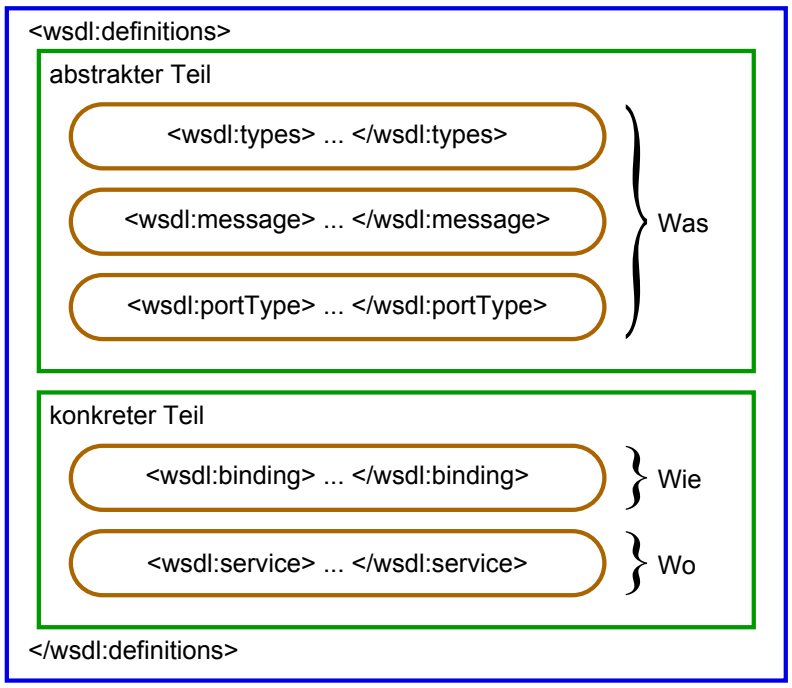

Abbildung 4.1: Aufbau einer WSDL
Quelle: Eigene Darstellung basierend auf [MTSM03], Figure 5.1

zueinander in Beziehung und lassen sich wiederum einem abstrakten Teil und einem konkreten Teil zuordnen. Der abstrakte Teil kann von einer anderen WSDL wiederverwendet werden und sollte weitestgehend (wenn auch nur unternehmensweit) standardisierte Webservice-Endpunkte mitsamt Datentypen und Operationen beinhalten. Damit diese nicht in jeder WSDL erneut definiert werden müssen, kann der entsprechende Namensraum über das `<wsdl:import>` Element eingebunden und benutzt werden. Im Gegensatz hierzu bezieht sich der konkrete Teil auf nur einen bestimmten Webservice.
Im Folgenden werden die WSDL Elemente aus Abbildung 4.1 erläutert, welche als „six major elements“ in der WSDL Spezifikation definiert sind.[4] Alle WSDL-Elemente sind direkte Kindelemente des Wurzelelements `<wsdl:definitions>`.

- `<wsdl:types>` – ein Containerelement für Datentypdefinitionen. Üblicherweise werden diese Datentypdefinitionen als globale Elemente in einen oder

[4] S. [WSDL1.1], Kapitel 2.1.

mehreren XML-Schemas innerhalb des `<wsdl:types>` Elements abgelegt. Die WSDL definiert hierbei also keine eigenen Datentypen, sondern benutzt diese aus der XML-Schema Spezifikation. Um erweiterbar zu bleiben erlaubt die WSDL Spezifikation dennoch ausdrücklich die Benutzung anderer Sprachen zur Definition der Datentypen.

- `<wsdl:message>` – eine abstrakte Definition des Inputs, Outputs oder Faults (Fehlertyps) einer Operation. Da das `<wsdl:message>` Element auch mehrere Datentypen umfassen kann, besteht dieses weiterhin aus einem oder mehreren logischen `<wsdl:part>` Elementen. Diese referenzieren jeweils einen Typ aus der `<wsdl:types>` Sektion über ein bestimmtes Attribut. In Bezug auf XML-Schemas können die Attribute `element` oder `type` benutzt werden um entweder ein `<xs:element>` oder ein `<xs:simpleType>`/`<xs:complexType>` zu referenzieren.

- `<wsdl:portType>` – eine abstrakte Definition eines Webservice-Endpunktes. Das `<wsdl:portType>` Element umfasst ein oder mehrere `<wsdl:operation>` Elemente, welche jeweils eine abstrakte Definition einer vom Webservice angebotenen Operation darstellen. Diese Operation kann je nach Interaktionstyp (One-Way, Request-Response, Solicit-Response und Notification) die Elemente `<wsdl:input>`, `<wsdl:output>` und `<wsdl:fault>` besitzen, oder nicht. Beispielsweise besitzt der One-Way Interaktionstyp nur das `<wsdl:input>` Element, wohingegen der Request-Response Interaktionstyp alle der oben angegebenen Elemente in genau dieser Reihenfolge besitzt. Der Solicit-Response Interaktionstyp wird von JAX-WS nicht unterstützt. Zudem müssen ab der Version 2.0 der WSDL alle Operationen über einen eindeutigen Namen verfügen. Daraus folgt, dass es keine überladenen Operationen mehr gibt, was natürlich eine Einschränkung seitens der Implementierung in JAX-WS bedeutet.

- `<wsdl:binding>` – bezieht sich auf einen bestimmten `<wsdl:portType>` (`type` Attribut) und bindet dessen Operationen bzw. Nachrichten an ein bestimmtes Nachrichtenformat und ein bestimmtes Protokoll. Das Binden ist hierbei erweiterbar und dessen Struktur ist protokollabhängig. In der PFS-WSDL wird z.B. der Input der Operation `<pfs:getPerson>` an das SOAP Protokoll (man beachte den `soap` Präfix, welcher den Namensraum der SOAP Erweiterung der WSDL definiert) gebunden. An dieser Stelle ist es sogar möglich, für den Input und den Output der gleichen Operation verschiedene Protokolle zu wählen.

- `<wsdl:port>` – Kindelement des `<wsdl:service>` Elements. Verbindet ein bestimmtes Binding mit einer konkreten Adresse um einen konkreten Webservice-Endpunkt innerhalb eines Webservices zu definieren.

- `<wsdl:service>` – definiert einen konkreten Webservice, indem ein oder mehrere `<wsdl:port>` Elemente aggregiert werden.

Bezüglich der Fragen „was“, „wo“ und „wie“ realisieren die Elemente `<wsdl:types>`, `<wsdl:message>`, und `<wsdl:portType>` also das „was“, `<wsdl:binding>` das „wie“ und `<wsdl:service>` das „wo“.

4.2 WSDL Binding und Styles

Wie in vorherigem Abschnitt bereits angedeutet, kann die WSDL an einigen Stellen (s. `<!-- extensibility element -->` in der WSDL Spezifikation) erweitert werden. Diese Erweiterungen dienen z.B. dazu, eine neue Datentypdefinition in einer von XML-Schema verschiedenen Sprache in der `<wsdl:types>` Sektion der WSDL einzupflegen. Extensiver Nutzen der erweiterbaren Elemente wird dort benötigt, wo die WSDL an ein spezifisches Protokoll und Nachrichtenformat gebunden wird. Also in der `<wsdl:binding>` Sektion der WSDL. Hier gibt es in der WSDL Spezifikation bereits drei vordefinierte Bindings. Diese sind das SOAP Binding, das HTTP GET & POST Binding und das MIME Binding. Da diese Bindings nicht zum WSDL-Kern gehören, besitzen sie jeweils einen eigenen Namensraum.
So erweiterbar die WSDL auch ist, auf der Java-Seite wird das ganze wieder etwas eingeschränkt. So erlaubt JAX-WS von Haus aus nicht das Einpflegen von anderen Erweiterungen. Begründet wird dies mit einer eingeschränkten Interoperabilität und eingeschränkter Portabilität der resultierenden Anwendung. Also gibt es in Bezug auf die Binding Erweiterung der WSDL nur native Unterstützung für die SOAP- und MIME- Bindings. Es gibt keine einheitliche Schnittstelle für benutzerdefinierte Bindings. Sollte sich ein anderes Binding etablieren, so wird dieses in einer späteren Version von JAX-WS implementiert werden. Dieser Fall scheint zumindest in naher Zukunft jedoch eher unrealistisch, wenn man die marktbeherrschende Präsenz von SOAP berücksichtigt.
Gerade der letzte Punkt, die marktbeherrschende Präsenz von SOAP, ist ausschlaggebend, sich das SOAP-Binding genauer anzuschauen und auf den WSDL/SOAP-Style[5] genauer einzugehen. Dieser beeinflusst maßgeblich den Aufbau des SOAP-Bodies und wird in JAX-WS über die `@SOAPBinding` Annotation realisiert. An dieser Stelle ist anzumerken, dass einige Bücher den WSDL/SOAP-Style nur auf die SOAP-Binding Erweiterung der WSDL (hier die Attribute `style` und `use`) beziehen. Dies ist jedoch nur die halbe Wahrheit, denn der WSDL/SOAP-Style ist eine Kombination des Aufbaus der WSDL (speziell der `<wsdl:type>` und `<wsdl:message>` Elemente) und der zuvor genannten SOAP-Binding Erweiterung. Generell ist der WSDL/SOAP-Style über die drei Eigenschaften Binding-Style,

[5] Vgl. [But03].

Use und Parameter-Style definiert. Wie bereits angedeutet resultieren diese Eigenschaften aus einer Kombination des Aufbaus der WSDL und den `style` und `use` Attributen der SOAP-Binding Erweiterung. Im Folgenden werden die verschiedenen WSDL/SOAP-Styles aufgeführt und beschrieben, wie diese den Aufbau des SOAP-Bodies beinflussen. Als exemplarisches Beispiel einer WSDL dient die PFS-WSDL, welche den PISys-Server aus Kapitel 2.3 beschreibt. Diese ist in Listing 4.1 dargestellt und zeichnet sich durch einen Binding-Style von Document, einen Use von Literal und einen Parameter-Style von Wrapped aus.

Listing 4.1: pfs.wsdl

```
<wsdl:definitions>
    <wsdl:types>
        <xs:schema>
            <xs:element name="getPerson">
                <xs:simpleType>
                    <xs:restriction base="xs:string">
                    <xs:pattern value=".{1,12}"/>
                    </xs:restriction>
                </xs:simpleType>
            </xs:element>
            <xs:element name="getPersonResponse">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="postleitzahl"
                           type="xs:int"/>
                        <xs:element name="interesse"
                           type="xs:string"  maxOccurs="10"
                           minOccurs="0"/>
                        <xs:choice>
                            <xs:element name="alter"
                               type="xs:int"/>
                            <xs:element name="jahrgang"
                               type="xs:date"/>
                        </xs:choice>
                        <xs:element name="familienstand">
                            <xs:simpleType>
                                <xs:restriction
                                   base="xs:string">
                                    <xs:enumeration
                                       value="ledig"/>
                                    <xs:enumeration
                                       value="verheiratet"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:element>
```

```
28                      </xs:sequence>
29                      <xs:attribute name="gueltig"
                           type="xs:boolean"/>
30                  </xs:complexType>
31              </xs:element>
32          </xs:schema>
33      </wsdl:types>
34      <wsdl:message name="getPersonRequestMessage">
35          <wsdl:part name="part1" element="pfs:getPerson"/>
36      </wsdl:message>
37      <wsdl:message name="getPersonResponseMessage">
38          <wsdl:part name="part1"
               element="pfs:getPersonResponse"/>
39      </wsdl:message>
40      <wsdl:portType name="PersonFinderPort">
41          <wsdl:operation name="getPerson">
42              <wsdl:input name="input1"
                   message="pfs:getPersonRequestMessage"/>
43              <wsdl:output name="output1"
                   message="pfs:getPersonResponseMessage"/>
44          </wsdl:operation>
45      </wsdl:portType>
46      <wsdl:binding name="PersonFinderSOAPBinding"
           type="pfs:PersonFinderPort">
47          <soap:binding style="document"
               transport="http://schemas.xmlsoap.org/soap/http"/>
48          <wsdl:operation name="getPerson">
49              <soap:operation/>
50              <wsdl:input name="input1">
51                  <soap:body use="literal"/>
52              </wsdl:input>
53              <wsdl:output name="output1">
54                  <soap:body use="literal"/>
55              </wsdl:output>
56          </wsdl:operation>
57      </wsdl:binding>
58      <wsdl:service name="PersonFinderService">
59          <wsdl:port name="PersonFinderPort"
               binding="pfs:PersonFinderSOAPBinding">
60              <soap:address location="http://localhost:8080/
                   PersonInfoSystem/PersonFinderService"/>
61          </wsdl:port>
62      </wsdl:service>
63  </wsdl:definitions>
```

In späteren Kapiteln wird gezeigt, das JAX-WS anhand der PFS-WSDL alle für die Veröffentlichung eines Webservices benötigten Artefakte generieren kann. Dadurch ist gewährleistet, dass zur Laufzeit alle ein- und ausgehenden XML-Nachrichten mit dem WSDL/SOAP-Style sowie den Datentypdefinitionen der WSDL konform sind. Bei der ebenfalls von JAX-WS angebotenen manuellen Implementierung muss sich der Entwickler selbst um diese Aufgabe kümmern. Listing 4.2 illustriert eine typische SOAP-Nachricht, wie sie vom PISys-Server verschickt wird.

Listing 4.2: getPersonResponse.xml

```
1  <S:Envelope
       xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
2      <S:Body>
3          <getPersonResponse xmlns="http://pisys.de/pfs/"
               gueltig="true">
4              <postleitzahl>81249</postleitzahl>
5              <interesse>Fussball</interesse>
6              <interesse>Computer spielen</interesse>
7              <jahrgang>1981-09-24Z</jahrgang>
8              <familienstand>ledig</familienstand>
9          </getPersonResponse>
10     </S:Body>
11 </S:Envelope>
```

Anhand dieses Listings und der PFS-WSDL aus Listing 4.1 wird in folgender Auflistung eine Beschreibung der verschiedenen WSDL/SOAP-Styles vorgenommen.

- *Binding-Style* (RPC oder Document) – Bei einem Binding-Style von RPC wird die SOAP-Nachricht in einer RPC konformen Form verschickt. Dies bedeutet, dass der Fokus auf der eigentlichen Operation mit ihren Parametern liegt. In einer SOAP-Nachricht existiert also für jede Operation ein Element unterhalb des SOAP-Bodies, wobei dessen direkte Kindelemente die Parameter der Operation darstellen. Für den Aufbau der WSDL ergibt sich dadurch folgende Implikation. Das `<wsdl:message>` Element darf mehrere `<wsdl:part>` Elemente besitzen, wobei jedoch deren `type` Attribut verwendet werden muss um die Datentypen zu referenzieren. Zusätzlich muss das `style` Attribut der SOAP Binding Erweiterung auf `rpc` gesetzt werden.
  Bei einem Binding-Style von Document werden keine Operationen aus den verschiedenen WSDL-Elementen zusammengebastelt, sondern die Kindelemente vom SOAP-Body entsprechen eins zu eins den referenzierten Elementen aus der `<wsdl:types>` Sektion der WSDL. Dadurch lassen sich die SOAP-Nachrichten zwar schlechter lesen, können aber anhand den darin enthaltenen Datentypdefinitionen validiert werden. Anders als bei RPC

wird in diesem Fall das `element` Attribut der `<wsdl:part>` Elemente verwendet um Elemente zu referenzieren. Anschließend wird das `style` Attribut der SOAP Binding Erweiterung auf den Wert `document` gesetzt.

- *Use* (Encoded oder Literal) – Der Use regelt die Kodierung der SOAP-Nachricht und ist die einzige der Style-Eigenschaften, welche vollständig über die SOAP-Binding Erweiterung angegeben wird. Bei einem Use von Literal wird die Serialisierung anhand der Datentypdefinitionen der WSDL vorgenommen. Für den Use von Encoded spezifiziert SOAP bestimmte Regeln, anhand derer die Serialisierung vorgenommen wird.

- *Parameter-Style* (Wrapped oder Unwrapped) – Der Parameter-Style bestimmt darüber, ob der SOAP-Body ein (Wrapped) oder mehrere (Unwrapped) Kindelemente besitzt. Diese Eigenschaft resultiert vollständig aus dem Aufbau der WSDL. Ist der Binding-Style Document, so kommt es auf die Anzahl der `<wsdl:part>` Elemente an. Bei RPC gibt es kein Unwrapped, da das Wrapper Element immer aus dem `name` Attribut der `<wsdl:operation>` generiert wird.

4.3 Zusammenfassung und Fazit

Dieses Kapitel ist sehr detailliert auf die Eigenschaften der WSDL eingegangen. Dabei wurden insbesondere die Unterschiede der einzelnen WSDL/SOAP-Style Eigenschaften hervorgehoben und deren Komplexität verdeutlicht. Für die PFS-WSDL wird der Document/Literal/Wrapped Style verwendet, welche auch gleichzeitig der empfohlene Style ist. In Bezug auf JAX-WS wird der Entwickler nur bei der manuellen Umsetzung eines JAX-WS Clients mit den unterschiedlichen WSDL/SOAP-Styles konfrontiert. Bei einer generierten Implementierung, bei der alle benötigten Java-Artifakte aus der WSDL erstellt werden, kann sich der Entwickler auf die korrekte Abbildung dieses Styles verlassen, welcher sich in Form von Annotationen in den entsprechenden Klassen manifestiert. Die von der JAX-WS Laufzeitumgebung erstellen SOAP-Nachrichten sind dadurch automatisch mit dem WSDL/SOAP-Style der WSDL konform. Dies ist ein großer Vorteil in JAX-WS und erleichtert erheblich den Umgang mit einer fremden WSDL.

5 Webservices mit JAX-RPC

In dem letzten Kapitel wurde gezeigt, wie die WSDL einen Webservice beschreibt. Hauptsächliches Augenmerk lag dabei auf der Verwendung der WSDL in Kombination mit SOAP. Berücksichtigt man nun die Komplexität dieses Gespanns (WSDL und SOAP), so führt dies zu einem immensen Programmieraufwand bei der manuellen Umsetzung eines Webservices über die bisherigen Möglichkeiten der Java API. Generell müsste sich der Entwickler um den Un-/Marhalling-Prozess der XML-Nachrichten sowie der Realisierung der gesamten Kommunikation selbst kümmern. Dies war der Anlass für die Entwicklung der *Java API for XML-Based RPC (JAX-RPC)*. JAX-RPC war der erste Versuch das Thema Webservices in die Java API Spezifikation zu bringen und wurde im Juni 2003 als JSR 101 veröffentlicht. Wie der Name schon vermuten lässt, ist JAX-RPC auf das Verschicken von SOAP-Nachrichten mit einem Binding-Style von RPC ausgelegt.
Inhalt dieses Kapitels ist die beispielhafte Implementierung eines JAX-RPC Webservices (Server- sowie Clientseite) anhand der WSDL des Person Finder Services, der PFS-WSDL. Mit anderen Worten wird ein Teil des in JAX-WS realisierten Person Finder Services in JAX-RPC nachimplementiert. Hierfür wird die Version 1.1.3 der *JAX-RPC Standard Implementation (JAX-RPC SI)* verwendet. Dabei werden insbesondere die Defizite dieser Implementierung herausgearbeitet, da diese die Motivation für die in späteren Kapiteln vorgestellte JAX-WS Technologie liefern.

5.1 JAX-RPC Server

In diesem Abschnitt wird die Entwicklung eines JAX-RPC Servers anhand der PFS-WSDL aus Listing 4.1 aufgezeigt. Wie in Abbildung 5.1 zu erkennen ist, werden hierfür die beiden von der JAX-RPC RI bereitgestellten Tools wscompile und wsdeploy verwendet.[1] Ausgehend von diesen beiden Tools werden in folgender Auslistung die einzelnen Schritte erläutert, welche zur Realisierung eines JAX-WS Servers notwendig sind.

1. Eine Webservice Implementierung in JAX-RPC (egal ob Client oder Server) erfolgt immer entweder über eine WSDL oder über ein Service Endpoint

[1] Die Tools wscompile und wsdeploy sind eine Weiterentwicklung des Tools xrpcc, welches in früheren JAX-RPC RI Versionen zum Einsatz gekommen ist.

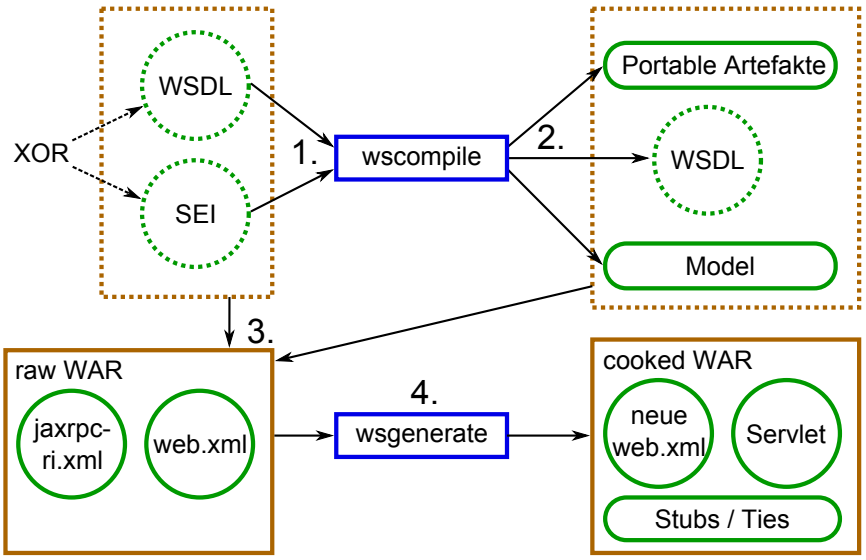

Abbildung 5.1: Erstellung eines Webservices mit der JAX-RPC SI
Quelle: Eigene Darstellung

Interface (s. Kapitel 7.1.1). Da in diesem Fall bereits eine WSDL in Form der PFS-WSDL vorliegt, wird das Tool wscompile mit der Option `-import` zum Einlesen dieser verwendet. Als Argument wird hierbei die Konfigurationsdatei aus Listing 5.1 übergeben.

Listing 5.1: config.xml

```
1 <configuration
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
2     <wsdl location="pis.wsdl"
        packageName="de.pisys.pfs" />
3 </configuration>
```

2. Anhand dieser Konfigurationsdatei werden alle für die Implementierung eines JAX-RPC Servers benötigten Java-Artefakte aus der PFS-WSDL generiert. Benötigte Artefakte sind vor allem das von `Remote` abgeleitete Service Endpoint Interface und eine Musterimplementierung dieses Interfaces, welches die konkrete Logik des Webservice-Endpunktes realisiert. Die Generierung der Java-Artefakte ist in JAX-RPC leider unvollständig und fehlerhaft. So können z.B. keine `<xs:choice>` Gruppen abgebildet werden. JAX-RPC wirft in diesem Fall einfach eine Fehlermeldung und beendet die Ausführung. Bei Enumerationen, wie dem Element `<pfs:familienstand>` in der PFS-WSDL, erzeugt wscompile zwar die Klasse `Familienstand`, welche die Werte `ledig` und `verheiratet` speichern kann, diese Klasse entspricht jedoch

nicht der JAX-RPC Definition eines Value-Types[2]. So fehlt in dieser Klasse ein öffentlicher und leerer Konstruktor. Fügt man diesen nachträglich ein, so erhält man immer noch eine Fehlermeldung namens „Ungültiger Typ für JAX-RPC-Struktur“. Diese Probleme führen dazu, dass die PFS-WSDL zunächst überarbeitet werden muss, damit wscompile funktionierende Java-Artefakte erzeugen kann.

3. Zusätzlich zu den Java-Artefakten wird für die Erzeugung einer WAR-Datei noch eine Datei namens jaxrpc-ri.xml benötigt. Diese Datei enthält Informationen über die konkrete Adresse des Webservice-Endpunktes, den Ort der WSDL und das verwendete Service Endpunkt Interface mitsamt Implementierung. Listing 5.2 stellt den Inhalt dieser Datei dar.

Listing 5.2: jaxrpc-ri.xml

```
<webServices
    xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/dd"
    version="1.0"
    targetNamespaceBase="http://pisys.de/pfs/"
    typeNamespaceBase="http://pisys.de/pfs/"
    urlPatternBase="/ws">
    <endpoint
        name="PersonFinderPort"
        displayName="PersonFinderPort"
        description="PersonFinderPort WebService"
        wsdl="/WEB-INF/xml/pfs.wsdl"
        interface="de.pisys.pfs.PersonFinderPort"
        implementation="de.pisys.pfs.
            PersonFinderPort_Impl"/>
    <endpointMapping
        endpointName="PersonFinderPort"
        urlPattern="/PersonFinderPort"/>
</webServices>
```

Generiert man nun aus den Java-Artefakten und der Datei jaxrpc-ri.xml ein WAR-Archiv, so kann dieses noch nicht auf einem Application Server bereitgestellt werden, da dieses weder ein Servlet noch eine EJB enthält. Es handelt sich hierbei um ein so genanntes „raw“ WAR-Archiv.

4. Da in diesem Fall der JAX-RPC Server als Servlet implementiert wird, ist es die Aufgabe des wsdeploy Tools einen Servlet in die web.xml einzutragen und Ties[3] für die Kommunikation mit einem Servicekonsumenten zu

[2] Vgl. [JAX-RPC-Types], Abschnitt Value-Types.

[3] Eine Tie-Komponente bezeichnet ein entferntes Objekt, welches wie ein Skeleton serverseitig implementiert ist, aber mit dem Client über das IIOP-Protokoll kommuniziert.

generieren. Als Servlet fungiert die Klasse `JAXRPCServlet`, welche Aufrufe automatisch an die generierte Tie-Komponente delegiert. Dieser Prozess wird von dem wsdeploy Tool glücklicherweise automatisch vollzogen, wobei als Resultat ein fertiges WAR-Archiv (cooked WAR) entsteht, welches auf einem Application Server bereitgestellt werden kann.

Obwohl die Tools wscompile und wsdeploy zusammen mit der JAX-RPC Runtime dem Entwickler viele Aufgaben bei der Entwicklung eines JAX-RPC Servers abnehmen, müssen dennoch viele Dinge manuell erledigt werden. Dazu gehört das Schreiben der Konfigurationsdateien config.xml und jaxrpc-ri.xml sowie das umständliche zweifache Erzeugen des WAR-Archivs. Größter Nachteil von JAX-RPC sind jedoch die stark eingeschränkten XML/Java Binding-Fähigkeiten. So kann man sich, wie im Beispiel der PFS-WSDL aus diesem Kapitel, nicht darauf verlassen, dass aus der WSDL funktionierende Java-Klassen generiert werden. Dies führt zu einem manuellen Vorbereiten der WSDL, was gerade bei einer größeren WSDL nicht immer tragbar ist und außerdem den Sinn eines Tools wie wscompile in Frage stellt.

5.2 JAX-RPC Client

Im Gegensatz zum vorigen Abschnitt wird im Folgenden die Entwicklung eines JAX-RPC Clients anhand der PFS-WSDL aufgezeigt. Wieder wird für die Erstellung der benötigten Java-Artefakte das Tool wscompile verwendet, diesmal aber mit der Option `-gen:client`. Die dabei verwendete Konfigurationsdatei ist, bis auf den Ort der WSDL, ähnlich zu der aus Listing 5.1. Erzeugt werden unter anderem die für die Kommunikation mit dem Server benötigte `PersonFinderPort_Stub` Stub Klasse sowie Klassen für die Serialisierung der zu übertragenden Objekte. Anhand dieser Klassen kann nun, wie in Listing 5.3 dargestellt ist, mit wenigen Zeilen ein JAX-RPC Client unter der Java SE implementiert werden. Die Möglichkeit der serverseitigen Implementierung unter der Java SE bietet JAX-RPC nicht an.

Listing 5.3: JAXRPCClient.java

```
1 PersonFinderService_Impl service =
2    new PersonFinderService_Impl();
3 PersonFinderPort port = service.getPersonFinderPort();
4 GetPersonResponse person = port.getPerson("Hans Meier");
```

Die Implementierung aus Listing 5.3 erfordert nur minimalen manuellen Aufwand und ermöglicht das synchrone Aufrufen der vom JAX-RPC Server angebotenen Operationen. Leider ist diese synchrone Form der Implementierung die einzige, welche von JAX-RPC angeboten wird:

> The JAX-RPC specification does not specify any standard APIs for the design of asynchronous stubs. This feature will be addressed in the future version of the JAX-RPC specification.[4]

An dieser Stelle ist bereits der Bezug zu JAX-WS zu erkennen, wo Operationen auch asynchron ausgeführt werden können. Gerade bei größeren Serviceanbietern, bei denen Operationen einige Zeit in Anspruch nehmen können, ist ein in JAX-RPC entwickelter Servicekosument demnach nicht zu gebrauchen.

5.3 Defizite

In diesem Kapitel wurden einige Defizite der JAX-RPC Spezifikation herausgearbeitet, die den Wunsch nach einer besseren Technologie aufkommen lassen. Diese Defizite lassen sich in folgenden Punkten zusammenfassen:

- Für die Entwicklung eines JAX-RPC Webservices müssen die Konfigurationsdateien jaxrpc-ri.xml und config.xml manuell erstellt werden. Die config.xml ist dabei für die Konfiguration des Tools wscompile zuständig, wobei die Datei jaxrpc-ri.xml Metainformationen über den zu veröffentlichenden Webservice liefert. Beide Dateien haben also eine berechtigte Existenz, welche jedoch durch das Konzept der in Java EE Version 5 vorgestellten Annotationen verloren geht. Wie spätere Kapitel zeigen, macht JAX-WS intensiven Gebrauch von diesen Annotationen.

- Eng mit dem obigen Punkt verbunden ist der Nachteil der zweifachen Erstellung eines WAR-Archivs. Bei der zweiten Erstellung durch das wsdeploy Tool wird die jaxrpc-ri.xml eingelesen um Tie- bzw. Stub-Komponenten zu erstellen. Auch wird in diesem Schritt ein Servlet erzeugt und in der web.xml eingetragen. Dieser Aufwand wird in JAX-WS ebenfalls durch das Konzept der Annotationen vermieden.

- Ein JAX-RPC Server kann nur als Servlet oder EJB veröffentlicht werden. Gerade bei kleineren Serviceanbietern wird man jedoch die Möglichkeit vermissen, einen JAX-RPC Server auch unter der Java SE implementieren zu können. Diese Möglichkeit bietet JAX-WS an.

- Die XML/Java Binding-Fähigkeiten von JAX-RPC sind stark eingeschränkt. In dem Beispiel der PFS-WSDL musste letztere vor dem Einlesen durch das wscompile Tool zunächst überarbeitet werden. Gerade bei einer größeren WSDL mit vielen importierten XML-Schemas ist dieser Aufwand nicht mehr tragbar, doch gerade hier liegt das Einsatzgebiet von JAX-RPC und

[4] S. [JSR101], Seite 27.

der Vorteil des wscompile Tools. In JAX-WS wird das Binding und Mapping zwischen XML-Schema und Java Klasse an eine eigene Spezifikation namens JAXB (s. Kapitel 6) delegiert.

- Die clientseitige Entwicklung eines JAX-RPC Services gestaltet sich zwar relativ komfortabel, jedoch fallen hier insbesondere die stark eingeschränkten Möglichkeiten der Implementierung auf. So ist es nicht vorgeschrieben, dass eine JAX-RPC Implementierung die Möglichkeit der asynchronen Kommunikation oder sog. One-Way Nachrichten (Operationen ohne Rückgabewert) unterstützt. Hier trumpft JAX-WS mit einer Vielzahl von Implementierungsmöglichkeiten auf. So wird die asynchrone Kommunikation über zwei verschiedene Modelle realisiert und JAX-WS bietet eine Unterstützung von One-Way Nachrichten an.

- JAX-RPC ist auf das Verschicken von SOAP-Nachrichten mit einem Binding-Style von RPC ausgelegt. So gibt es unter anderem keine Möglichkeit, bei der Generierung der WSDL über das wscompile Tool den WSDL/SOAP-Style von RPC/Encoded zu beeinflussen. Dennoch setzt sich in jüngster Zeit gerade der Document/Literal/Wrapped Style durch. JAX-WS hingegen unterstützt alle möglichen WSDL/SOAP-Styles und bietet vielfältige Möglichkeiten an, die Generierung der WSDL zu beinflussen.

Obige Auflistung liefert einen groben Überblick über die Defizite der JAX-RPC Technologie und gibt Hinweise, wie diese in JAX-WS gelöst wurden. Auf diese Hinweise wird in späteren Kapiteln vertieft eingegangen.

5.4 Zusammenfassung und Fazit

Dieses Kapitel hat einen Überblick über die JAX-RPC Technologie geliefert. Anhand einer beispielhaften Implementierung eines JAX-RPC Webservices (Server- und Client-Seite) wurden einerseits die Vorteile dieser Technologie gegenüber den Low-Level APIs für XML wie DOM und SAX aufgezeigt, zum anderen wurde aber auch auf Verbesserungspotentiale hingewiesen und angedeutet, in welcher Form diese in JAX-WS umgesetzt wurden. Beim Lesen der JAX-RPC Spezifikation ist besonders auffällig, dass an vielen Stellen auf eine Nachfolgertechnologie verwiesen wird, die bekannte Defizite von JAX-RPC beheben soll. Es scheint, als habe man bewusst auf die Einführung der Java EE/SE 5 gewartet um von den dort spezifizierten Java Annotationen Gebrauch machen zu können.
Das nächste Kapitel widmet sich der JAXB Technologie, welche eine wichtige Grundlage für die Entwicklung eines Webservices mit der JAX-WS Technologie bildet. So übernimmt JAXB die Aufgabe des XML/Java Binding und Mapping, also alles was mit den Datentypdefinitionen innerhalb der WSDL zu tun hat.

6 JAXB XML/Java Binding und Mapping

Ein großes Hindernis bei der Umsetzung eines Webservices betrifft die Umwandlung von ankommenden XML-Nachrichten in konkrete Methodenaufrufe und die Rückwandlung der Ergebnisse in ausgehende XML-Nachrichten. In Bezug auf die WSDL müssen dabei zwei Fragen geklärt werden.

1. Wie können aus der `<wsdl:types>` Sektion der WSDL adäquate Java-Klassen erzeugt werden und umgekehrt?
   Die erzeugten Java-Klassen dienen in diesem Fall als Java Repräsentationen der Datentypdefinitionen aus der WSDL. Im Folgenden wird davon ausgegangen, dass diese Datentypdefinitionen XML-Schemas sind. Dabei können XML-Schemas sowie Java-Klassen eine hohe Komplexität aufweisen, die in Einklang gebracht werden muss.

2. Wie kann die Serialisierung von Objekten dieser Java-Klassen zu Instanzen des XML-Schemas (XML-Nachrichten) und entsprechend die Deserialisierung umgesetzt werden?

Anhand dieser Fragen kann eine Unterscheidung zwischen Binding und Mapping getroffen werden, die sich an der Definition von Hansen orientiert.[1] Binding sowie Mapping bedeuten demnach zunächst einmal die Schaffung einer Abbildung zwischen XML-Schemas und Java-Klassen. Der Unterschied liegt darin, dass beim Mapping sowohl Java-Klassen als auch XML-Schemas bereits existieren, beim Binding jedoch nur eine dieser beiden Seiten. Beim Binding werden daher zunächst die Java-Klassen aus den XML-Schemas generiert, oder umgekehrt. Diese Generierung fällt ebenfalls unter den Begriff Binding. In Bezug auf die obigen Fragen bezieht sich die erste Frage demnach auf das Binding, die zweite Frage entweder auf das Binding oder auf das Mapping (je nachdem ob die erste Frage gestellt wurde, oder nicht).
Da es viele andere Anwendungsgebiete gibt, in denen ein effizientes XML/Java Binding und Mapping wünschenswert wäre, hat man diese Funktionalität in JAX-WS ausgelagert und an eine unabhängige Spezifikation namens *Java Architecture for XML Binding (JAXB)*[2] delegiert. JAXB und JAX-WS werden dabei parallel

[1] Vgl. [Han07], Seite 195.
[2] Vgl. [JSR222].

weiterentwickelt. Abbildung 6.1 illustriert das Einsatzgebiet von JAXB innerhalb von JAX-WS. Es ist zu erkennen, dass das XML/Java Binding und Mapping kom-

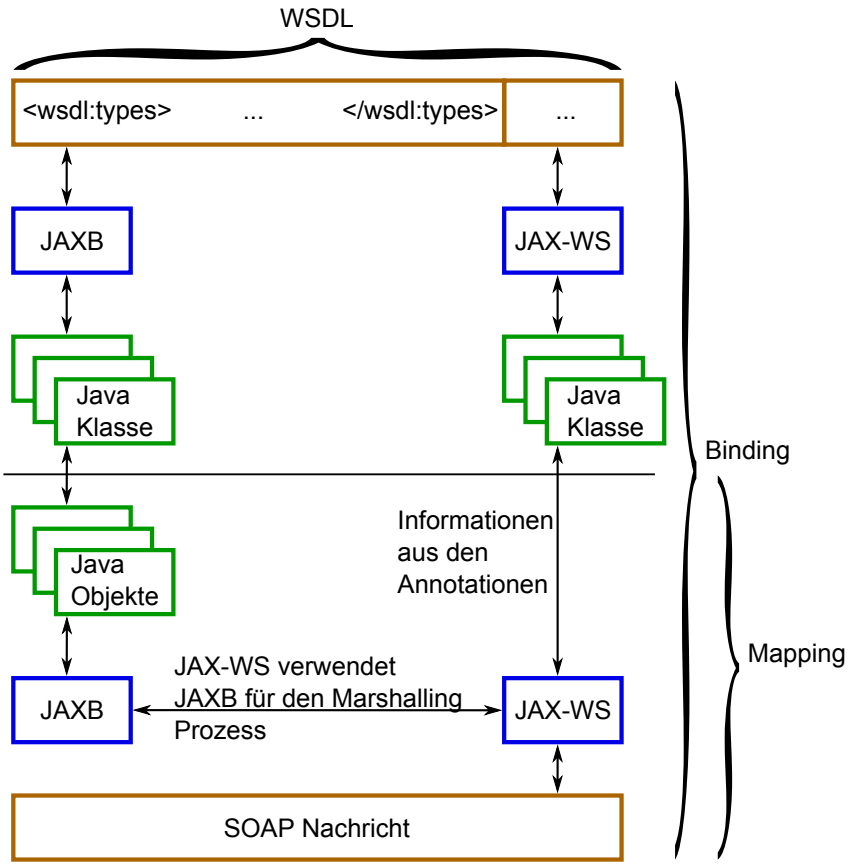

Abbildung 6.1: Einordnung von JAXB innerhalb von JAX-WS
Quelle: Eigene Darstellung

plett JAXB überlassen wird. Daher wird auch in der JAX-WS Spezifikation an vielen Stellen Bezug zu JAXB genommen. Wie spätere Kapitel zeigen, ist es trotzdem möglich, einen JAX-WS Webservice auch ohne JAXB zu implementieren. Dann muss man sich allerdings auf Low-Level XML-Technologien wie DOM oder SAX zum Parsen der XML-Nachrichten einlassen. Dieser Ansatz ist in der Praxis nur bei kleineren XML-Nachrichten praktikabel, da ansonsten schnell der Überblick über den erforderlichen Quelltext verloren geht. Leider bietet die JAX-WS Spezifikation nicht die Benutzung eines anderen XML/Java Binding und Mapping Tools an Stelle von JAXB an:

> JAX-WS 2.0 will defer data binding to JAXB; it is not a goal to provide a plug-in API to allow other types of data binding technologies to be used in place of JAXB.[3]

[3] S. [JSR224], Seite 3.

Dies ist ein erheblicher Nachteil, da JAXB, wie die folgenden Abschnitte zeigen werden, nur sehr eingeschränkte Mapping-Fähigkeiten besitzt. Aufgrund der Größe der JAXB Spezifikation (fast 400 Seiten) kann in dieser Diplomarbeit nicht jeder Aspekt von JAXB berücksichtigt werden. Stattdessen liegt der Fokus auf der Benutzung von JAXB als Werkzeug für JAX-WS.

6.1 JAXB Architektur

In diesem Abschnitt wird auf die grundlegende Architektur von JAXB eingegangen. Wie in Abbildung 6.2 dargestellt ist, besteht diese aus drei Teilen.

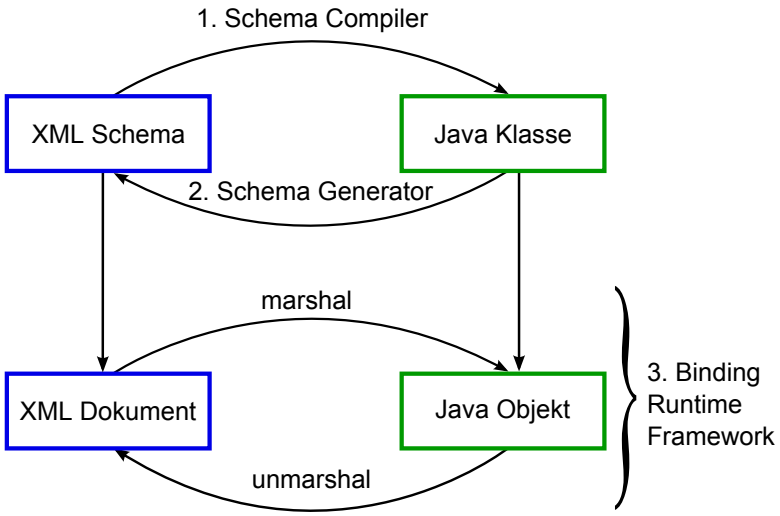

Abbildung 6.2: 3-schichtige Architektur von JAXB
Quelle: Eigene Darstellung

1. *Schema Compiler* – Der Schema Compiler erzeugt aus XML-Schemas entsprechende Java-Klassen. Der Binding-Vorgang hinter dieser Generierung kann dabei über die so genannten Binding Language Declarations beeinflusst werden, welche in Abschnitt 6.3.3 beschrieben sind. Diese werden zur Laufzeit nicht mehr verwendet, sondern manifestieren sich als entsprechende Java Annotationen im Quelltext der generierten Klassen.

2. *Schema Generator* – Der Schema Generator funktioniert in umgekehrter Richtung zu dem Schema Compiler und erzeugt aus vorhandenen Java-Klassen die entsprechenden XML-Schemas nach einem Standard Binding. Dieses Standard Binding kann in diesem Fall über Java Annotationen beeinflusst werden, welche in Abschnitt 6.3.2 aufgezeigt werden. Im Gegensatz zu den Binding Language Declarations werden die Java Annotationen von

der JAXB Laufzeitumgebung verwendet um den Un-/Marshalling Prozess zu beeinflussen.

3. *Binding Runtime Framework* – Die Laufzeitumgebung von JAXB trägt den Namen Binding Runtime Framework und ist somit der Teil der JAXB Architektur, welcher für das Binding/Mapping zur Laufzeit zuständig ist. Das Binding Runtime Framework nutzt dabei die Struktur und Annotationen der Java-Klassen um das Marshalling und Unmarshalling von XML-Dokumenten zu realisieren. Marhalling bezeichnet in diesem Zusammenhang den Prozess, aus einem XML-Dokument einen Java Objekt-Baum (engl. content tree) mit dem gleichen Inhalt wie dem des XML-Dokuments zu generieren. Dieser Java Objekt-Baum kann daraufhin in Java problemlos ausgelesen oder verändert werden. Die umgekehrte Richtung, einen Java Objekt-Baum in ein XML-Dokument mit gleichem Inhalt zu transformieren wird in diesem Zusammenhang als Unmarshalling bezeichnet. Sowohl Marshalling als auch Unmarshalling bieten die Möglichkeit zur Validation der einzulesenden bzw. zu generierenden XML-Schema Instanz an. In JAXB ist die Validation ab der hier verwendeten Version 2.0 aus Performanzgründen optional.

Als Schema Compiler und Schema Generator werden im Folgenden die Referenzimplementierungen des GlassFish Projekts[4] namens XJC (Schema Compiler) und schemagen (Schema Generator) verwendet. Nach dieser allgemeinen Erläuterung der JAXB Architektur wird nun auf die konkrete Verwendung von JAXB innerhalb von JAX-WS eingegangen. Beginnt man bei der Entwicklung eines JAX-WS Webservices (Client- oder Serverseite), so trifft man auf eines von drei unterschiedlichen Szenarien.[5] Diese in Abbildung 6.3 skizzierten Szenarien stellen die Ausgangsbasis bei der Entwicklung eines Webservices dar und sind somit für die abgrenzende Bewertung der in dieser Diplomarbeit vorgestellten Technologien besonders wichtig. Daher werden in späteren Kapiteln, in denen es um die Entwicklung eines JAX-WS Servers/Clients geht, diese Szenarien erneut aufgegriffen. In Bezug auf JAXB sind insbesondere die ersten beiden Szenarien relativ einfach zu realisieren, da JAXB sehr gute Binding-Fähigkeiten besitzt.

- *„Start von WSDL“* – Der Webservice wird anhand einer vorgegebenen WSDL in Java implementiert. Dieser Ansatz ist typischerweise bei der Entwicklung eines Servicekonsumenten anzutreffen und ist in Bezug auf JAXB über die Benutzung eines Schema Compilers gelöst (grüne Wiese).

- *„Start von Java“* – Es existieren bereits Java-Klassen. Der Webservice (mit zugehöriger WSDL) soll anhand dieser bestehenden Java-Klassen verwirklicht werden. Auch wenn dieses Szenario oftmals zu einer unschönen WSDL

[4] Vgl. [GlassFish].
[5] Vgl. [Han07].

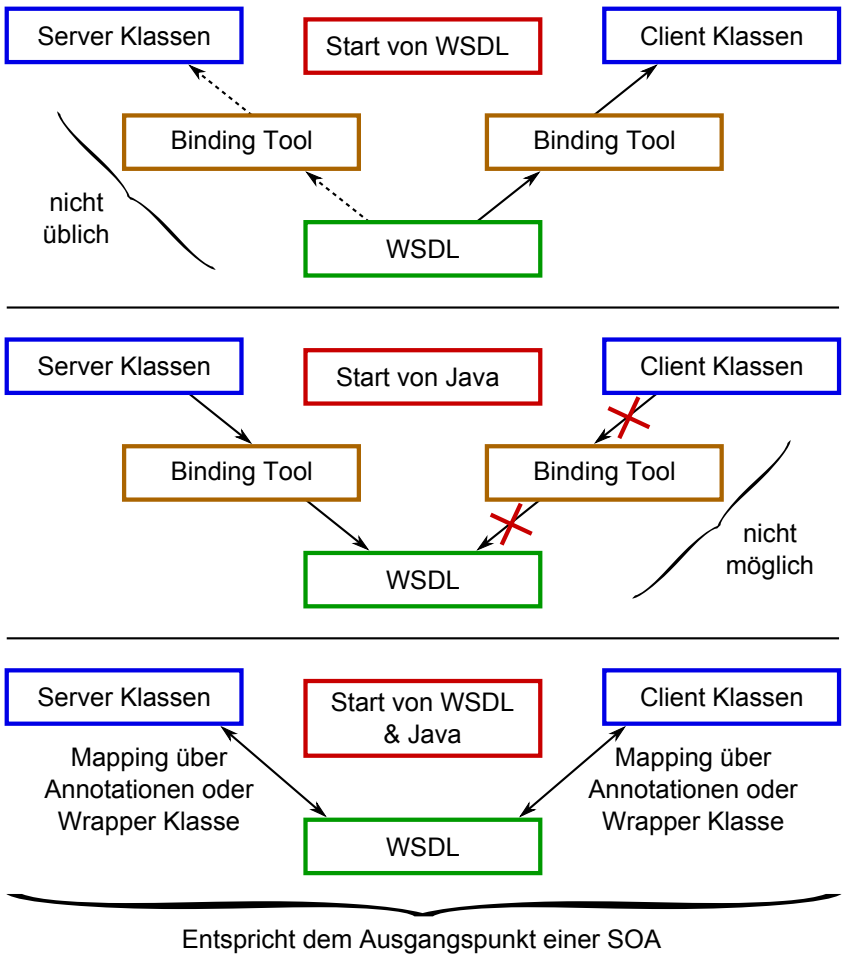

Abbildung 6.3: Mögliche Szenarien bei der Entwicklung eines Webservices
Quelle: Eigene Darstellung

führt, so ist dies der bevorzugte Weg, den viele Entwickler bei der Erstellung eines Serviceanbieters einschlagen. Für dieses Szenario bietet sich die Verwendung des Schema Generators an.

- *„Start von WSDL & Java“* – Hier existieren sowohl eine WSDL, als auch die Java-Klassen. Die Java-Klassen können in diesem Fall jedoch komplett vom Aufbau der WSDL abweichen. Dieses Szenario bezieht sich ausschließlich auf die Mapping Fähigkeiten von JAXB.

In den folgenden Abschnitten wird gezeigt, wie die drei JAXB Architekturkomponenten in den oben aufgelisteten Szenarien eingesetzt werden können. Dabei wird auch aufgezeigt, für welches Szenario diese mehr oder weniger gut geeignet sind. Als exemplarisches Beispiel dient das XML-Schema aus der `<wsdl:types>` Sektion der PFS-WSDL von Kapitel 4.

6.2 Schema Compiler

Das Erste womit der Entwickler eines Servicekonsumenten konfrontiert wird, ist die WSDL des Serviceanbieters („Start von WSDL“ Szenario). Diese ist meist unter einer festen URL abrufbar und oftmals sehr komplex (s. Kapitel 4). Um die Komplexität zu mindern, werden oft mehrere kleinere WSDL angeboten, die jeweils einen spezifischen Webservice-Endpunkt innerhalb des Webservices implementieren. Auch liegt in den meisten Fällen eine Dokumentation zur Erklärung und Benutzung der WSDL bei.
Glücklicherweise braucht der Entwickler eines Servicekonsumenten unter JAX-WS im Idealfall die WSDL des Serviceanbieters keines Blickes zu würdigen. Dieser Vorteil ruht zu großen Teilen auf der Verwendung von JAXB und seinen Binding-Fähigkeiten. Von der WSDL ausgehend sind diese in Form eines Schema Compilers implementiert. Dieser Abschnitt widmet sich also der Funktionsweise eines Schema Compilers um die `<wsdl:types>` Sektion der PFS-WSDL aus Kapitel 4 an adäquate Java-Klassen zu binden. Innerhalb der `<wsdl:types>` Sektion gibt es in diesem Fall nur ein XML-Schema mit einem komplexen und einem einfachen Element. Dieses XML-Schema wird im Folgenden als *PFS-Schema* bezeichnet.

Listing 6.1: PFS-Schema aus den Datentypdefinitionen der PFS-WSDL

```
1  <xs:schema>
2      <xs:element name="getPerson">
3          <xs:simpleType>
4              <xs:restriction base="xs:string">
5                  <xs:pattern value=".{1,12}"/>
6              </xs:restriction>
7          </xs:simpleType>
8      </xs:element>
9      <xs:element name="getPersonResponse">
10         <xs:complexType>
11             <xs:sequence>
12                 <xs:element name="postleitzahl"
                       type="xs:int"/>
13                 <xs:element name="interesse"
                       type="xs:string" maxOccurs="10"
                       minOccurs="0"/>
14                 <xs:choice>
15                     <xs:element name="alter" type="xs:int"/>
16                     <xs:element name="jahrgang"
                           type="xs:date"/>
17                 </xs:choice>
18                 <xs:element name="familienstand">
19                     <xs:simpleType>
20                         <xs:restriction base="xs:string">
21                             <xs:enumeration value="ledig"/>
```

```
22                                  <xs:enumeration
                                       value="verheiratet"/>
23                              </xs:restriction>
24                          </xs:simpleType>
25                      </xs:element>
26                  </xs:sequence>
27                  <xs:attribute name="gueltig" type="xs:boolean"/>
28              </xs:complexType>
29          </xs:element>
30  </xs:schema>
```

Übersetzt man nun das Listing 6.1 mit XJC, so werden die drei Dateien package-info.java, ObjectFactory.java und GetPersonResponse.java generiert. Die Klasse `GetPersonResponse` ist dabei die Java Repräsentation des komplexen Elements `<pfs:getPersonResponse>`. Im Folgenden werden diese drei Klassen erläutert.

Listing 6.2: package-info.java

```
1 @javax.xml.bind.annotation.XmlSchema(namespace =
     "http://pisys.de/pfs/", elementFormDefault =
     javax.xml.bind.annotation.XmlNsForm.QUALIFIED)
2 package de.pisys.pfs;
```

Diese Klasse ist anfürsich nur ein Container für die `@XmlSchema` Annotation, welche den Paketnamen annotiert. Diese Annotation speichert wichtige Informationen über das globale `<xs:schema>` Element. Dazu zählen vor allem die im Schema verwendeten benutzerdefinierten Namensräume. Durch die Existenz dieser Klasse enthalten alle XML-Nachrichten, welche durch im selben Paket liegende JAXB-Klassen erzeugt werden, den dort angegebenen Namensraum. JAXB-Klassen sind spezielle Java-Klassen, anhand dener JAXB den Un-/Marshalling-Prozess durchführt.

Listing 6.3: ObjectFactory.java

```
1  @XmlRegistry
2  public class ObjectFactory {
3      private final static QName _GetPerson_QNAME =
4          new QName("http://pisys.de/pfs/", "getPerson");
5
6      public ObjectFactory() {}
7
8      public GetPersonResponse createGetPersonResponse() {
9          return new GetPersonResponse();
10     }
11
12     @XmlElementDecl(namespace = "http://pisys.de/pfs/", name
           = "getPerson")
```

```
13      public JAXBElement<String> createGetPerson(String
            value) {
14          return new JAXBElement<String>(_GetPerson_QNAME,
                String.class, null, value);
15      }
16  }
```

Die `ObjectFactory` Klasse besitzt für alle wichtigen im PFS-Schema enthaltenen Elemente entsprechende Kreatoren. Als wichtige Elemente sind zum einen alle globalen Elemente zu verstehen, zum anderen aber auch alle nicht globalen komplexen Elemente. Letztere werden standardmäßig in Form von inneren Klassen abgebildet. Alle anderen Elemente werden als Variablen abgebildet. Kreatoren, die keine Instanz einer entsprechenden Java-Klasse erzeugen können (da keine vorhanden ist), liefern ein Objekt vom Typ `JAXBElement` zurück. Letzteres ist die JAXB Repräsentation eines XML-Elements.

Listing 6.4: GetPersonResponse.java

```
1  /* Keine Annotationen dargestellt */
2  public class GetPersonResponse {
3      protected int postleitzahl;
4      protected List<String> interesse;
5      protected Integer alter;
6      protected XMLGregorianCalendar jahrgang;
7      protected String familienstand;
8      protected Boolean gueltig;
9
10     public List<String> getInteresse() {
11         if (interesse == null) {
12             interesse = new ArrayList<String>();
13         }
14         return this.interesse;
15     }
16     /* Weitere Getter und Setter ... */
17 }
```

Die Klasse `GetPersonResponse` aus Listing 6.4 ist das JAXB Standard Binding vom globalen komplexen Element `<pfs:getPersonResponse>` aus dem PFS-Schema. Die Annotationen dieser Klasse sind insbesondere für das Binding Runtime Framework wichtig und werden demnach erst in Kapitel 6.3.2 behandelt. An dieser Stelle wird der Fokus auf die Binding-Fähigkeiten vom JAXB Schema Compiler zur Erzeugung der Klasse ohne Annotationen gelegt. Dabei fallen folgende Punkte auf:

- Das Element `<pfs:postleitzahl>` mit dem XML-Schema Datentyp `xs:int` ist ein Pflichtelement (der Vorgabewert für `xs:minOccurs` und `xs:maxOccurs`

bei Elementen ist 1). Somit muss `<pfs:postleitzahl>` auf jeden Fall in der XML-Nachricht auftauchen. Da die Java-Variable `postleitzahl` demnach also einen Wert annehmen muss, besitzt die Variable den primitiven Datentyp `int`, der nicht auf `null` gesetzt werden kann. Anders hingegen verhält es sich bei dem Element `<pfs:alter>`, welches in einer `<xs:choice>` Auswahl liegt und dem Attribut `pfs:gueltig` (Attribute sind standardmäßig immer optional). Diese werden in Java durch Wrapper-Klassen repräsentiert, welche den Wert `null` annehmen können.

- Das Element `<pfs:interesse>` hat ein `xs:maxOccurs` größer 1. Somit kann es in der XML-Nachricht mehrfach auftreten und muss in Java als Kollektionsdatentyp (z.B. Liste oder Array) abgebildet werden. Beim Standard Binding von JAXB werden solche Elemente als Liste mit dem generischen Datentyp der Java Repräsentation des XML-Schema Datentyps (in diesem Fall `xs:string`) realisiert. Es ist auffallend, dass die Variable `interesse` nur einen Getter und keinen Setter besitzt. Über die Methode `getInteresse()` erhält man eine Referenz auf die Live List. Diese kann nun über die Referenz modifiziert werden, wobei alle Änderungen sofort sichtbar sind.

- Betrachtet man die Elemente `<pfs:alter>` und `<pfs:jahrgang>` in dem PFS-Schema so fällt auf, dass beide Elemente in ein und derselben `<xs:choice>` Gruppe liegen. Es kann also immer nur eins von den beiden Elementen in einer XML-Nachricht existieren. Diese Einschränkung ist in Java nicht mehr enthalten. Hier können sowohl `alter` als auch `jahrgang` belegt werden. Die daraus resultierende (durch den Marshalling-Prozess) fehlerhafte XML-Nachricht fällt erst bei der optionalen Validation mit dem PFS-Schema auf. Es handelt sich hierbei also um eine XML-Schema Restriktion, welche beim JAXB Binding nicht umgesetzt werden kann.

- Analog zu dem vorigen Punkt berücksichtigt JAXB beim Standard Binding Prozess auch keine Enumerationen aus dem PFS-Schema. Daher erhält die Variable `familienstand` einfach den Datentyp `String`. Im Gegensatz zu vorigem Punkt, lässt sich diese XML-Schema Restriktion jedoch in JAXB über die Binding Language Declarations umsetzen.

Dieser Abschnitt widmete sich den Binding-Fähigkeiten des JAXB Schema Compilers. In Bezug auf die Entwicklung eines Webservices erleichtert der Schema Compiler insbesondere das „Start von WSDL“ Szenario, da sich der Entwickler nicht mehr selber um das Data Binding (XML-Schema → Java-Klasse) der `<wsdl:types>` Sektion kümmern muss. Dennoch hat gerade der letzte Punkt, die Demonstration der Standard Binding-Fähigkeiten von JAXB am Beispiel der Klasse `GetPersonResponse` gezeigt, dass JAXB selbst bei einem noch sehr einfachen Beispiel nicht jedes Binding korrekt umsetzen kann.

6.3 Binding Runtime Framework

Im vorherigen Abschnitt wurde die Funktionsweise des Schema Compilers angeschnitten. Dieser ist im Normalfall nur für das einmalige Erzeugen des Java-Klassengeflechts zuständig und erleichtert maßgeblich das „Start von WSDL“ Szenario bei der Entwicklung eines Webservices.
Problematischer hingegen ist der „Start von WSDL & Java“ Ansatz, in dem eine WSDL und ein bereits existierendes Java-Klassengeflecht vorliegen. Dieser Ansatz spiegelt die Einführung einer SOA wieder, da eine SOA immer nachträglich in die bestehende IT-Struktur eines Unternehmens eingefügt wird. Benötigt wird vor allem ein Tool mit guten Mapping-Fähigkeiten, welches die Datentypdefinitionen aus der bestehenden WSDL in einer sinnvollen Form auf die bestehenden Java-Klassen abbilden kann und umgekehrt. Abbildung 6.4 illustriert die von JAXB angebotenen Möglichkeiten des Mappings, welche sich vor allem in einer manuellen Bearbeitung der Java Mapping Annotationen sowie der Binding Language Declarations manifestieren. Der Einsatz einer Wrapper-Klasse ist hierbei nur der Vollständigkeit halber erwähnt. In den folgenden Abschnitten wird gezeigt, wie gut diese Mappings nach dem Beispiel der PFS-WSDL und der Klasse `GetPersonResponse` umgesetzt werden können.

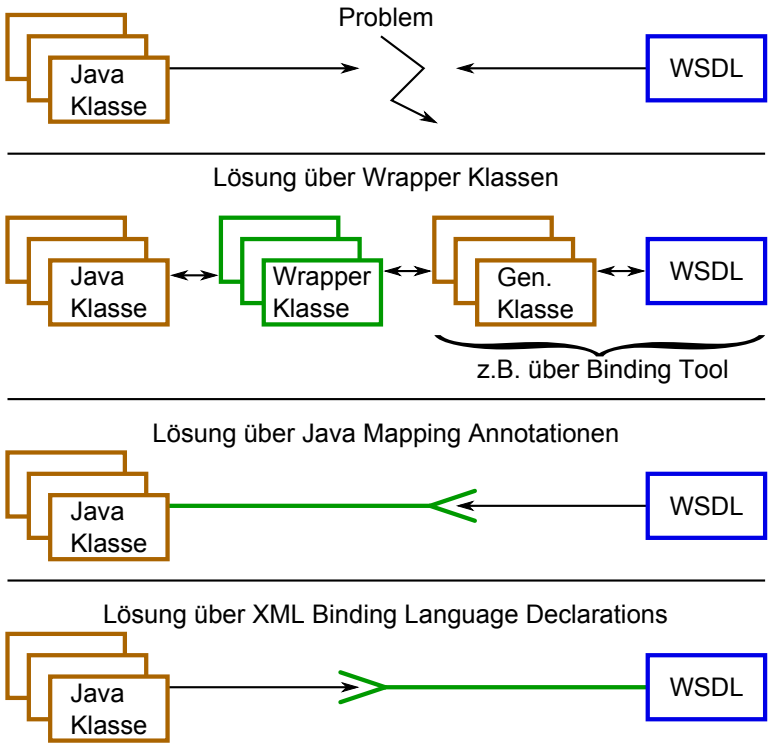

Abbildung 6.4: Möglichkeiten des Mappings von JAXB
Quelle: Eigene Darstellung

6.3.1 Wrapper-Ansatz

Der Wrapper-Ansatz ist keine spezielle Funktionalität von JAXB, jedoch erleichtert JAXB die Umsetzung dieses Ansatzes über die Verwendung des Schema Compilers. Dieser wird dazu genutzt, um anhand der bestehenden XML-Schemas das Java-Klassengeflecht zu generieren (s. Abschnitt 6.2). Nun hat man zwei Java-Klassengeflechte (das generierte und das bestehende) welche miteinander kooperieren müssen. Der Wrapper-Ansatz geht davon aus, dass man das bestehende Klassengeflecht nicht ändern kann, da es sich dabei um bereits übersetzte Klassen handelt. Eine Neuübersetzung ist oftmals zu teuer oder gar nicht möglich (keine Quelltexte vorhanden, da gekauftes Projekt). Um also das geschilderte Mapping zu ermöglichen, wird eine Java-Klasse geschrieben, welche als Bindeglied der beiden Klassengeflechte fungiert und Aufrufe entsprechend transformiert und weiterleitet. Der Wrapper-Ansatz ist für einen Entwickler relativ leicht umzusetzen, da dieser keine XML und nur minimale JAXB Kenntnisse (Schema Compiler) verlangt. Allerdings muss die Wrapper-Klasse immer angepasst werden, wenn sich die WSDL ändert.
In den folgenden zwei Abschnitten wird der Fokus auf die von JAXB angebotenen Möglichkeiten der Annotation gelegt, die ein benutzergesteuertes Mapping ermöglichen sollen. Dadurch kann auf den Einsatz einer Wrapper-Klasse verzichtet werden. Der Entwickler muss sich allerdings in JAXB einarbeiten und unter Umständen tiefere Kenntnisse in XML und XPath besitzen. Generell können über den Annotationen-Ansatz entweder die bestehenden Java-Klassen oder die XML-Schemas der WSDL (in Form der Binding Language Declarations) annotiert werden. Im Folgenden werden beide Möglichkeiten exemplarisch behandelt.

6.3.2 Java Mapping Annotationen

JAXB macht regen Gebrauch von Annotationen um ein benutzergesteuertes Mapping zu ermöglichen. Die in Kapitel 6.2 durch den Schema Compiler generierten Klassen besitzen bereits einige der von JAXB angebotenen Annotationen und wurden auch an dieser Stelle erklärt. Das JAXB Binding Runtime Framework wertet diese Annotationen aus um einerseits die Struktur der zu generierenden XML-Nachricht zu bestimmen (Marshalling), andererseits aber auch um die XML-Nachricht wieder in Java-Objekte zu konvertieren (Unmarshalling).
Im Folgenden werden die Annotationen der generierten Klasse `GetPersonResponse` aus Listing 6.4 erläutert. Diese wird, zusammen mit der Klasse `app-info.java`, vom Binding Runtime Framework für den Un-/Marshalling Prozess verwendet.

Listing 6.5: GetPersonResponse.java – annotierte Version

```
1 @XmlAccessorType(XmlAccessType.FIELD)
2 @XmlType(name = "", propOrder = {
3     "postleitzahl",
```

```
4       "interessen",
5       "alter",
6       "jahrgang",
7       "familienstand"
8   })
9   @XmlRootElement(name = "getPersonResponse")
10  public class GetPersonResponse {
11      protected int postleitzahl;
12      protected List<String> interesse;
13      protected Integer alter;
14      @XmlSchemaType(name = "date")
15      protected XMLGregorianCalendar jahrgang;
16      @XmlElement(required = true)
17      protected String familienstand;
18      @XmlAttribute
19      protected Boolean gueltig;
20
21      /* Getter und Setter ... */
22  }
```

Folgende Auflistung erklärt die in Listing 6.5 generierten Annotationen.

- `@XmlAccessorType` (annotiert Paket oder Klasse) – Definiert, welche JavaBean Properties oder Feldvariablen in die XML-Nachricht gemappt werden. Der Parameter `XmlAccessType.FIELD` gibt hierbei an, dass jede statische und nicht mit `@XmlTransient` annotierte Feldvariable automatisch in der XML-Nachricht enthalten ist.

- `@XmlType` (annotiert Enumeration oder Klasse) – Diese Annotation steht in Verbindung mit der `@XmlAccessorType` Annotation. Letztere gibt eine Zusammenstellung „was“ alles gemappt wird, die `@XmlType` Annotation besagt dann „wie“ diese Zusammenstellung gemappt wird. In diesem Fall wird die Zusammenstellung zu einem anonymen Typen gemappt, wobei die Elemente in der durch `propOrder` definierten Reihenfolge in der XML-Nachricht enthalten sein müssen.

- `@XmlRootElement` (annotiert Enumeration oder Klasse) – Legt ein Wrapper-Element mit dem Namen `name` um den aus einer Klasse oder Enumeration erzeugten komplexen Datentyp (`xs:complexType`). Dieses Element stellt ein globales Element innerhalb eines XML-Schemas dar.

- `@XmlSchemaType` (annotiert JavaBean Property, Feldvariable oder Paket) – Besagt, dass der Datentyp `XMLGregorianCalendar` auf den XML-Schema Typen `xs:date` abgebildet werden soll. Diese Angabe ist an dieser Stelle notwendig, da es in der XML-Schema Spezifikation acht verschiedene Da-

tentypen für Zeitangaben[6] wie `xs:dateTime`, `xs:time`, `xs:date` und `xs:g*` gibt. Damit in Java nicht acht neue Datentypen definiert werden müssen, hat man sich auf `XMLGregorianCalendar` beschränkt. Dieser wird durch die `@XmlSchemaType` Annotation auf den jeweils passenden XML-Schema Typen gemappt.

- `@XmlElement` (annotiert JavaBean Property oder Feldvariable) – Normalerweise werden automatisch alle JavaBean Properties oder Feldvariablen innerhalb einer Klasse als Elemente in den aus der Klasse entstehenden `xs:complexType` gemappt. An dieser Stelle wird durch die Annotation das Element `<pfs:familienstand>` als Pflichtelement (`required=true`) definiert.

- `@XmlAttribute` (annotiert Feldvariable oder JavaBean Property) – Die Variable `gueltig` wird als Attribut deklariert. Somit wird diese Variable in der XML-Nachricht zu einem Attribut des Elements `<pfs:getPersonResponse>` abgebildet. Lässt man diese Annotation weg, so wirft Java beim Marshalling die Fehlermeldung „Property gueltig is present but not specified in `@XmlType.propOrder`“. Dies liegt daran, dass durch die Verwendung der Annotation `@XmlAccessorType(XmlAccessType.FIELD)` bereits gefordert wird, dass das komplette Feld, inklusive der Variable `gueltig`, serialisiert werden soll.

Die soeben beschriebenen Annotationen sind aus dem Standard Binding von JAXB durch den Schema Compiler generiert worden. In diesem Abschnitt geht es jedoch um den „Start von WSDL & Java“ Ansatz, in dem die WSDL und die Java-Klassen deutlich voneinander abweichen können. Um also die Mapping-Fähigkeiten von JAXB zu demonstrieren, wird im Folgenden mit einer veränderten Version des PFS-Schemas gearbeitet. Dieses Schema ist in Listing 6.6 dargestellt.

Listing 6.6: Geänderte Version des PFS-Schemas

```
1 <xs:schema>
2   <xs:complexType name="getPersonResponse">
3     <xs:sequence>
4       <xs:element name="gueltig" type="xs:boolean"/>
5       <xs:element name="familienstand" type="xs:string"
           nillable="true"/>
6       <xs:element name="interessen" minOccurs="0">
7         <xs:complexType>
8           <xs:sequence>
9             <xs:element name="interesse" type="xs:string"
                 nillable="true" maxOccurs="unbounded"
                 minOccurs="0"/>
```

[6] Vgl. [XS2].

```
10            </xs:sequence>
11          </xs:complexType>
12        </xs:element>
13        <xs:element name="geburtsjahr" type="xs:gYear"
             minOccurs="0"/>
14      </xs:sequence>
15    </xs:complexType>
16 </xs:schema>
```

Folgende Auflistung zeigt die Unterschiede zwischen dem alten PFS-Schema aus Listing 6.1 und dem neuen PFS-Schema.

- Es gibt kein globales (Wrapper-)Element mehr. Stattdessen besitzt die PFS-WSDL nun einen Binding-Style von RPC (s. Kapitel 4.2), wodurch die `<wsdl:types>` Sektion nur noch komplexe/einfache Typen und keine globalen Elemente mehr beinhalten darf.

- Das Attribut `pfs:gueltig` ist nun ein Pflichtelement.

- Es gibt ein neues Element `<pfs:interessen>`, welches als lokales Wrapper-Element für die `<pfs:interesse>` Elemente fungiert. Dadurch sind Letztere in der XML-Nachricht gruppiert und nicht auf einer Ebene wie die anderen Elemente.

- Die Elemente `<pfs:postleitzahl>` und `<pfs:alter>` gibt es nicht mehr.

- Das Element `<pfs:geburtsdatum>` wird zu `<pfs:geburtsjahr>` und besitzt aufgrund dieser Änderung nicht mehr den XML-Schema Typen `xs:date`, sondern `xs:gYear`.

- Das Element `<pfs:familienstand>` ist jetzt `xs:nillable`. Das heißt, wenn die Variable `familienstand` nicht gesetzt ist, so erscheint das Element trotzdem in der XML-Nachricht, erhält aber explizit ein Attribut mit dem Namen `xs:nil` und dem Wert `xs:true`.

- Die Reihenfolge der Elemente hat sich aufgrund der Umstrukturierung geändert.

An dieser Stelle wird nun kein Schema Compiler benutzt um die Java-Klassen über die Binding-Fähigkeiten von JAXB zu erzeugen. Stattdessen wird auf die aus dem alten PFS-Schema generierte Klasse `GetPersonResponse` zurückgegriffen. Diese wird nun anders annotiert um das Mapping mit dem geänderten PFS-Schema zu ermöglichen. Außer den Annotationen dieser Klasse bleibt der Quelltext dabei unverändert.

Listing 6.7: GetPersonResponse.java mit neuen Annotationen

```
1 @XmlAccessorType(XmlAccessType.FIELD)
2 @XmlType(name = "getPersonResponse", propOrder = {"gueltig",
      "familienstand", "interessen", "jahrgang"})
3 public class GetPersonResponse {
4     @XmlTransient
5     protected int postleitzahl;
6     @XmlElementWrapper(name = "interessen")
7     protected List<String> interesse;
8     @XmlTransient
9     protected Integer alter;
10    @XmlElement(name = "geburtsjahr")
11    @XmlSchemaType(name = "gYear")
12    protected XMLGregorianCalendar jahrgang;
13    @XmlElement(required = true, nillable = true)
14    protected String familienstand;
15    @XmlElement(required = true)
16    protected Boolean gueltig;
17
18    /* Getter und Setter ... */
19 }
```

Wie man in Listing 6.7 erkennen kann, können selbst komplexere Mappings mit den Java Mapping Annotationen elegant gelöst werden. Interessant sind in diesem Listing zum einen die `@XmlElementWrapper(name="interessen")` Annotation, welche ein Wrapper-Element mit dem Namen `<pfs:interessen>` um die `<pfs:interesse>` Elemente legt und zum anderen die `@XmlTransient` Annotation, welche ein Mapping der annotierten Feldvariable oder JavaBean Property verhindert. Trotz dieser vermeintlichen Vorteile geht dieser Ansatz mit zwei ganz erheblichen Nachteilen einher.

- Das Mapping kann immer nur eins zu eins erfolgen. Es können zum Beispiel nicht zwei Java-Variablen (wie `strasse` und `plz`) auf ein einziges XML-Element (wie `<adresse>`) gemappt werden. Abbildung 6.5 illustriert solche auch als Multivariate Type Mappings[7] bezeichneten Mappings. Diese werden in der derzeit aktuellen Version von JAXB (Version 2.1) nicht unterstützt.

- Ein Mapping zeichnet sich dadurch aus, dass der Quelltext der Java-Klassen oft fehlt oder eine Neuübersetzung nicht praktikabel ist. In einem solchen Szenario ist dieser Ansatz leider nicht durchführbar. Die Klassen müssen mit den neuen Annotationen erneut übersetzt werden um das Mapping zu ermöglichen. In diesem Zusammenhang stellt sich die Frage, ob dann nicht

[7] Vlg. [Han07], Seite 245.

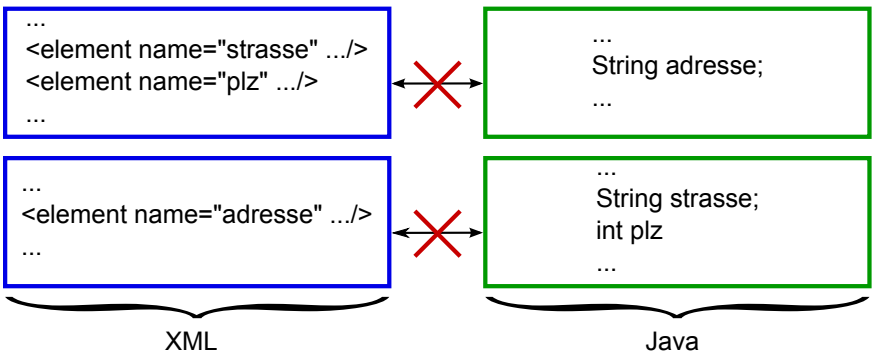

Abbildung 6.5: Multivariate Type Mappings
Quelle: Eigene Darstellung

eine Anpassung des Quelltextes (wenn dies möglich ist) komfortabler wäre, als mit den JAXB Java Annotationen zu arbeiten. Obiges Problem könnte z.B. durch ein Konzept von externen Java Annotationen behoben werden. Diese könnten dann zur Laufzeit geladen werden und die zu annotierenden Java-Elemente, wie bei den externen Binding Language Declarations, über eine zu XPath ähnliche Technologie referenzieren.

Die im folgenden Abschnitt vorgestellten Binding Language Declarations gehen das Mapping von der anderen Seite, der XML-Seite an. Es wird gezeigt, dass leider auch dieser Weg eine Neuübersetzung des Quelltextes erforderlich macht und eng mit diesem Ansatz verwandt ist.

6.3.3 Binding Language Declarations

Im vorigen Abschnitt wurde das Mapping-Problem durch JAXB Annotationen im Java Quelltext gelöst. Wie bereits erwähnt, ist dieser Ansatz oftmals nicht praktikabel (kein Quelltext vorhanden, Neuübersetzung zu aufwendig). Auch die in diesem Kapitel vorgestellten Binding Language Declarations schaffen keine Abhilfe. Es gibt leider keine Möglichkeit diese zur Laufzeit dynamisch zu laden, so dass eine erneute Generierung der Java-Klassen aus dem XML-Schema inkl. der Binding Language Declarations erfolgen muss.
Der einzige Nutzen der Binding Language Declarations liegt demnach darin, die Struktur der über den Schema Compiler erzeugten Java-Klassen zu ändern. Um dieses Einsatzgebiet sinnvoll erörtern zu können, sind zwei Annahmen zu treffen. Zum einen hat sich die JAXB-Klasse `GetPersonResponse` in einer neuen Version des Person Info Systems geändert und besitzt nun den in Listing 6.8 dargestellten Aufbau. Zum anderen wird davon ausgegangen, dass diese Klasse Teil eines Großprojekts mit vielen Abhängigkeiten ist und die Struktur der Klasse, ausgenommen der Annotationen dieser Klasse, daher nicht geändern werden darf.

Listing 6.8: PersonResponse.java

```
1  package com.pfs;
2  public class PersonResponse {
3      protected int postleitzahl;
4      protected List<String> interesse = new Vector<String>();
5      protected Integer alter;
6      protected Date jahrgang;
7      protected PersonResponse.FamilienstandEnum
           familienstand;
8      protected Boolean gueltig;
9
10     public enum FamilienstandEnum {
11         ledig,
12         verheiratet;
13         public String value() {
14             return name();
15         }
16         public static PersonResponse.FamilienstandEnum
               fromValue(String v) {
17             return valueOf(v);
18         }
19     }
20     /* Weitere Getter und Setter ... */
21 }
```

Folgende Eigenschaften sind im Vergleich zur Klasse GetPersonResponse.java aus Listing 6.4 geändert worden.

- Der Paketname wurde von `de.pisys.pfs` auf `com.pfs` gekürzt.
- Der Klassenname wurde von `GetPersonResponse` auf `PersonResponse` geändert.
- Die Variable `interesse` hat nun den Kollektionsdatentyp Vector.
- Die Variable `jahrgang` hat nicht mehr den Datentyp `XMLGregorianCalendar`, sondern `Date`.
- Die Variable `familienstand` ist nun eine Enumeration mit dem Typ `PersonResponse.FamilienstandEnum`. Mögliche Werte sind die aus dem PFS-Schema bekannten Werte `ledig` und `verheiratet`.

Das PFS-Schema bleibt in diesem Fall jedoch unverändert und besitzt immer noch den aus Listing 6.1 bekannten Aufbau. Auf der einen Seite könnte nun über die JAXB Java Annotationen versucht werden, ein Mapping mit dem alten PFS-Schema herzustellen. In diesem Abschnitt werden jedoch die Binding Language Declarations verwendet um dieses Mapping durchzuführen. Diese werden

dabei entweder direkt in das XML-Schema eingetragen oder in einer externen Datei (der sog. Binding-Datei) ausgelagert und referenzieren das XML-Schema über XPath[8]. Beide Wege verwenden dabei das `<xs:appinfo>` Element[9] um eine Beziehung zwischen den Binding Language Declarations und den Elementen des XML-Schemas herzustellen. Das `<xs:appinfo>` Element ist keine Besonderheit von JAXB, sondern Teil der XML-Schema Spezifikation und ist für die Angabe von programmspezifischen Informationen vorgesehen.
In diesem Fall werden die Binding Language Declarations in einer externen Binding-Datei mit dem Namen bindings.xml ausgelagert und referenzieren das PFS-Schema über XPath. Listing 6.9 stellt den Inhalt dieser Binding-Datei dar.

Listing 6.9: bindings.xml

```
<jaxb:bindings>
    <jaxb:bindings schemaLocation="pfs-schema.xsd">
        <jaxb:bindings node="/xs:schema">
            <jaxb:schemaBindings>
                <jaxb:package name="com.pfs"/>
            </jaxb:schemaBindings>
            <jaxb:bindings node="./xs:element
               [@name='getPersonResponse']">
                <jaxb:class name="PersonResponse"/>
                <jaxb:bindings
                   node=".//xs:element[@name='interesse']">
                    <jaxb:property collectionType=
                       "java.util.Vector"/>
                </jaxb:bindings>
                <jaxb:bindings node=".//xs:element
                   [@name='jahrgang']">
                    <jaxb:property>
                        <jaxb:baseType
                           name="java.util.Date"/>
                    </jaxb:property>
                </jaxb:bindings>
                <jaxb:bindings
                   node=".//xs:element[@name='familienstand']
                    /xs:simpleType">
                    <jaxb:typesafeEnumClass
                       name="FamilienstandEnum"/>
                    <jaxb:bindings node=".//xs:enumeration
                       [@value='ledig']">
                        <jaxb:typesafeEnumMember
                           name="ledig"/>
                    </jaxb:bindings>
```

[8] Vgl. [XPath1.1].
[9] Vgl. [XS1].

```
22                      <jaxb:bindings node=".//xs:enumeration
                           [@value='verheiratet']">
23                          <jaxb:typesafeEnumMember
                               name="verheiratet"/>
24                      </jaxb:bindings>
25                  </jaxb:bindings>
26              </jaxb:bindings>
27          </jaxb:bindings>
28      </jaxb:bindings>
29  </jaxb:bindings>
```

Über das `<jaxb:bindings>` Element wird genau ein beliebiges Element aus dem PFS-Schema referenziert. In diesem Fall dient dazu ein XPath-Audruck, der im `jaxb:node` Attribut steht. Sind bei dem referenzierten Element (das sog. Target Element) die Elemente `<xs:annotation>` und `<xs:appinfo>` noch nicht vorhanden, so werden diese automatisch erstellt. Anschließend wird der Inhalt des `<jaxb:bindings>` Elements in das `<xs:appinfo>` Element geschrieben. In Diesen befinden sich die Binding Language Declarations, deren Funktionsweise im Kontext dieses Beispiels erläutert werden.

- `<jaxb:schemaBindings>` → `<jaxb:package>`
  Ändert den Paketnamen der erzeugten Java-Klasse.

- `<jaxb:class>` → `jaxb:name`
  Ändert den Namen der erzeugten Java-Klasse.

- `<jaxb:property>` → `jaxb:collectionType`
  Ändert den standardmäßigen Kollektionsdatentypen eines Elements mit `xs:maxOccours` > 1.

- `<jaxb:property>` → `<jaxb:baseType>`
  Ändert den Datentypen der standardmäßigen Java-Abbildung. An dieser Stelle wird der interne Parser von JAXB verwendet um `<xsd:date>` auf `Date` abzubilden. Sollte dies nicht möglich sein, so bietet JAXB die Registrierung von externen Java-Parsern über die Attribute `jaxb:parseMethod` und `jaxb:printMethod` des Kindelements `jaxb:<baseType>` an.

- `<jaxb:typesafeEnumClass>` → `jaxb:name`
  Erzwingt die Generierung einer enum-Klasse aus einem `xs:simpleType`. Das Attribut `jaxb:name` gibt den Namen der enum-Klasse in Java an.

- `<jaxb:typesafeEnumMember>` → `jaxb:name`
  Gibt den Namen eines einzelnen Elements innerhalb der Enumeration an.

Verwendet man nun den Schema Compiler um aus dem alten PFS-Schema unter Einbeziehung dieser Binding-Datei die Java-Klassen zu generieren, so entsteht

genau die in Listing 7.6 aufgezeigte JAXB-Klasse. Diese kann nun die veraltete Klasse `GetPersonResponse` ersetzen ohne dabei projektspezifische Abhängigkeiten zu gefährden. Die Binding Language Declarations werden nach diesem Schritt nicht mehr verwendet und dienen nur zur Erzeugung der JAXB Java Annotationen der generierten Klasse. Diese wiederum werden schließlich vom Binding Runtime Framework für den Un-/Marshalling-Prozess verwendet. Dabei stellt sich die Frage, ob dann nicht gleich auf die Java Annotationen zurückgegriffen werden sollte, da diese für die meisten Entwickler sicherlich einfacher zu realisieren sind.

6.4 Zusammenfassung und Fazit

In diesem Kapitel wurden die Architektur und Funktionsweise von JAXB erläutert. Obwohl der Fokus dieser Diplomarbeit auf JAX-WS liegt, nimmt JAXB doch einen erheblichen Teil der JAX-WS Spezifikation ein. Dies liegt auch daran, dass JAX-WS kein anderes XML/Java Binding und Mapping Tool als JAXB vorsieht.
In Bezug auf die drei möglichen Szenarien mit denen der Entwickler bei der Erstellung eines Webservices konfrontiert werden kann (s. Abschnitt 6.1), lässt sich unter Berücksichtigung dieses Kapitels folgende Beurteilung für JAXB aufstellen.

- „Start von WSDL“ – Hier liegen die Stärken von JAXB. Mit dem Schema Compiler gestaltet es sich sehr einfach, aus einem bestehenden XML-Schema die zugehörigen Java-Klassen zu generieren. Diese sind dann bereits mit den von JAXB angebotenen Annotationen versehen anhand derer das Binding Runtime Framework den Un/-Marshalling-Prozess automatisch durchführen kann. Einziger Nachteil sind die eingeschränkten Möglichkeiten des Standard Bindings – Enumerationen werden z.B. nicht auf Java-Klassen abgebildet. Diese Probleme können jedoch mit den Java Annotationen oder den Binding Language Declarations gelöst werden. Was bleibt sind die nicht umsetzbaren XML-Schema Restriktionen.

- „Start von Java“ – Dieses Szenario ist ein wenig aufwendiger als der „Start von WSDL“ Ansatz. Zwar gibt es einen Schema Generator, der dem Entwickler die Erzeugung des XML-Schemas abnimmt, dazu müssen die Java-Klassen jedoch zuerst über die JAXB Java Annotationen annotiert werden. Dennoch ist auch dieses Szenario in Bezug auf JAXB sehr einfach zu realisieren.

- „Start von WSDL & Java“ – Für dieses Szenario eignet sich JAXB nur sehr bedingt. Ist der Quelltext der Java-Klassen nicht zugänglich, dann kann ein Mapping nur über den Einsatz einer Wrapper-Klasse realisiert werden.

Sowohl die JAXB Java Annotationen als auch die Binding Language Declarations erfordern eine erneute Übersetzung der JAXB-Klassen. Wünschenswert wäre hier der Ansatz, ein dynamisches Mapping zu realisieren (z.B. in einer externen Mapping-Datei).

Anhand dieser Einordnung lässt sich bereits erkennen, dass JAX-WS Schwierigkeiten mit der Realisierung des für die Einführung einer SOA typischen Szenarios „Start von WSDL & Java“ hat. Zumindest wenn die entsprechende JAX-WS Implementierung JAXB als XML/Java Binding und Mapping Tool verwendet. Die folgenden beiden Kapitel beschreiben jeweils die clientseitigen sowie die serverseitigen Implementierungsmöglichkeiten von JAX-WS.

7 JAX-WS Client

In den letzen Kapiteln wurden mitunter die Grundlagen behandelt, auf denen ein JAX-WS Webservice basiert. Neben einer ausführlichen Erläuterung der WSDL inkl. der verschiedenen Styles wurde in Kapitel 6 insbesondere vertieft auf JAXB eingegangen. Da JAXB und JAX-WS parallel entwickelt werden, finden sich einige Gemeinsamkeiten zwischen den beiden Technologien wieder. So machen beide Technologien regen Gebrauch der Java Annotationen bzw. der XML Binding Language Declarations und können Java-Klassen aus XML-Dokumenten generieren und umgekehrt. JAX-WS kann jedoch nur mit Hilfe von JAXB aus der `<wsdl:types>` Sektion der WSDL Java-Klassen generieren. Letzteres spiegelt die generierte Entwicklung eines Servicekonsumenten unter JAX-WS wieder. Es gibt jedoch auch die Möglichkeit einer manuellen Implementierung, welche ganz ohne die JAXB Technologie auskommt. Auf der anderen Seite abstrahiert JAX-WS von allen Details der Kommunikation, um die sich der Entwickler im Normalfall selber kümmern müsste. Abbildung 7.1 illustriert die Möglichkeiten der Im-

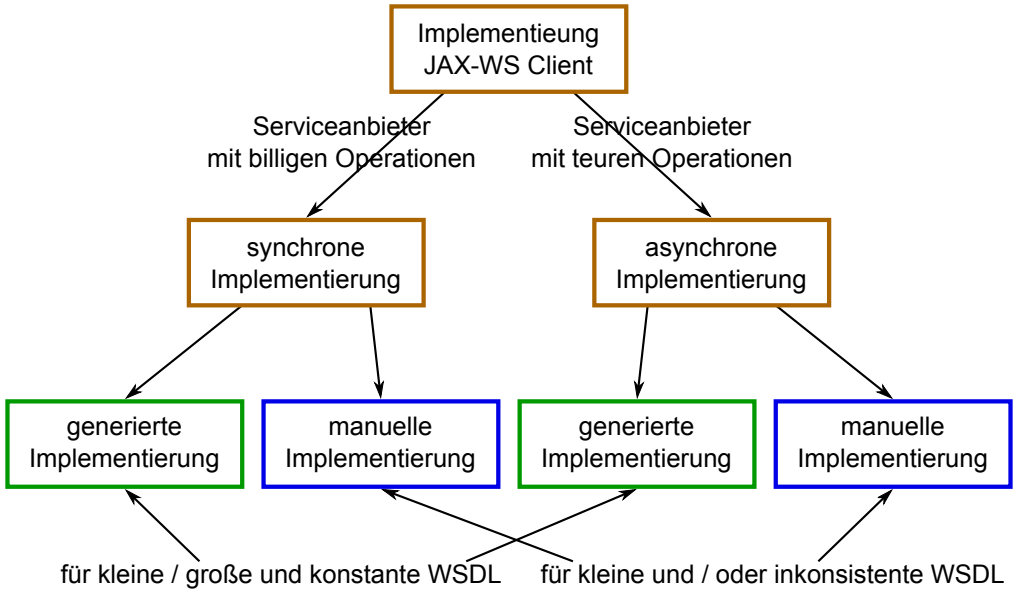

Abbildung 7.1: Implementierung eines JAX-WS Clients
Quelle: Eigene Darstellung

plementierung eines JAX-WS Clients. Es ist zu erkennen, dass außer der Wahl zwischen einer generierten oder einer manuellen Implementierung, der Entwickler

sich überdies noch zwischen einer synchronen und einer asynchronen Implementierung entscheiden muss.
Die Wahl der geeigneten Implementierung ist nicht trivial und kann nicht willkürlich festgelegt werden. Sie entscheidet sich vielmehr von der Größe und Konsistenz der angebotenen WSDL sowie dem Laufzeitverhalten der vom Serviceanbieter angebotenen Operationen. Die folgenden Abschnitte behandeln alle wichtigen Implementierungen und geben Aufschluss, in welchem Szenario diese am besten eingesetzt werden können.

7.1 Synchrone Implementierung

Die synchrone Implementierung geht davon aus, dass die vom Serviceanbieter angebotenen Operationen verhältnismäßig billig sind. Dies bedeutet, dass Operationen schnell ausgeführt und die Ergebnisse entsprechend schnell an den Servicekonsumenten zurückgeliefert werden. Der Serviceanbieter zeichnet sich also durch ein gutes Laufzeitverhalten aus.
Da die synchrone Implementierung sowohl generiert also auch manuell erfolgen kann, werden im Folgenden beide Möglichkeiten dargestellt.

7.1.1 Generierte Implementierung

Die generierte Implementierung eines JAX-WS Clients anhand einer WSDL spiegelt das „Start von WSDL“ Szenario aus Kapitel 6.1 wieder und ist ein Vertreter der „grünen Wiese“. Generiert bedeutet in diesem Zusammenhang also, dass alle benötigten Java-Artefakte von der WSDL aus generiert werden. Für diesen Zweck gibt es in JAX-WS die High-Level API, welche unter anderem auf der Nutzung von JAXB beruht und über einen WSDL zu Java Generator realisiert ist.
Um die generierte Implementierung zu demonstrieren wird nun anhand der PFS-WSDL aus Listing 4.1 der Webservice-Client erstellt. Genauer gesagt, wird anhand des `<wsdl:portType>` Elements aus der PFS-WSDL das *Service Endpoint Interface (SEI)* erstellt, welches den konkreten Webservice-Endpunkt darstellt. Im Kontext des Person Finder Services gibt es nur einen Webservice-Endpunkt mit dem Namen Person Finder Port, welcher in der PFS-WSDL durch das Element `<pfs:PersonFinderPort>` repräsentiert ist.
Für die Erzeugung der Java-Artefakte aus der PFS-WSDL wird die Referenzimplementierung des WSDL zu Java Generators namens wsimport verwendet. Unter den fünf von wsimport erzeugten Dateien befinden sich auch die drei aus dem JAXB Schema Compiler (s. Kapitel 6.2) erzeugten Dateien namens package-info.java, ObjectFactory.java und GetPersonResponse.java. Neu sind hingegen die Dateien PersonFinderService.java und PersonFinderPort.java, welche direkt von JAX-WS erzeugt und im Folgenden erläutert werden.

Listing 7.1: PersonFinderPort.java

```
1 @WebService(name = "PersonFinderPort",
2    targetNamespace = "http://pisys.de/pfs")
3 @SOAPBinding(parameterStyle =
    SOAPBinding.ParameterStyle.BARE)
4 @XmlSeeAlso({
5     ObjectFactory.class
6 })
7 public interface PersonFinderPort {
8     @WebMethod
9     @WebResult(name = "getPersonResponse", targetNamespace =
          "http://pisys.de/pfs", partName = "part1")
10    public GetPersonResponse getPerson(
11        @WebParam(name = "getPerson", targetNamespace =
              "http://pisys.de/pfs", partName = "part1")
12        String part1);
13 }
```

Das Interface `PersonFinderPort` ist das aus dem `<pfs:PersonFinderPort>` Element der PFS-WSDL erzeugte SEI. Wichtig ist vor allem die `@WebService` Annotation die dieses Interface als SEI definiert. Dies ist auch die einzige Annotation, welche JAX-WS bei der Erzeugung eines Webservices aus einer vorhandenen Java-Klasse benötigt. Innerhalb des SEI werden alle mit `@WebMethod` annotierten Methoden als Operationen vom Webservice-Endpunkt angeboten. Die Annotationen `@WebResult` und `@WebParam` speichern nur optionale Mappinginformationen. An dieser Stelle ist insbesondere die `@SOAPBinding` Annotation interessant. Diese resultiert aus der SOAP-Binding Erweiterung der WSDL und überträgt den WSDL/SOAP-Style (Binding-Style, Use und Parameter-Style) in die Java Welt. Da die `style` und `use` Elemente dieser Annotation korrekt mit den standardmäßigen Werten belegt sind, sind diese in der Annotation nicht sichtbar. Interessanterweise ist jedoch das Element `parameterStyle` dieser Annotation mit dem Wert `BARE`[1] belegt, obwohl die PFS-WSDL einen Parameter-Style von Wrapped besitzt. Dies hängt damit zusammen, dass die PFS-WSDL einen Binding-Style von Document besitzt und jedes `<wsdl:message>` Element nur ein `<wsdl:part>` Element besitzt. Dadurch fungieren die referenzierten Elemente automatisch als Wrapper-Elemente und es bedarf keinem zusätzlichen Wrapper-Element. Dennoch ist die `@SOAPBinding` Annotation an dieser Stelle irreführend.
Zur Realisierung der Anfragen an einen Serviceanbieter müssen nun die mit `@WebMethod` annotierten Methoden des SEI aufgerufen werden. Hier tritt die eigentliche „Magie“ von JAX-WS zutage, die sich in der Klasse `PersonFinderService` in Form des dynamischen Proxies manifestiert. Listing 7.2 spiegelt den Inhalt dieser Klasse wieder.

[1] Der `parameterStyle BARE` entspricht dem WSDL/SOAP-Style Unwrapped

Listing 7.2: PersonFinderService.java

```
1 @WebServiceClient(name = "PersonFinderService",
    targetNamespace = "http://pisys.de/pfs",
2   wsdlLocation = "http://localhost:8080/PersonInfoSystem/
      PersonFinderService?WSDL")
3 public class PersonFinderService extends Service {
4     /* PERSONFINDERSERVICE_WSDL_LOCATION */
5
6     public PersonFinderService(URL wsdlLocation, QName
        serviceName) {
7         super(wsdlLocation, serviceName);
8     }
9
10    public PersonFinderService() {
11        super(PERSONFINDERSERVICE_WSDL_LOCATION,
12            new QName("http://pisys.de/pfs",
                "PersonFinderService"));
13    }
14
15    @WebEndpoint(name = "PersonFinderPort")
16    public PersonFinderPort getPersonFinderPort() {
17        return super.getPort(
18            new QName("http://pisys.de/pfs",
                "PersonFinderPort"), PersonFinderPort.class);
19    }
20 }
```

Die von wsimport erzeugte Klasse `PersonFinderService` ist die Java Repräsentation des `<pfs:PersonFinderService>` Elements aus der PFS-WSDL, also eine Aggregation verschiedener Webservice-Endpunkte und spiegelt die clientseitige Sicht auf den Webservice wieder. Ähnlich zu JAXB macht JAX-WS regen Gebrauch von Annotationen um die Eigenschaften der WSDL-Elemente (in diesem Fall des `<wsdl:service>` Elements) auf Java-Klassen abzubilden. Diese sind in diesem Fall jedoch selbsterklärend.
Für den Entwickler ist insbesondere die `getPersonFinderPort()` Methode von Interesse. Augenscheinlich liefert diese eine Instanz der Klasse `PersonFinderPort`, also des SEI zurück. Das Laufzeitsystem von JAX-WS generiert jedoch bei dem Aufruf der Methode `Service.getPort()` einen dynamischen Proxy und konfiguriert diesen anhand der Informationen aus den Java Annotationen des SEI sowie der Service-Klasse `PersonFinderService` vor. Der Programmierer muss sich also nicht selbst um die Erstellung einer Proxy Instanz kümmern, sondern nutzt die zurückgelieferte Referenz auf den Person Finder Port um Anfragen abzusetzen. Obwohl der dynamische Proxy bereits fester Bestandteil von Java 1.3 ist, wird er in Bezug auf JAX-WS im folgenden Abschnitt genauer erläutert, da er die

Kommunikation zwischen Servicekonsument und Serviceanbieter erheblich vereinfacht.

Dynamischer Proxy

Der dynamische Proxy[2] verwirklicht das *Stellvertreter Entwurfsmuster* in Java und stellt eine clientseitige Stub-Komponente dar. Eine Stub-Komponente bezeichnet allgemein ein Stück Programmcode, welches als Platzhalter für ein anderes Stück Programmcode dient. In Bezug auf JAX-WS ist die Stub-Komponente also ein Platzhalter für die Implementierung des Person Finder Ports, welcher natürlich nur im PISys-Server implementiert ist. Methodenaufrufe auf Objekte dieses Typs werden über die Stub-Komponente nach dem Prinzip des *Remote Procedure Calls (RPC)*in SOAP-Nachrichten transformiert und über das Netzwerk an den PISys-Server geschickt. Wie bereits erwähnt, wird die Stub-Komponente von der JAX-WS Laufzeitumgebung beim Aufruf der `Service.getPort()` Methode automatisch erstellt und entsprechend konfiguriert. Diese Eigenschaft, zusammen mit der Realisierung des RPC, lassen die Stub-Komponente zu einem dynamischen Proxy werden. Wie in Abbildung 7.2 zu erkennen ist, implementiert jede

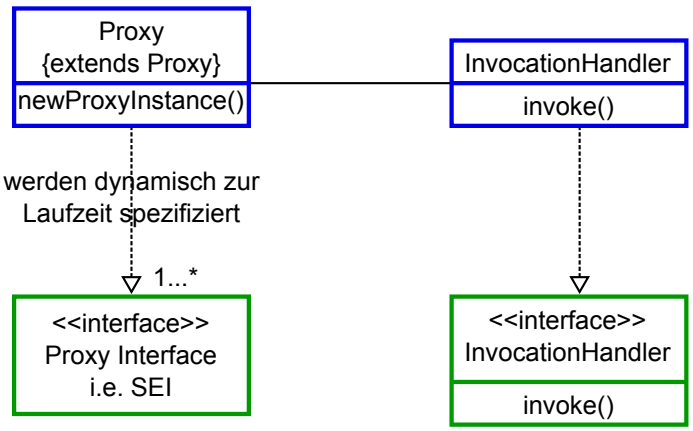

Abbildung 7.2: Dynamischer Proxy
Quelle: Eigene Darstellung

Instanz dieses dynamischen Proxies ein oder mehrere Interfaces (in diesem Zusammenhang auch als Proxy Interface bezeichnet) und besitzt genau einen Invocation Handler, welcher durch ein Objekt der Klasse `InvocationHandler` realisiert wird. Dies sind auch die Argumente, die der Methode `Proxy.newProxyInstance()` übergeben werden, um eine Proxy Instanz zu erzeugen. Als Interface wird dabei das SEI `PersonFinderPort` verwendet. Methodenaufrufe auf der Proxy Instanz, wie z.B. `getPerson()`, werden dabei nicht direkt in der Proxy Instanz ausgeführt, sondern an das zugehörige `InvocationHandler` Objekt delegiert. Dieses führt über

[2]Vgl. [Proxy].

die Methode `invoke()` dann die vom Proxy übergebene Methode aus. Letzteres geschieht über Reflection. Der Invocation Handler ist also das Herzstück des dynamischen Proxys und ist, im Fall von JAX-WS, für das Un-/Marshalling zwichen SOAP-Nachrichten und Java-Objekten zur Laufzeit zuständig.

Nutzung des Webservice-Endpunkts

Da die Generierung der benötigten Java-Artefakte aus der WSDL und das Erzeugen eines dynamischen Proxys komplett von JAX-WS übernommen wird, ist dies der einzige Teil, welcher vom Entwickler selbst geschrieben werden muss. Laut Abbildung 7.3 gibt es im generierten Fall zwei Möglichkeiten um eine Referenz auf einen konkreten Webservice-Endpunkt zu erhalten, welche man für die Kommunikation mit dem Serviceanbieter verwenden kann. Die augenscheinlichste

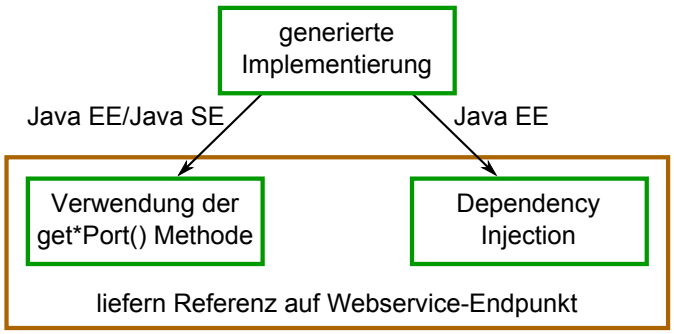

Abbildung 7.3: Generierte Implementierung eines JAX-WS Clients
Quelle: Eigene Darstellung

Möglichkeit ist hierbei die Verwendung der `getPersonFinderPort()` Methode aus der erzeugten Service-Klasse `PersonFinderService`. So erhält man mit folgenden zwei Zeilen eine Referenz auf den serverseitigen Person Finder Port und kann dessen `getPerson()` Operation nutzen. In Wirklichkeit versteckt sich hinter dieser Referenz jedoch eine Instanz des dynamischen Proxys, welcher die gesamte Kommunikation mit dem PISys-Server durchführt.

Listing 7.3: Generierter Client
runSyncGen() Methode aus dem PISys-Client

```
1 PersonInfoService service = new PersonInfoService();
2 PersonFinderPort port = service.getPersonFinderPort();
```

Arbeitet man in einem Application-Container, so kann auf die Möglichkeit der Dependency Injection zurückgegriffen werden. Dabei wird zur Laufzeit eine Instanz von einer benötigten Ressource in die annotierte Variable injiziert. In JAX-WS gibt es hierfür die von der `@Resource` abgeleitete `@WebServiceRef` Annotation. Mit dieser Annotation kann entweder eine Instanz einer Service-Klasse (in diesem

Fall der Klasse `PersonFinderService`) oder eine Instanz eines SEI der Serviceklasse (in diesem Fall des Interfaces `PersonFinderPort`) in die entsprechende Variable injiziert werden. Beide Möglichkeiten werden im folgenden Listing aufgezeigt.

Listing 7.4: Generierter Client über Dependency Injection

```
1 @WebServiceRef(value=PersonInfoService.class,
     type=PersonFinderPort.class)
2 public static PersonFinderPort port;
3
4 @WebServiceRef
5 public static PersonInfoService service;
```

Bei der ersten Möglichkeit (Injektion einer SEI Instanz) ist zwingend das Element `value` anzugeben. JAX-WS hat sonst keine Möglichkeit, die Service-Klasse zu finden. Das Element `type` hingegen ist optional und wird anhand der Klasse der annotierten Variablen bestimmt. Da ein Service jedoch mehrere Ports beinhalten kann, bietet sich die Angabe dieser Annotation an.
Optional kann zusätzlich noch der Ort der WSDL über das Element `wsdlLocation` angegeben werden. Dabei wird das gleichnamige Element in der `@WebService` Annotation des SEI überschrieben.

7.1.2 Manuelle Implementierung

Im vorherigen Abschnitt wurde auf die Möglichkeiten der generierten Implementierung eines Servicekonsumenten eingegangen. Dabei wurden alle benötigten Java-Artefakte vor der Laufzeit des PISys-Clients aus der WSDL generiert. Die generierte Implementierung geht jedoch mit zwei Nachteilen einher.

1. Die WSDL sollte sich zur Lebenszeit des Servicekonsumenten nicht ändern. Ändert sie sich doch, so müssen unter Umständen alle generierten Java-Klassen über den WSDL zu Java Generator erneut generiert und in das Projekt eingepflegt werden. Dieser Nachteil tritt insbesondere dann in den Vordergrund, wenn die WSDL sehr inkonsistent ist und laufend Änderungen (Neuerungen) unterworfen ist.

2. Gerade bei einer sehr kleinen WSDL ist der Einsatz des WSDL zu Java Generators „mit Kanonen auf Spatzen geschossen“. Hier würde eine manuelle Implementierung oftmals schneller und effizienter sein.

Um die obigen Nachteile auszugleichen, bietet JAX-WS die manuelle Implementierung eines Webservices an. Dabei wird der Webservice von Hand implementiert und auf die Nutzung von generierten Artefakten verzichtet. Wie in Abbildung 7.4 illustriert ist, basieren alle manuellen Implementierungen eines JAX-WS Clients auf dem `Dispatch<T>` Interface. In den folgenden Abschnitten wird das `Dispatch<T>` Interface zum einen dafür genutzt, die `getPerson()` Methode vom

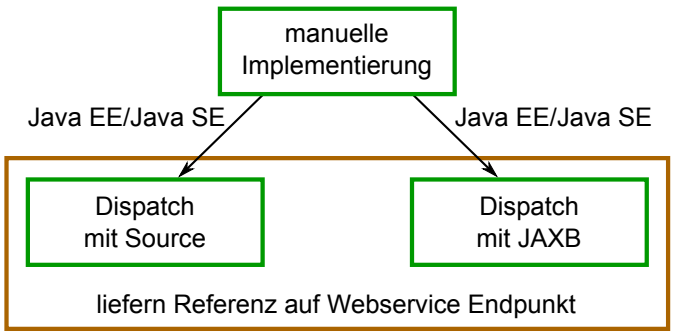

Abbildung 7.4: Manuelle Implementierung eines JAX-WS Clients
Quelle: Eigene Darstellung

Person Finder Port direkt über eine manuelle XML-Nachricht aufzurufen und zum anderen, diesen Aufruf über eine manuelle JAXB-Klasse abzusetzen.

Dispatch mit Source

In Listing 7.5 wird der manuelle Aufruf vom Person Finder Port anhand einer selbstgebauten XML-Nachricht aufgezeigt.

Listing 7.5: Manueller Client über Source
runSyncManLowLevel() Methode aus dem PISys-Client

```
1  Service service = Service.create(
2     new URL("http://localhost:8080/PersonInfoSystem/
          PersonFinderService?WSDL"),
3     new QName("http://pisys.de/pfs", "PersonInfoService"));
4  Dispatch<Source> dispatch = service.createDispatch(
5     new QName("http://pisys.de/pfs", "PersonFinderPort"),
          Source.class, Service.Mode.PAYLOAD);
6
7  /** XML Nachricht bauen */
8  DocumentBuilderFactory dbf =
       DocumentBuilderFactory.newInstance();
9  DocumentBuilder db = dbf.newDocumentBuilder();
10 Document doc = db.newDocument();
11 Element element = doc.createElementNS("http://pisys.de/pfs",
       "getPerson");
12 element.setNodeValue("Hans Meier");
13
14 /** Aufruf an den Webservice starten */
15 Source source = dispatch.invoke(new DOMSource(element));
```

Zunächst wird hierbei, wie im generierten Fall, eine Instanz der Service-Klasse erstellt. Die URL der WSDL kann dabei auch erst zur Laufzeit bestimmt wer-

den. Anders als im generierten Fall wird jedoch nicht die `getPersonFinderPort()` Methode der Service-Klasse aufgerufen um einen dynamischen Proxy vom Typ `PersonFinderPort` zu erzeugen. Stattdessen erhält man über einen Aufruf der Methode `createDispatch()` ein generisches `Dispatch` Objekt. In diesem Abschnitt ist dieses vom Typ `Source`. `Source` erlaubt die direkte Nutzung der von Java angebotenen Low-Level XML-APIs. Letztere werden in diesem Beispiel dazu verwendet, den Body einer SOAP-Nachricht zu generieren und diesen über `dispatch.invoke()` zu verschicken. Da JAX-WS in diesem Fall die Erstellung des SOAP Headers automatisch übernimmt, bleibt dem Programmierer die Arbeit erspart, diesen manuell zu definieren. Mit dem Attribut `Service.Mode.MESSAGE` der `createDispatch()` Methode hat der Programmierer dennoch die volle Kontrolle über die SOAP-Nachricht (inklusive Header). Interessanterweise kann das Ergebnis der `invoke()` Methode nur als `Source` oder `StAXSource` gecastet werden. Obwohl der Methode ein `DOMSource` Objekt übergeben wird, wirft der Cast zu `DOMSource` eine `ClassCastException`. Hierzu hätte eine Hinweis in der Spezifikation sicher nicht geschadet.
Obiges Listing deckt das typische Problem bei der manuellen Erstellung von XML-Nachrichten über die Low-Level Java APIs für XML (DOM, SAX und StAX) auf – es erfordert viel Quelltext um selbst einfache XML-Nachrichten zu generieren. Bei größeren XML-Nachrichten stößt man schnell an die Grenzen der Übersichtlichkeit und Wartbarkeit des erforderlichen Quelltextes. Für solche Fälle bietet JAX-WS abermals die Benutzung von JAXB an, diesmal im Kontext der manuellen Implementierung eines Servicekonsumenten.

Dispatch mit JAXB

Die manuelle Implementierung eines JAX-WS Servicekonsumenten mittels JAXB bietet sich an, wenn man genaue Informationen über den Aufbau der übertragenen XML-Nachrichten besitzt. In diesem Fall kann auf die Low-Level Java APIs für XML verzichtet werden und man bedient sich der Fähigkeiten von JAXB. Anders als bei der generierten Implementierung werden die JAXB-Klassen in diesem Fall nicht generiert, sondern von Hand implementiert. Obwohl JAXB recht großzügig mit fehlenden Annotationen umgeht, so gibt die Spezifikation keinen Aufschluss darüber, welche Annotationen an dieser Stelle zwingend für ein erfolgreiches Mapping erforderlich sind. Listing 7.6 zeigt die manuell erstellte Klasse `PFSResponse`, welche von JAXB für das Un-/Marshalling der XML-Nachrichten verwendet wird.

Listing 7.6: PFSResponse.java

```
1 @XmlRootElement(name = "getPersonResponse")
2 public class Response {
3     public int postleitzahl;
4     public List<String> interesse;
5     public Integer alter;
```

```
6     public XMLGregorianCalendar jahrgang;
7     public String familienstand;
8     @XmlAttribute
9     public Boolean gueltig;
10 }
```

Es ist zu erkennen, dass `@XmlRootElement` und `@XmlAttribute` in diesem Fall die einzigen benötigen Annotationen sind. Die `@XmlRootElement` Annotation dient JAXB als Einstiegspunkt für den Un-/Marshalling-Prozess und gibt an, dass der Inhalt dieser Klasse auf ein `<pfs:getPersonResponse>` Element abgebildet werden soll. Ohne die `@XmlAttribute` Annotation würde JAXB erfolglos versuchen, ein Element mit dem Namen `<pfs:gueltig>` auf diese Java-Variable abzubilden. Letztere würde also immer den Wert `null` annehmen.
Zusätzlich zu dem Aufbau der übertragenen XML-Nachrichten, benötigt es noch Informationen über die in der XML-Nachricht verwendeten Namensräume. Diese werden standardmäßig aus dem Paketnamen der jeweiligen JAXB-Klassen generiert. In diesem Fall wird nur ein Namensraum generiert, welcher jedoch von dem im Person Finder Service verwendeten Namensraum von `http://pisys.de/pfs/` abweicht. Da für das `<pfs:getPersonResponse>` Element demnach nur ein benutzerdefinierter Namensraum verwendet wird, bietet es sich an, diesen auf Paketebene zu definieren. Für letzteres sieht JAXB die Klasse `package-info` vor, welche sich im selben Paket wie die JAXB-Klasse `PFSResponse` befindet.
Unglücklicherweise wirft JAXB keine Fehler, wenn Elemente oder Attribute beim Un-/Marshalling-Prozess (z.B. aufgrund fehlender Namensrauminformationen) nicht abgebildet werden können. Da der Entwickler mit der manuell erstellten JAXB-Klasse unter Umständen jedoch nur einen Teil der Informationen aus der XML-Nachricht abgreifen will, ist dieser Ansatz durchaus verständlich.
Listing 7.7 zeigt schließlich den erforderlichen Quelltext zur Realisierung der manuellen Implementierung mittels JAXB.

Listing 7.7: Manueller Client über JAXB
runSyncManJAXB() Methode aus dem PISys-Client

```
1 JAXBContext jbctx =
    JAXBContext.newInstance(MyGetPersonResponse.class);
2 Service service = Service.create(
3     new URL("http://localhost:8080/PersonInfoSystem/
          PersonFinderService?WSDL"),
4     new QName("http://pisys.de/pfs/", "PersonInfoService"));
5 Dispatch<MyGetPersonResponse> dispatch =
    service.createDispatch(new QName("http://pisys.de/pfs/",
    "PersonFinderPort"), jbctx, Service.Mode.PAYLOAD);
6
7 JAXBElement<String> element = new JAXBElement<String>(
8     new QName("http://pisys.de/pfs/", "getPerson"),
```

```
9     String.class, null, "Hans Meier");
10 MyGetPersonResponse gpr = dispatch.invoke(element);
```

Zunächst muss der JAXB-Kontext erstellt werden. Dieser beinhaltet ein oder mehrere JAXB-Klassen, anhand derer JAXB den Un-/Marshalling-Prozess durchführt. In diesem Fall wird nur die Klasse `PFSResponse` für das Unmarshalling benötigt, da für das Marshalling eine Instanz von `JAXBElement` (entspricht der JAXB Repräsentation eines einfachen XML-Elements) zum Einsatz kommt. Letzteres repräsentiert den Input für den Webservice und stellt das `<pfs:getPerson>` Element aus der PFS-WSDL dar. Aufgerufen wird der Webservice schließlich über die `invoke()` Methode. Diese führt das Unmarshalling der eintreffenden XML-Nachricht durch und liefert die Ergebnisse in einer Instanz der Klasse `PFSResponse` zurück.

7.2 Asynchrone Implementierung

Ein Webservice muss bei einer Anfrage oft komplizierte Berechnungen oder komplexe Datenbankabfragen durchführen. Im synchronen Fall würde der Aufrufer diese Zeitspanne abwarten um anschließend die vom Webservice zurückgelieferten Ergebnisse zu präsentieren. Da diese Zeitspanne jedoch mehrere Minuten umfassen kann, vergeudet der Aufrufer durch diese Wartezeit sehr viel Rechenleistung. Diese könnte anderweitig besser genutzt werden, wenn die Anfrage in einem eigenen Thread gestartet wird. Hierfür bietet Java von Haus aus die Klasse `Thread` bzw. das Interface `Runnable` an. Verwendet man diese in dieser Urform, so treten einige Nachteile ans Tageslicht.

- Durch Implementierung der `run()` Methode und dem Starten des Threads entsteht viel Overhead im Quelltext.
- Threads sind aufgrund von Performancevorteilen von Haus aus nicht synchronisiert. Gerade wenn viele Benutzer an einem Servicekonsumenten arbeiten und Anfragen abschicken ist die effiziente Synchronisierung dieser nebenläufigen Prozesse Pflicht. Eine effiziente Synchronisierung erfordert jedoch einiges an Programmieraufwand.
- Analog zu obigem Punkt muss sich der Entwickler generell um alle Low-Level Details der Threadprogrammierung kümmern. Dazu gehören auch die sinnvolle Beendigung der Threads, Setzen von Prioritäten, Verwalten von Threadgruppen, usw.

In JAX-WS wird diese Komplexität unter zwei unterschiedlichen Modellen der asynchronen Implementierung versteckt, dem *Polling-Modell* und dem *Callback-Modell*. Beide Modelle können sowohl mit der generierten Implementierung, als

auch mit der manuellen Implementierung über das `Dispatch<T>` Interface kombiniert werden. Im Folgenden werden beide Modelle für beide Implementierungsmöglichkeiten angeschnitten.

7.2.1 Polling-Modell

Unter Polling wird allgemein das aktive Warten auf ein bestimmtes Objekt verstanden. Innerhalb des aktiven Wartens werden zyklische Anfragen an das Objekt verschickt um dessen Status auszulesen. Liefert dieser Status den gesuchten Wert, kann im Anschluss der Zustand des Objekts ausgelesen werden.
Wie in Abbildung 7.5 zu erkennen ist, basiert das Polling-Modell unter JAX-WS ebenfalls auf dieser Beschreibung. Bei dem Objekt handelt es sich in diesem Fall

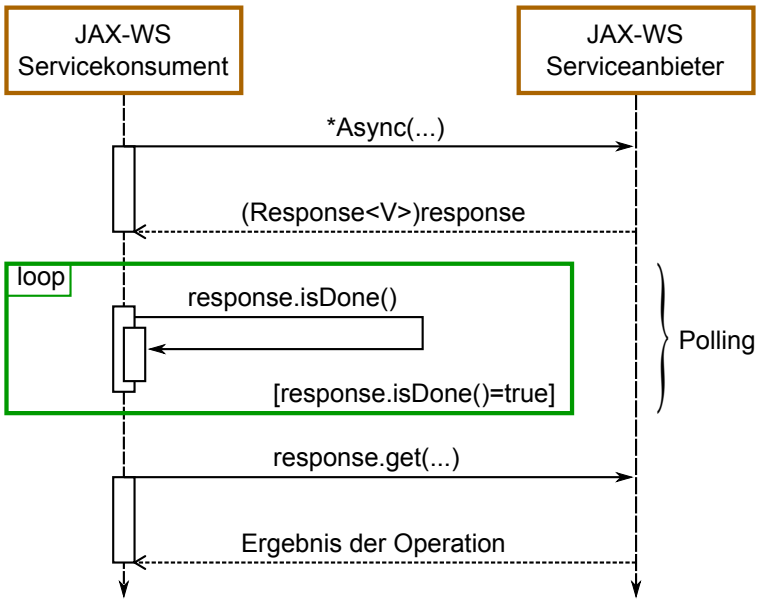

Abbildung 7.5: Polling-Modell in JAX-WS
Quelle: Eigene Darstellung

um das zurückgelieferte Resultat des Serviceanbieters. Auf der technischen Seite beruht das Polling-Modell von JAX-WS auf dem komfortablen `Response<T>` Interface. Dieses ist eine minimale Erweiterung vom `Future<V>` Interface mit der zusätzlichen Methode `getContext()` um den Kontext der XML-Nachricht zu erhalten. Das `Response<T>` Interface ist ein generischer Platzhalter für das zukünftige Ergebnis eines asynchronen Aufrufs. In diesem Sinn wird die `Response<T>` Instanz erst zu späterer Zeit mit dem Ergebnis des Aufrufs einer Operation des JAX-WS Servers befüllt. Wann dieses Ergebnis eingetroffen ist, erfährt der Aufrufende über Polling der `Response<T>` Instanz. Liefert die Methode `isDone()` den Wert `true`, so ist das Ergebnis eingetroffen und kann über die `get()` Methode

abgeholt werden. Die Zeit zwischen der Initialisierung der `Response<T>` Instanz und dem Abholen der Ergebnisse über die `get()` Methode ist nicht blockierend. Der Aufrufer kann diese Zeitspanne also für andere Aufgaben verwenden. In den folgenden beiden Abschnitten wird die Verwendung von `Response<T>` innerhalb der generierten und der manuellen Implementierung aufgezeigt.

Das Polling-Modell bei manueller Implementierung

JAX-WS hat das Polling-Modell bei der manuellen Implementierung über die `invokeAsync()` Methoden vom `Dispatch` Interface realisiert. Analog zu dem synchronen Beispiel aus Listing 7.5 wird der Aufruf in Listing 7.8 über die gleiche DOMSource Instanz gestartet. Unterschiede sind der Aufruf der `invokeAsync()` Methode anstelle der `invoke()` Methode und das Polling der zurückgelieferten `Resource<T>` Instanz.

Listing 7.8: Polling-Modell bei manueller Implementierung
runAsyncManLowLevelPolling() Methode aus dem PISys-Client

```
1 Response<Source> response = dispatch.invokeAsync(
2    new DOMSource(element));
3 while (!response.isDone()) {
4     /** Erledige andere Dinge */
5 }
6 Source sourceAsync = response.get();
```

Die `invokeAsync()` Methode ist an dieser Stelle für die asynchrone Ausführung der Anfrage zuständig. Indirekt erzeugt diese Methode eine Executor Instanz, welche wiederum für das Erstellen und Verwalten eines Threads zuständig ist.

Das Polling-Modell bei generierter Implementierung

In dem generierten Fall ist die asynchrone Nutzung eines Serviceanbieters mit etwas mehr Aufwand verbunden. Synchrone Aufrufe werden, wie in Listing 7.3 gezeigt wurde, über die aus der WSDL generierten Methoden eines Ports ausgeführt. Leider besitzt die Klasse `PersonFinderPort` aber nur die synchrone Methode `getPerson()`. JAX-WS geht davon aus, dass asynchrone Aufrufe nur in wenigen und nicht allen Fällen nützlich sind und überlässt deren optionale Erstellung dem Benutzer. Dabei werden die asynchronen Methoden, ebenso wie ihre synchronen Abbilder, vor Laufzeit aus der WSDL generiert.
Um die Erstellung asynchroner Methoden zu aktivieren wird Gebrauch der aus dem Kapitel 6.3.3 bekannten Binding Language Declarations gemacht. Diese sind an dieser Stelle ebenfalls wieder in einer separaten Datei definiert und referenzieren die WSDL über XPath. Listing 7.9 zeigt die benötigte Binding-Datei, die den WSDL zu Java Generator veranlasst, eine asynchrone Version der `getPerson()` Methode zu generieren.

Listing 7.9: async-bindings.xml

```
<bindings> <!-- Namensraeume und wsdlLocation -->
    <bindings
       node="//wsdl:portType[@name='PersonFinderPort']">
        <bindings node="wsdl:operation[@name='getPerson']">
            <enableAsyncMapping>true</enableAsyncMapping>
        </bindings>
    </bindings>
</bindings>
```

Ist der Wert vom `<jaxws:enableAsyncMapping>` Element `true`, so generiert JAX-WS passend zu der referenzierten Operation eine asynchrone Java-Methode. Diese Methode trägt immer das Schlüsselwort `Async` hinter dem Methodennamen. Listing 7.10 zeigt also den asynchronen Aufruf der `getPerson()` Methode, welche in diesem Fall den Namen `getPersonAsync()` trägt. Das Polling der zurückgelieferten `Response<T>` Instanz verhält sich wie im manuellen Fall.

Listing 7.10: Polling-Modell bei generierter Implementierung runAsyncGenPolling() Methode aus dem PISys-Client

```
PersonFinderPort port = (new PersonInfoService()).
   getPersonFinderPort();
Response<GetPersonResponse> response =
   port.getPersonAsync("Hans Meier");
while (!response.isDone()) {
    /** Erledige andere Dinge */
}
GetPersonResponse person = response.get();
```

7.2.2 Callback-Modell

Im vorigen Abschnitt wurde das Polling-Modell eines asynchronen Aufrufs von JAX-WS behandelt. Dieser Abschnitt widmet sich nun der anderen Seite des asynchronen Aufrufs, dem Callback-Modell. Das Callback-Modell ist eine Instanz des *Beobachter Entwurfsmusters* (engl. Observer), welches in Java typischerweise über Listener implementiert ist. Interessiert sich ein Objekt für den Zustand eines anderen Objekts, so implementiert dieses Beobachter-Objekt ein von der Klasse des zweiten Objekts zugehöriges Listener-Interface und meldet sich dadurch beim zweiten Objekt, dem beobachteten Objekt, an. Bei bestimmten Zustandsänderungen des beobachteten Objekts werden nun durch das implementierte Interface ein oder mehrere Methoden des Beobachter-Objekts automatisch ausgeführt. Ein Polling des beobachteten Objekts ist also nicht mehr notwendig, kann in JAX-WS jedoch optional durchgeführt werden. Abbildung 7.6 skizziert den schemenhaften Ablauf des Callback-Modells unter JAX-WS. Dieses wird demnach durch das

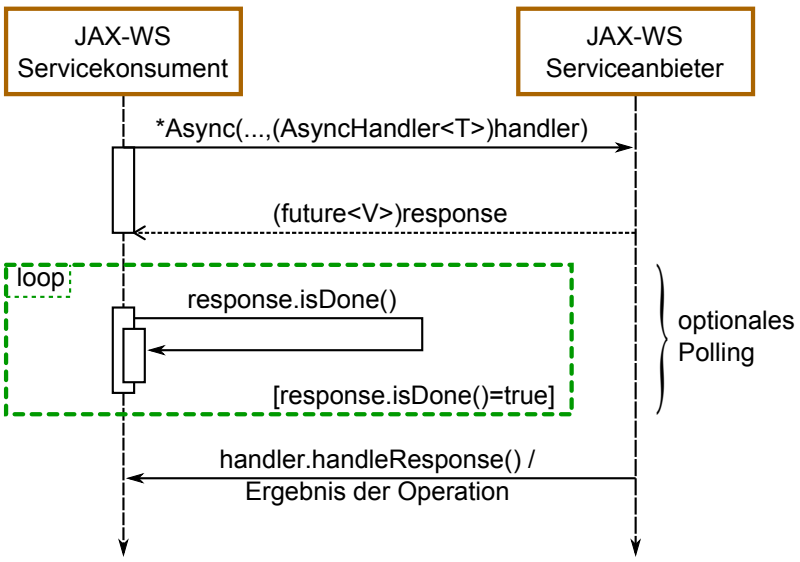

Abbildung 7.6: Callback-Modell in JAX-WS
Quelle: Eigene Darstellung

`AsyncHandler<T>` Interface realisiert. Zunächst wird ein Objekt einer Implementierung dieses Interfaces dem asynchronen Aufruf übergeben. Beendet sich der Aufruf, so wird die `handleResponse()` Methode dieses Objekts ausgeführt. Auf der Implementierungsseite ähnelt dieser Ansatz demnach, bis auf die Übergabe des `AsyncHandler<T>` Objekts, sehr dem des Polling-Modells. Listing 7.11 zeigt eine beispielhafte Implementierung dieses `AsyncHandler<T>` Interfaces.

Listing 7.11: PFSAsyncHandler.java (implementiert AsyncHandler<T>)

```
1  public class PISAsyncHandler implements AsyncHandler {
2
3      public PISAsyncHandler() {}
4
5      public void handleResponse(Response response) {
6          Object object = (Object) response.get();
7          if (object instanceof GetPersonResponse) {
8          /** Verarbeite GetPersonResponse Objekt */
9          } else if (object instanceof Source) {
10         /** Verarbeite Source Objekt */
11         }
12     }
13 }
```

In diesem Listing ist zu erkennen, dass der Methode `handleResponse()` eine Instanz vom Typ `Response<T>` übergeben wird. Entgegen den Empfehlungen der JAX-WS Spezifikation, werden `AsyncHandler<T>` und `Response<T>` an dieser Stel-

le nicht typisiert, da sowohl `Source`, als auch `GetPersonResponse` Objekte erwartet werden. Somit ist dieser `AsyncHandler` sowohl im generierten, als auch im manuellen Fall anwendbar, der PISys-Client allerdings weniger portabel.
Bei der manuellen Implementierung wird nun eine Version der `invokeAsync()` Methode vom `Dispatch<T>` Interface verwendet, der ein `AsyncHandler<T>` Objekt übergeben werden kann. Da der Quelltext bis zu dem Aufruf dieser Methode dem des Polling-Modells entspricht, enthält Listing 7.12 nur die Zeile des Aufrufs.

Listing 7.12: Callback-Modell bei manueller Implementierung
runAsyncManLowLevelCallback() Methode aus dem PISys-Client

```
1 dispatch.invokeAsync(new DOMSource(element),
2    new PISAsyncHandler(this));
```

Synonym hierzu funktioniert die Übergabe eines `AsyncHandler<T>` Objekts bei der generierten Implementierung.

Listing 7.13: Callback-Modell bei generierter Implementierung
runAsyncGenCallback() Methode aus dem PISys-Client

```
1 port.getPersonAsync("Hans Meier",
2    new PISAsyncHandler(this));
```

Hierbei ist jedoch zu beachten, dass die asynchrone Methode `getPersonAsync()`, genau wie bei dem Polling-Modell, erst durch die `<jaxws:enableAsyncMapping>` Binding Language Declaration über die WSDL erzeugt werden muss.

7.3 Zusammenfassung und Fazit

Dieses Kapitel widmete sich der clientseitigen Implementierung eines Webservices unter JAX-WS. Im direkten Vergleich mit der JAX-RPC Spezifikation fallen insbesondere die asynchronen Möglichkeiten der Implementierung auf, welche von der JAX-RPC Spezifikation nicht vorgeschrieben sind. Letzteres ist in Anbetracht des typischen Einsatzgebietes einer Technologie wie JAX-WS oder JAX-RPC, der Realisierung von großen Webservices mit oftmals teuren Operationen seitens des Serviceanbieters, nicht verständlich und ein großer Fortschritt in der Java Webservice Welt.
Unter Berücksichtigung der in Kapitel 6.1 aufgeführten Szenarien auf die ein Entwickler bei der Entwicklung eines Webservices trifft, lässt sich folgende Beurteilung für die clientseitige Implementierung eines Webservices unter JAX-WS aufstellen.

- „Start von WSDL“ – Dieses Szenario wird bei der initialen Entwicklung eines Servicekonsumenten am häufigsten anzutreffen sein und ist, ähnlich wie bei der Bewertung von JAXB, ein Vertreter der „grünen Wiese“. Je

nach Größe und Konsistenz der WSDL bietet JAX-WS entweder die generierte oder die manuelle Implementierung des Servicekonsumenten an. In Bezug auf die generierte Implementierung sorgt, ähnlich wie Schema Compiler von JAXB, ein WSDL zu Java Generator für die Erzeugung der benötigten Java-Artefakte aus der WSDL. Laut Abschnitt 7.1.1 können diese Artefakte genutzt werden, um in nur zwei Zeilen eine Referenz auf den Webservice-Endpunkt eines Serviceanbieters zu erhalten. Der Entwickler kann dadurch direkt die vom Webservice-Endpunkt angebotenen Operationen nutzen und muss sich weder um die Erstellung eines dynamischen Proxys für die Realisierung der Kommunikation mit dem Serviceanbieter, noch um den Un-/Marshalling-Prozess zwischen Java-Objekten und SOAP-Nachrichten kümmern. Dies ist ein gewaltiger Vorteil von JAX-WS und vereinfacht die Entwicklung eines Servicekonsumenten erheblich. Hat man es mit einer relativ kleinen und/oder inkonsistenten WSDL zu tun, so bietet JAX-WS die manuelle Implementierung an. Diese ist im Vergleich zu der generierten Implementierung ein wenig aufwendiger, da sich der Entwickler selbst um den korrekten Aufbau der Anfragen und dem Auslesen der Antworten kümmern muss. Wie Abschnitt 7.1.2 gezeigt hat gibt es für diesen Fall, außer der Verwendung der Low-Level Java APIs für XML, auch die Möglichkeit der Nutzung von JAXB. Die JAXB-Klassen müssen in diesem Fall verständlicherweise manuell erstellt werden. Die asynchrone Kommunikation kann im generierten Fall über eine externe Binding-Datei optional bei der Erstellung der Java-Artefakte aus der WSDL generiert werden. Im manuellen Fall kann einfach eine der `invokeAsync()` Methoden verwendet werden.

- „Start von Java“ – Typischerweise wird ein Webservice nicht von der Client-Seite aus implementiert. JAX-WS geht davon aus, dass für die Implementierung eines Servicekonsumenten immer eine WSDL existieren muss und bietet für dieses Szenario demnach keine Unterstützung an. Es bleibt die manuelle Erstellung der WSDL.

- „Start von WSDL & Java“ – Dieses Szenario muss von zwei Sichtweisen betrachtet werden und ist nicht trivial zu lösen. Ist der Webservice bereits in dem Java Projekt implementiert und hat sich nur dessen WSDL geändert, so kann über die vielfältigen JAX-WS und JAXB Annotationen versucht werden, wieder eine Homogenität herzustellen. Dies hat allerdings die erneute Übersetzung der Quelltexte zufolge. Soll der Webservice initial in das bestehende Java Projekt integriert werden, so ähnelt dieses Szenario dem „Start von WSDL“ Szenario.

8 JAX-WS Server

Dieses Kapitel beschreibt die Möglichkeiten der serverseitigen Implementierung eines Webservices unter JAX-WS. In Abbildung 8.1 ist zu erkennen, dass im Vergleich zu den im vorigen Kapitel aufgeführten clientseitigen Implementierungsmöglichkeiten, JAX-WS für die serverseitige Implementierung ein weniger reichhaltiges Spektrum dieser anbietet. Dies liegt daran, dass eine serverseitige Im-

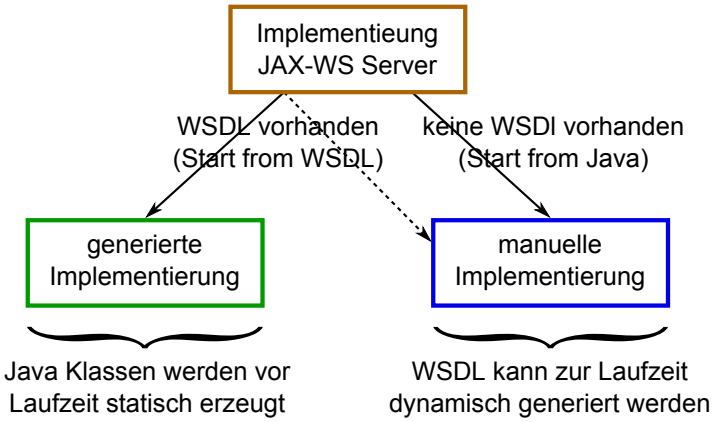

Abbildung 8.1: Implementierung eines JAX-WS Servers
Quelle: Eigene Darstellung

plementierung keine synchronen Aufrufe durchführt. So wird jeder Aufruf einer Operation des Webservice-Endpunktes in einem eigenen Thread gestartet. Da ein Serviceanbieter typischerweise Anfragen von mehreren Servicekonsumenten bearbeiten muss und es dabei sehr schnell zu sich überlappenden Anfragen kommt, ist diese Entscheidung durchaus nachvollziehbar. Im Folgenden werden also die generierten und manuellen Möglichkeiten der Entwicklung eines JAX-WS Servers aufgezeigt. Die generierte Implementierung bezieht sich dabei, wie auch im clientseitigen Fall, auf das Vorhandensein einer WSDL, aus der vor Laufzeit des JAX-WS Servers alle benötigten Java-Artefakte generiert werden. Die manuelle Implementierung erfolgt wiederum ohne Zuhilfenahme dieser Artefakte.

8.1 Generierte Implementierung

Die generierte Implementierung spiegelt in diesem Fall die serverseitige Auslegung des „Start von WSDL“ Szenarios wieder. Zwar kann laut Abbildung 8.1

dieses Szenario auch über eine manuelle Implementierung realisiert werden, dieser Fall ist in Hinblick auf die hervorragenden Binding-Fähigkeiten von JAXB jedoch eher vernachlässigbar.
Für die Generierung der Java-Artefakte aus der PFS-WSDL wird wieder der WSDL zu Java Generator wsimport verwendet, welcher die aus Kapitel 7.1.1 bekannten Dateien package-info.java, ObjectFactory.java, GetPersonResponse.java, PersonFinderService.java und PersonFinderPort.java erstellt. Bei der Erstellung eines JAX-WS Clients wurde nun die Methode `getPersonFinderPort()` der Service-Klasse `PersonFinderService` verwendet um eine Instanz des SEI `PersonFinderPort` als Referenz auf einen Webservice-Endpunkt des Serviceanbieters zu erhalten. Dieses SEI muss natürlich serverseitig implementiert sein. Listing 8.1 zeigt demnach die serverseitige Implementierung des SEI `PersonFinderPort` mit Hilfe der zuvor über das wscompile Tool erzeugten Artefakte aus der PFS-WSDL.

Listing 8.1: Generierter Server
gen Paket aus dem PISys-Server

```
1  @WebService(serviceName = "PersonFinderService",
2          portName = "PersonFinderPort",
3          endpointInterface = "de.pisys.pfs.PersonFinderPort",
4          targetNamespace = "http://pisys.de/pfs/",
5          wsdlLocation = "pfs.wsdl")
6  public class Server implements PersonFinderPort {
7
8      public GetPersonResponse getPerson(String name) {
9          ObjectFactory of = new ObjectFactory();
10         GetPersonResponse response =
               of.createGetPersonResponse();
11
12         /* Befülle response */
13         return response;
14     }
15 }
```

Wie beim SEI trägt die Implementierung ebenfalls die `@WebService` Annotation. Letztere besitzt allerdings zusätzliche Elemente, welche unter anderem den zu implementierenden Port (Webservice-Endpunkt) innerhalb der WSDL spezifizieren (eine WSDL kann mehrere Ports und Bindings enthalten) sowie den Ort der WSDL angeben.
Ist der Webservice-Endpunkt, in diesem Fall der Person Finder Port, implementiert, so muss dieser für die Nutzung durch einen Servicekonsumenten bereit gestellt werden. Der Aufwand dieser Bereitstellung variiert je nach der verwendeten Java Plattform Edition. In Java EE würde der Application Container anhand der `@WebService` Annotation die Bereitstellung des Webservice-Endpunkts automatisch durchführen.

JAX-WS kann seit der Mustang Version (Version 6) der Java SE einen Webservice auch unter der Java SE Plattform Edition veröffentlichen. Dies ist ein gewaltiger Fortschritt gegenüber der JAX-RPC Technologie, in der ein Serviceanbieter zwingend in einem Servlet oder einer EJB implementiert werden musste. Listing 8.2 zeigt den Quelltext, welcher zur Veröffentlichung der Implementierung des `PersonFinderPort` SEI unter Java SE erforderlich ist.

Listing 8.2: server-endpoint.java

```
1 Endpoint endpoint = Endpoint.create(new PersonFinderPort());
2 endpoint.publish("http://localhost:8080/PersonInfoSystem
    PersonFinderService");
```

In diesem Listing ist zu erkennen, dass obiger Fortschritt über die `Endpoint` Klasse realisiert ist. Durch die Benutzung dieser Klasse kann also in nur zwei Zeilen ein funktionierender Webservice-Endpunkt unter Java SE erzeugt und veröffentlicht werden. Die Klasse des Objekts, welche der Methode `create()` übergeben wird, muss zwingend entweder die `@Webservice` oder die `@WebServiceProvider` Annotation besitzen. Die `@WebServiceProvider` wird dabei für die Low-Level Implementierung eines JAX-WS Servers verwendet. Da die Veröffentlichung des PISys-Servers unter allen durchgeführten Implementierungen aus den zwei in Listing 8.2 dargestellten Zeilen besteht, werden diese in den folgenden Abschnitten dieses Kapitels nicht mehrfach dargestellt.
Die in diesem Abschnitt aufgeführte Implementierung eines Serviceanbieters anhand einer WSDL ist in der Realität eher unüblich. Typischerweise wird ein Serviceanbieter nach dem „Start von Java“ Szenario von Java aus implementiert. Verwendet man dabei nicht die Low-Level Java APIs für XML sondern JAXB, dann kann die WSDL von der JAX-WS Laufzeitumgebung anhand der Klassenstruktur und der aufgeführten Annotationen dynamisch generiert werden. Beide Ansätze werden im folgenden Abschnitt behandelt.

8.2 Manuelle Implementierung

Abbildung 8.2 illustriert die Möglichkeiten der manuellen Implementierung eines JAX-WS Servers. Diese wird insbesondere dann durchgeführt, wenn der JAX-WS Server ohne dem Vorhandensein einer WSDL implementiert wird, also in einem strikten „Start von Java“ Szenario.
Dies ist ein Unterschied zu der clientseitigen manuellen Implementierung, die ohne eine WSDL weder manuell, noch generiert durchgeführt werden kann. Dennoch muss ein in JAX-WS implementierter Serviceanbieter eine WSDL zur Verfügung stellen. Glücklicherweise wird diese von der JAX-WS Laufzeitumgebung automatisch erstellt. Diesen Vorteil bietet nach Abbildung 8.2 allerdings nur die manuelle High-Level Implementierung über JAXB an, welche im folgenden Abschnitt

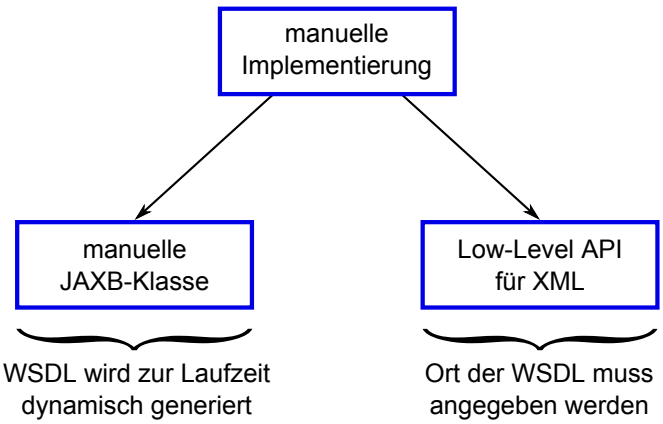

Abbildung 8.2: Manuelle Implementierung eines JAX-WS Servers
Quelle: Eigene Darstellung

behandelt wird. Bei der Low-Level Implementierung muss explizit eine WSDL angegeben werden, die den veröffentlichenden Webservice beschreibt.

8.2.1 High-Level Implementierung über JAXB

In diesem Abschnitt wird die manuelle High-Level Implementierung eines JAX-WS Servers ausgehend von Java beschrieben. Diese basiert hauptsächlich auf der Verwendung der `@WebService` Annotation und der impliziten Nutzung von JAXB. Ziel dieses Abschnitts ist die minimale Implementierung des Person Finder Services anhand manuell erstellter Java-Klassen und die Bewertung dieser Implementierung. Minimal bedeutet dabei die Restriktion auf so wenig Annotationen wie möglich. Dadurch wird zur Laufzeit zwar nicht die PFS-WSDL aus Listing 4.1 generiert, es spiegelt jedoch den minimalen Aufwand einer solchen Implementierung wieder und zeigt, welche standardmäßigen Werte JAX-WS für die Namensräume, den WSDL/SOAP-Style usw. annimmt. Auf der anderen Seite kann durch die JAX-WS und JAXB Annotationen nie der Aufbau der ursprünglichen PFS-WSDL erreicht werden. Es fehlen zumindest die über JAXB derzeit nicht in Java umsetzbaren XML-Schema Restriktionen.
Listing 8.3 zeigt die Implementierung des manuell erstellen Person Finder Ports, welcher eine manuell erstellte JAXB-Klasse nutzt. Da erstere Klasse die gesamte Geschäftslogik implementiert, wird diese auch als *Service Implementation Bean (SIB)* bezeichnet.

Listing 8.3: Manueller Server über JAXB
jaxb Paket aus dem PISys-Server

```
1 @WebService(targetNamespace = "http://pisys.de/pfs/")
2 @SOAPBinding(parameterStyle =
    SOAPBinding.ParameterStyle.BARE)
```

```
3 public class PersonFinderPort {
4     public PFSResponse getPerson(String name) {
5         PFSResponse response = new PFSResponse();
6             /** Befülle response */
7             return response;
8         }
9     }
10 }
```

In diesem minimalen Beispiel reicht die `@WebService` Annotation leider nicht aus um die Klasse als funktionierenden Webservice-Endpunkt des PISys-Servers zu definieren. Da JAX-WS standardmäßig ein weiteres Wrapper-Element um das `<pfs:getPersonResponse>` Element legt, ist die `@SOAPBinding` Annotation nötig, welche in diesem Fall den Parameter-Style auf `BARE` (Unwrapped) setzt. Ohne diese Annotation würde der PISys-Server die XML-Nachrichten in einem falschen Format generieren, so dass diese vom PISys-Client nicht gelesen werden könnten. Für den Un-/Marshalling-Prozess des `<pfs:getPersonResponse>` Elements kommt die aus Listing 7.6 bekannte JAXB-Klasse `PFSResponse` zum Einsatz, welche ebenfalls nur zwei Annotationen benötigt.
Bei der Veröffentlichung des so implementierten Person Finder Ports über die `Endpoint` Klasse, generiert die JAX-WS Laufzeitumgebung nur über die Angabe der `@WebService` Annotation alle für die Bereitstellung der PISys-Servers benötigten Artefakte. Dazu gehört insbesondere die dynamisch generierte WSDL, welche unter `http://localhost:8080/PersonInfoSystem/PersonFinderService?WSDL` abgelegt wird. Listing 8.4 zeigt diese WSDL, wobei aus Gründen der Übersichtlichkeit die `<wsdl:types>` Sektion entfernt wurde, da diese ausschließlich von JAXB beeinflusst wird.

Listing 8.4: Generierte Version der pfs.wsdl

```
1 <definitions  xmlns:tns="http://pisys.de/pfs/"
    targetNamespace="http://pisys.de/pfs/"
    name="PersonFinderService">
2     <message name="getPerson">
3         <part element="tns:getPerson" name="getPerson"/>
4     </message>
5     <message name="getPersonResponse">
6         <part element="tns:getPersonResponse"
              name="getPersonResponse"/>
7     </message>
8     <portType name="PersonFinderPort">
9         <operation name="getPerson">
10            <input message="tns:getPerson"/>
11            <output message="tns:getPersonResponse"/>
12        </operation>
13    </portType>
```

```
14      <binding name="PersonFinderPortBinding"
           type="tns:PersonFinder">
15          <soap:binding style="document"
               transport="http://schemas.xmlsoap.org/soap/http"/>
16          <operation name="getPerson">
17              <soap:operation soapAction=""/>
18              <input>
19                  <soap:body use="literal"/>
20              </input>
21              <output>
22                  <soap:body use="literal"/>
23              </output>
24          </operation>
25      </binding>
26      <service name="PersonFinderService">
27          <port name="PersonFinderPort"
               binding="tns:PersonFinderPortBinding">
28              <soap:address location="http://localhost:8080/
                   PersonInfoSystem/PersonFinderService"/>
29          </port>
30      </service>
31  </definitions>
```

Diese WSDL beschreibt genau den gleichen Webservice wie die manuell erstellte PFS-WSDL. Man kann sich vorstellen, dass durch weitere Annotationen die PFS-WSDL bis auf die `<wsdl:types>` Sektion eins zu eins nachgebildet werden kann. Die nicht umsetzbaren XML-Schema Restriktionen könnten noch nachträglich in die Datentypdefinitionen der `<wsdl:types>` Sektion eingepflegt werden.
An dieser Stelle ist anzumerken, dass die dynamische Generierung der WSDL ein gewaltiger Vorteil der JAX-WS Technologie ist und das „Start von Java“ Szenario erheblich erleichtert. Zumal diese Generierung, wie dieser Abschnitt gezeigt hat, einen sehr ausgereiften Eindruck hinterlässt.

8.2.2 Low-Level Implementierung

Anders als bei der im vorigen Abschnitt gezeigten High-Level Implementierung über JAXB, kann ein Webservice-Endpunkt auch über die Verwendung der Low-Level Java APIs für XML umgesetzt werden. Bei dieser Implementierung entsteht jedoch implizit ein erheblicher Nachteil - JAX-WS kann über den Quelltext der Low-Level APIs für XML keine WSDL generieren. In diesem Zusammenhang findet sich in der JAX-WS Spezifikation der Hinweis, dass

> A WSDL file is required to be packaged with a Provider implementation.[1]

[1] S. [JSR109], Abschnitt 5.3.2.2.

Diese Forderung ist jedoch nachvollziehbar, da ein Servicekonsument im Normalfall nur über die WSDL Informationen über die vom Serviceanbieter angebotenen Operationen erhält. Dadurch wird die Low-Level Implementierung insbesondere im „Start von WSDL & Java“ Szenario interessant, da dort bereits eine WSDL vorhanden ist. Außerdem eignet sich diese Form der Implementierung aufgrund der Größe des erforderlichen Quelltextes nur für kleinere Serviceanbieter (vgl. Kapitel 7.1.2).
In JAX-WS wird die oben beschriebene Low-Level Implementierung über das `Provider<T>` Interface realisiert. Jede Anfrage ruft die in diesem Interface definierte Methode `invoke()` auf, welche über ein simples `return` das Ergebnis an den Aufrufer zurückliefert. Der Übergabe- und Rückgabewert dieser Methode wird über das Interface `Provider<T>` typisiert. Mögliche Werte für `T` sind `SOAPMessage`, `DataSource` und die bereits in Kapitel 7.1.2 verwendete Klasse `Source`. Listing 8.5 zeigt die zum vorigem Kapitel äquivalente Low-Level Implementierung des Person Finder Ports unter Verwendung der Klasse `SOAPMessage`.

Listing 8.5: Manueller Server über SOAPMessage
lowlevel Paket aus dem PISys-Server

```
1  @WebServiceProvider(portName = "PersonFinderPort",
       wsdlLocation = "de/pisys/pfs/pfs.wsdl",
2      serviceName = "PersonFinderService", targetNamespace =
         "http://pisys.de/pfs/")
3  @ServiceMode(Service.Mode.MESSAGE)
4  public class PersonFinderPort implements
       Provider<SOAPMessage> {
5
6      public SOAPMessage invoke(SOAPMessage request) {
7          Node getPersonNode = (Node)
               request.getSOAPBody().getChildElements().next();
8          if (getPersonNode.getTextContent().equals("Hans
               Meier")) {
9              SOAPMessage response =
                   MessageFactory.newInstance().createMessage();
10             /* befülle response */
11             return response;
12         }
13         return null;
14     }
15 }
```

Da JAX-WS aus dieser Implementierung verständlicherweise keine WSDL generieren kann, muss über das `wsdlLocation` Element dieser Annotation der Ort der WSDL angegeben werden. Die Veröffentlichung dieser SEI Implementierung geschieht wieder über die in Listing 8.2 aufgezeigte Verwendung der `Endpoint` Klasse.

8.3 Laufzeitverhalten

In diesem und im vorigen Kapitel wurden insgesamt sieben clientseitige und drei serverseitige Implementierungen des Person Info Systems aufgezeigt. Besonderes Augenmerk lag dabei in der Bewertung des erforderlichen Aufwands einer solchen Implementierung und dessen Einordnung in ein bestimmtes Webservice-Szenario. Diese Punkte geben jedoch keinen Aufschluss über das effektive Laufzeitverhalten einer solchen Implementierung.
Generell gestaltet es sich schwierig, die Güte einer Webservice-Implementierung anhand des Laufzeitverhaltens zu bestimmen, da letzteres von vielen anderen Faktoren abhängig ist. So beeinflussen die Übertragungsgeschwindigkeit im Internet und die im Webservice umgesetzte Geschäftslogik, welche mitunter komplexe Datenbankabfragen durchführen kann, maßgeblich die Zeitspanne vom Aufruf einer Methode des Webservice-Endpunks bis zur Beendigung dieser. In Bezug auf die Webservice-Implementierung nimmt der Un-/Marshalling-Prozess der XML-Nachrichten die meiste Zeit in Anspruch. Daher wird in diesem Abschnitt eher die Güte des Un-/Marshalling-Prozesses der jeweiligen Webservice-Implementierung aufgezeigt.
Tabelle 8.1 illustriert die benötigte Zeitspanne der clientseitigen Aufrufe an die drei verschiedenen Implementierungen des PISys-Servers. Für diesen Zweck wird die im PISys-Client eingebaute Lasttest-Funktion verwendet, welche den jeweiligen Aufruf genau hundertmal absetzt und die dabei benötigte Zeit (in ms) misst und zur Anzeige bringt.[2] Dabei ist auffallend, dass zwischen den drei server-

Client \Server	gen	jaxb	lowlevel
SyncGen	1987	1759	2062
SyncManLowLevel	2002	2063	2214
SyncManJAXB	1865	1971	2123
AsyncManLowLevelPolling	4170	4641	4109
AsyncGenPolling	4444	4337	3989
AsyncManLowLevelCallback	5080	5095	5247
AsyncGenCallback	3898	4837	4444
Summe	23446	24703	24188

Tabelle 8.1: Laufzeitverhalten der einzelnen Implementierungen bei hundertfachem Aufruf (Zeit in ms)

seitigen Implementierungen kein Unterschied im Laufzeitverhalten besteht. Es scheint, als lieferten die generierte Implementierung, die Implementierung über eine benutzerdefinierte JAXB-Klasse sowie die Implementierung über die Low-Level Java APIs für XML die selben Resultate. Gleiches lässt sich auch an den

[2] Als Testplattform dient ein Pentium 4 PC mit einem 2,4GHz P4 Prozessor und 1024 MB Ram. Als Betriebssystem wird Windows XP (SP2) eingesetzt.

hierzu äquivalenten synchronen clientseitigen Implementierungen feststellen. Diese benötigen für hundert Aufrufe im Schnitt ca. 2 Sekunden. Etwas langsamer hingegen verhalten sich die asynchronen Aufrufe, welche über beide Modelle der asynchronen Kommunikation hinweg ca. 4,5 Sekunden benötigen. Dies könnte an der von der Java Laufzeitumgebung durchgeführten Threadverwaltung liegen.
Zusammenfassend können aus dem Inhalt dieses Abschnitts zwei Aussagen getroffen werden. Zum einen ist der Un-/Marshalling-Prozess in JAXB sehr effizient umgesetzt worden und steht der äquivalenten Low-Level Implementierung in keinster Weise nach. Zum anderen benötigen die asynchronen Aufrufe eine etwas längere Zeitspanne als die Synchronen. Dabei ist zu beachten, dass die betrachteten Zeitangaben jeweils einem hundertfachen Aufruf entsprechen. In der Realität ist diese Zeitspanne, verglichen mit der Übertragungsgeschwindigkeit im Netz sowie der durchzuführenden Geschäftslogik, eher vernachlässigbar.

8.4 Zusammenfassung und Fazit

Dieses Kapitel hat die Möglichkeiten der serverseitigen Implementierung eines Webservices unter JAX-WS aufgezeigt. Im direkten Vergleich mit der JAX-RPC Technologie fällt insbesondere die Einfachheit einer solchen Implementierung auf. Wo bei JAX-RPC noch die Konfigurationsdateien jaxrpc-ri.xml und config.xml (für das wscompile Tool) manuell geschrieben und das WAR-Archiv zweifach erstellt werden musste, reicht in JAX-WS eine einfache `@WebService` Annotation im Quelltext aus. Die JAX-WS Laufzeitumgebung richtet daraufhin, je nach verwendeter Java Plattform, einen dynamischen Proxy für die Kommunikation mit den Servicekonsumenten ein und erstellt alle erforderlichen Artefakte.
In Bezug auf die drei möglichen Szenarien die bei der Entwicklung eines Serviceanbieters anzutreffen sind, lässt sich folgende Beurteilung für die serverseitige Implementierung eines Webservices unter JAX-WS aufstellen:

- „Start von WSDL“ – Für dieses Szenario bietet JAX-WS die generierte Implementierung über die Verwendung eines WSDL zu Java Generators an. Dieser generiert, wie im clientseitigen Fall, alle für die Entwicklung eines JAX-WS Servers benötigten Java-Artefakte. Der Entwickler braucht sich nur noch um die Implementierung des generierten SEI kümmern. Hat man es mit einer vergleichsweise kleinen WSDL zu tun, so bietet JAX-WS zudem die Benutzung der Low-Level API für XML in Form der Klasse `@Source` an, welche das direkte Arbeiten mit der XML-Nachricht ermöglicht.

- „Start von Java“ – Typischerweise wird die Entwicklung eines Serviceanbieters über die Realisierung der Geschäftslogik begonnen. Soll dann eine bestimmte Klasse dieser Implementierung als Webservice angeboten werden, so wird diese Klasse in JAX-WS einfach mit `@WebService` annotiert. JAX-WS generiert, unter Benutzung von JAXB, daraufhin automatisch

eine WSDL und erzeugt alle für die Kommunikation mit den Servicekonsumenten benötigten Artefakte. Überdies bietet JAX-WS über die Benutzung der Klasse `SOAPMessage` eine einfache API an, die ein direktes Arbeiten mit den XML-Nachrichten ermöglicht. Dieses und das vorige „Start von WSDL“ Szenario stellen aufgrund der geschilderten Vorteile die „grüne Wiese“ bei der Entwicklung eines JAX-WS Webservices dar.

- „Start von WSDL & Java“ – Dieses Szenario ist der typische Ausgangspunkt für die Einführung einer SOA und ist über JAX-WS am schwierigsten umzusetzen. Generell muss man, ähnlich wie bei der clientseitigen Umsetzung dieses Szenarios, unterscheiden, ob sich nur die WSDL geändert hat, oder ob der Webservice initial in das bestehende Projekt integriert wird. Letzteres ähnelt dem „Start von WSDL“ Szenario mit dem Unterschied, das die generierten Klassen in irgendeiner Form in das bestehende Projekt eingebunden werden müssen. Wenn sich nur die WSDL geändert hat, so kann bei der generierten Implementierung über die JAX-WS bzw. JAXB Annotationen versucht werden, ein Mapping mit der geänderten WSDL herzustellen. Der Quelltext (ohne Annotationen) kann dabei aufgrund vielfältiger Abhängigkeiten oftmals nicht verändert werden. Bei einer manuellen Implementierung über die Low-Level API für XML gestaltet sich dieser Fall einfacher, da zum einen die WSDL vergleichsweise kleiner ist und zum anderen der direkte Umgang mit den XML-Nachrichten großzügiger implementiert werden kann, so dass unter Umständen kein Mapping durchgeführt werden muss.

9 Fazit

Die Stärken von JAX-WS liegen insbesondere in der initialen Entwicklung eines Webservices. Die initiale Entwicklung spiegelt sich dabei in den beiden Szenarien „Start von Java“ und „Start von WSDL“ wieder, welche beide sehr effizient und einfach in JAX-WS umgesetzt werden können. Hierfür bietet JAX-WS eine Vielzahl möglicher Implementierungen an, die neben einer eleganten High-Level API auch den direkten Umgang mit der XML-Nachricht ermöglichen. JAX-WS nimmt dem Entwickler dabei die Aufgabe der Erzeugung aller für die Bereitstellung des Webservices benötigten Artefakte ab, realisiert die Kommunikation über das Konzept des dynamischen Proxys und führt das Un-/Marshalling der XML-Nachrichten durch. Doch gerade der letzte Punkt ist ausschlaggebend, dass JAX-WS in den für die Einführung einer SOA typischen „Start von WSDL & Java“ Szenario nur bedingt einsetzbar ist. Dies resultiert daraus, dass JAX-WS nur JAXB als Binding und Mapping Tool vorsieht, JAXB aber streng genommen ein reines Binding Tool ist. Dies hat die Implikation, dass eine Änderung an der WSDL immer die erneute Übersetzung aller betroffenen Java-Klassen zur Folge hat, dies aber nicht immer möglich ist. Obwohl das Problem ein Defizit der JAXB Spezifikation ist, so lässt JAX-WS leider kein anderes Binding und Mapping Tool als JAXB zu, weshalb dies gleichermaßen als Schwachstelle von JAX-WS zu sehen ist.

Literaturverzeichnis

[BP1.1] WS-I: WS-I Basic Profile 1.1. (2006), April. http://www.ws-i.org/Profiles/BasicProfile-1.1.html, Abruf: 19.02.2008

[But03] Butek, Russell: Which style of WSDL should I use? (2003), Oktober. http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/, Abruf: 07.01.2008

[Cer02] Cerami, Ethan: *Web Services Essentials*. O'Reilly, 2002

[GlassFish] GlassFish Community: GlassFish Application Server for Java EE 5. (2008). https://glassfish.dev.java.net/, Abruf: 12.01.2008

[Han07] Hansen, Mark D.: *SOA Using Java Web Services*. Prentice Hall, 2007

[JAX-RPC-Types] Sun Microsystems, Inc.: Types Supported by JAX-RPC. http://java.sun.com/j2ee/1.4/docs/tutorial/doc/JAXRPC4.html, Abruf: 08.02.2008

[Jos07] Josuttis, Nicolai M.: *SOA in Practice*. O'Reilly, 2007

[JSR101] Sun Microsystems, Inc.: Java APIs for XML based RPC. (2003). http://www.jcp.org/en/jsr/detail?id=101, Abruf: 23.02.2008

[JSR109] Sun Microsystems, Inc.: Web Services for Java EE. (2006), Mai. http://www.jcp.org/en/jsr/detail?id=109, Abruf: 24.12.2007

[JSR222] Sun Microsystems, Inc.: Java Architecture for XML Binding (JAXB) 2.0. (2006), November. http://jcp.org/en/jsr/detail?id=222, Abruf: 18.12.2008

[JSR224] Sun Microsystems, Inc.: JavaTM API for XML-Based Web Services (JAX-WS). http://www.jcp.org/en/jsr/detail?id=224, Abruf: 02.03.2008

[Lie07] Liebhart, Daniel: Was ist eigentlich eine serviceorientierte Architektur (SOA)? (2007), August. http://www.tecchannel.de/webtechnik/soa/1725211/, Abruf: 20.11.2007

[MTSM03] McGovern, James ; Tyagi, Sameer ; Stevens, Michael ; Matthew, Sunil: *Java Web Services Architecture*. Morgan Kaufmann Publishers, 2003

[Proxy] Sun Microsystems, Inc.: Dynamic Proxy Classes. (1999). `http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html`, Abruf: 05.02.2008

[WCL+05] Weerawarana, Sanjiva ; Curbera, Francisco ; Leymann, Frank ; Storey, Tony ; Ferguson, Donald F.: *Web Services Platform Architecture SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, 2005

[Win99] Winer, Dave: XML-RPC Specification. (1999), Juni. `http://www.xmlrpc.com/spec`, Abruf: 25.03.2008

[WSDL1.1] The World Wide Web Consortium: Web Services Description Language (WSDL) 1.1. (2001). `http://www.w3.org/TR/2001/NOTE-wsdl-20010315`, Abruf: 07.02.2008

[XPath1.1] The World Wide Web Consortium: XML Path Language (XPath), Version 1.0. `http://www.w3.org/TR/xpath`, Abruf: 27.03.2008

[XS1] The World Wide Web Consortium: XML Schema Part 1: Structures. (2001), Mai. `http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/`, Abruf: 17.01.2008

[XS2] The World Wide Web Consortium: XML Schema Part 2: Datatypes. (2001), Mai. `http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/`, Abruf: 23.01.2008

Erklärung

Hiermit erkläre ich, dass ich die Arbeit selbständig und nur unter Zuhilfenahme der aufgeführten Hilfsmittel sowie den Hinweisen meiner Betreuer und Mitarbeitern der Firma PENTASYS AG angefertigt habe.

München, den 30.4.2008

Ralf Klein Heßling