Seminararbeit

„Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen“

Die Entwicklung von SOAP-Webservices mit Java 6 und Glassfish V 3.1.2

Von Tobias Miller

Gliederung

Ich, Tobias Miller, habe diese Seminararbeit selbstständig erstellt.

München, den 7. Juni 2013

Tobias Miller (Matrikelnummer: 05530910)
_____

1 Motivation

Die heutigen IT-Systemlandschaften großer Unternehmen weisen einen hohen Grad an Heterogenität auf. Ein solch betrachteter Systemverbund besteht aus vielen Systemen mit unterschiedlichsten Architekturen und Technologien. Die einzelnen Fachgebiete der betriebswirtschaftlichen Geschäftslogik eines Unternehmens werden dabei auf einzelne Systeme abgebildet. Obwohl durch den Einsatz von ERP-Systemen klassische Bereiche wie etwa Vertrieb oder Einkauf weitgehend abgedeckt werden, setzen viele große Unternehmen ergänzend zu dem jeweiligen ERP-System proprietäre IT-Systeme ein. Angesichts dieser Zusammenhänge sind die Geschäftsdaten und Geschäftslogik auf alle fachspezifischen IT-Systeme verteilt. Da betriebswirtschaftliche Geschäftsprozesse jedoch fachübergreifend ablaufen, entsteht bezogen auf die IT die Anforderung, ein integriertes Zusammenspiel der beteiligten IT-Systeme zu gewährleisten. Durch Schnittstellen zwischen den einzelnen IT-Systemen ergibt sich insgesamt ein unternehmensweites Netz, über das Geschäftsdaten ausgetauscht werden. Über die A2A-Kommunikation hinweg besteht zusätzlich auch eine unternehmensübergreifende Kommunikation mit Geschäftspartnern.

In der heutigen Zeit bilden SOAP basierte Webservices in der architektonischen Entwicklung von IT-Systemlandschaften einen signifikanten Anteil. Obwohl diese Seminararbeit eine grundlegende Wissensbasis über SOAP-Webservices einschließlich aller verwendeten Technologien voraussetzt und primär auf die Entwicklung von Webservices mit Java abzielt, werden nachfolgend in einem kurzen Rundumschlag die Kernaspekte dargelegt:

Ein Anwendungssystem stellt eine in sich abgeschlossene Geschäftslogik in Form eines Webservice zur Verfügung. Die Implementierung dieser Geschäftsfunktionalität erfolgt in einer bestimmten Programmiersprache und auf diese wird auch grundsätzlich von anderen Modulen des gleichen Anwendungssystems zugegriffen. Über die Bereitstellung als Webservice soll auch anderen Anwendungssystemen mit einer abweichenden technologischen Umgebung der Zugriff auf die Geschäftsfunktionalität ermöglicht werden. Der Dienstanbieter stellt dem Dienstnutzer eine technologieneutrale Servicebeschreibung zur Verfügung, die der Dienstnutzer in seine Systemumgebung einbindet.

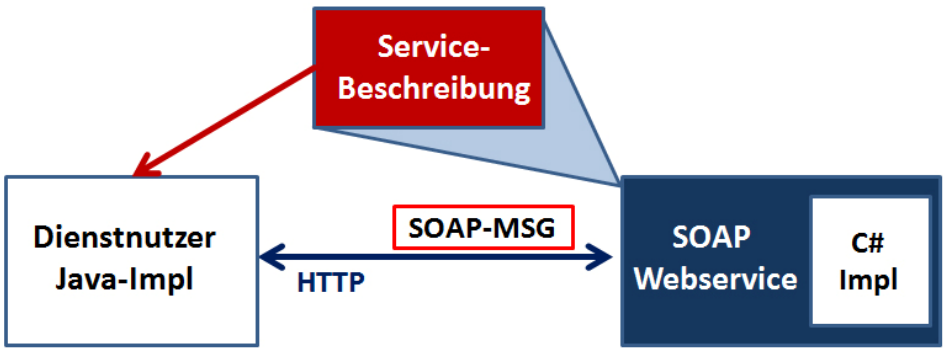

Abbildung 1 - Grundlegende Funktionsweise von SOAP-Webservices

Durch die Einbindung der Servicebeschreibung werden dem Dienstnutzer sogenannte Stub-Klassen in seiner proprietären Programmierumgebung generiert, durch die er auf die publizierten Webserviceoperationen zugreifen kann. Ruft der Dienstnutzer innerhalb seiner Programmierumgebung eine Webserviceoperation in Form eines Methodenaufrufs auf, so

wird der dahinterliegende Nachrichtenverkehr zwischen Dienstnutzer und Dienstimplementierung über den Austausch von XML-Dokumenten realisiert. Mittels eines zwischengelagerten Proxy werden entlang des Nachrichtenflusses Transformationen zwischen Objekten der jeweiligen Zielsprache und XML vollzogen. Diese implizit stattfindenden Vorgänge bleiben dem Dienstnutzer verborgen.

Einer der wesentlichen Vorteile von Webservices in einem Integrationsszenario ist eine hohe lose Kopplung zwischen den beteiligten Systemen. Dies ist darauf zurückzuführen, dass der Dienstnutzer bzw. das Anwendungssystem A keine Implementierungsdetails des auf Anwendungssystem B hinterlegten Webservices wissen muss. Die zugrundeliegende Dienstimplementierung kann in einer beliebigen Programmiersprache erfolgen. Somit ergibt sich durch SOAP-Webservices die Möglichkeit, unterschiedlich technologische Plattformen miteinander zu verbinden. Zudem besteht bei SOAP-Webservices gerade im Unternehmensumfeld ein weiterer Vorteil: Auszutauschende Geschäftsdaten können oft nur auf eine komplexe Nachrichtenstruktur abgebildet werden. Aufgrund der Möglichkeit XML-basierte SOAP-Nachrichten beliebig hierarchisch zu schachteln, können Nachrichtentypen je nach benötigter Struktur formuliert werden.

2 Webservice-Standards

Wird das Thema der Standardisierung von Webservices im Allgemeinen betrachtet, so existieren viele Spezifikationen von unterschiedlichen Normungsgremien. Demnach ist die Situation komplizierter als beispielsweise bei Java Transaktionen, die weitgehend von der Spezifikation Java Transaction API abgedeckt werden. Bevor auf die Java-spezifischen Spezifikationen eingegangen wird, soll zunächst ein Überblick über die grundsätzlichen SOAP Webservices Standards gegeben werden.

2.1 SOAP-Webservice Standards

Alle in diesem Zusammenhang relevanten Spezifikationen wurden von den Konsortien W3C und OASIS entwickelt. Das World Wide Web Konsortium ist neben dem Themengebiet der Webservices generell das hauptverantwortliche Gremium für die Standardisierung von Webtechnologien wie etwa XHTML oder CSS. Zu den von W3C beherbergten Standards mit Bezug zu Webservices gehören XML, XSD, SOAP und WSDL. Demgegenüber trägt OASIS (Organization for the Advancement of Structured Information Standards) Standards im E-Business Bereich und die nicht von W3C übernommene Teilmenge an Webservice-Standards wie etwa UDDI oder die sogenannten WS-*-Standards. [1, p. 418] Die WS-*-Standards erweitern die Webservice-Basistechnologie, um die Zuverlässigkeit eines Webservices bzgl. mehrerer Kriterien wie Sicherheit, Nachrichtentransfer, Nachrichtenübertragung oder Transaktionssicherheit zu verbessern.

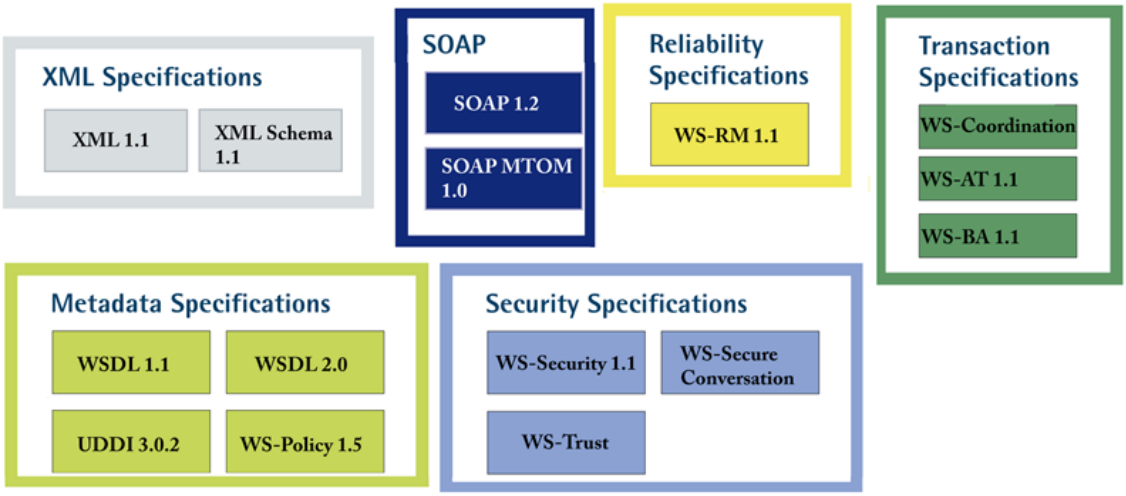

Abbildung 2 - Einige von W3C und OASIS entwickelten Webservice-Standards[1]

In der Historie der Webservice-Entstehung haben W3C und OASIS die Standards XSD, SOAP, WSDL, UDDI und die WS-*-Standards neu erarbeitet und sich bei der Entwicklung dieser auf den bereits bestehenden XML-Standard gestützt. Nachfolgend werden die Basis-Webservice-Standards kurz vorgestellt:

1. XML dient als Integrationstechnologie und wird als Basisfundament verwendet, auf dem die Webservice-spezifischen Standards aufsetzen. Durch XML wird der Rahmen festgelegt, innerhalb dessen eine genauere Spezifizierung mittels WSDL, SOAP und UDDI vorgenommen wird. [1, p. 416]
2. Das Transportprotokoll HTTP fungiert als Standard-Übertragungsprotokoll für XML-Nachrichten. Alternativ wäre auch eine Übertragung über HTTPS, FTP, SMTP, JMS etc. möglich. [1, pp. 416,418]
3. SOAP legt das XML-Nachrichtenformat für Webservices genau fest. Dieses Anwendungsprotokoll setzt auf dem Übertragungsprotokoll HTTP auf. Somit ist eine SOAP-Nachricht eine XML-Nachricht, die ein spezielles Nachrichtengerüst aufweist.
4. Der WSDL-Standard definiert eine Servicebeschreibung, die folgende Bestandteile enthält: Daten- und Nachrichtentypen, die möglichen Interaktionen zwischen Webservice-Client und Webservice einschließlich Parametern und Rückgabewerten, den URL-Endpunkt als erreichbare Adresse und verwendete Kommunikationsprotokolle. Wird die Semantik einer WSDL-Servicebeschreibung auf Java übertragen, so kann diese mit einem Interface gleichgesetzt werden. [1, p. 417]
5. Über den ebenfalls XML-basierten UDDI-Standard können Webservice-Verzeichnisse abgebildet werden. Bezogen auf ein Unternehmen kann damit ein zentrales Verzeichnis mit Informationen über alle angebotenen Webservices zur Verfügung gestellt werden.

[1] Es existieren viel mehr Webservice-Standards. Eine gesamte Übersicht kann über den nachfolgenden Link eingeholt werden:
http://www.innoq.com/soa/ws-standards/poster/innoQ%20WS-Standards%20Poster%202007-02.pdf

2.2 Java Webservice Standards

Im Rahmen des Java Community Process[2] wurden mehrere Spezifikationen aufgesetzt, die sich auf die Umsetzung der Webservice-Technologie im Java-Umfeld beziehen. Diese Palette an Standards ist Bestandteil von Java EE 6 und Java SE 6. In die Dachspezifikation JEE 6 sind die Java-Webservice bezogenen Spezifikationen Java API for XML Webservices (JAX-WS), Java Architecture for XML Binding (JAXB), Web Services Metadata und Java API for XML Registries (JAXR) eingearbeitet. [1, p. 419] Da JAX-WS und JAXB die wesentlichen Grundbestandteile der Java Webservice Technologie ausmachen, soll nachfolgend ausschließlich zu diesen beiden Spezifikationen Stellung genommen werden.

2.2.1 Java API for XML Webservices (JAX-WS)

Java API for XML Webservices ist der erste Standard, der diverse Java-APIs von unterschiedlichen Webservices-Engine-Herstellern einheitlich normt. Die Spezifikation beschreibt im Allgemeinen den Funktionsumfang, der benötigt wird, um Webservices auf Basis von Java anzusprechen und zu implementieren. [2, p. 33] Genauer gesagt wird sowohl das Mapping zwischen Java und der Webservice Description Language als auch das umgekehrte Mapping definiert. Hierfür werden eine Menge von Annotationen und Programmier-Schnittstellen definiert [1, p. 419].

Der Kerngedanke von JAX-WS ist es, dem Anwendungsentwickler die Komplexität der SOAP-Webservice-Technologie zu verbergen. Bei der Konzeption des Standards orientierte man sich aus diesem Grund wie auch bei anderen JEE 6 Spezifikationen stark an dem deklarativen Programmierstil mittels Annotationen. Der Anwendungsentwickler braucht nur an bestimmten Stellen seine implementierte Java-Anwendungslogik mit Webservice-spezifischen Annotationen versehen, die anderen Aufgaben wie die eigentliche Webservice-Erzeugung und die Low-Level-Nachrichtenprozessierung zur Laufzeit werden von JAX-WS übernommen [1, p. 419].

An diesem Punkt ist richtigerweise anzumerken, dass die JAX-WS-Spezifikation sehr stark mit der JAXB-Spezifikation verwoben ist [3, p. 1]. Damit ist gemeint, dass große Aufgabenteile der eigentlichen Webservice-Erzeugung (Mapping zwischen Java und WSDL) und der Low-Level-Nachrichtenprozessierung zur Laufzeit (Mapping zwischen Java und SOAP-Nachrichten) nicht autark von JAX-WS bewerkstelligt werden bzw. in der Spezifikation beschrieben sind. Denn diese Aufgabenteile sind in der JAXB-Spezifikation definiert und daher verweist die JAX-WS-Spezifikation an diversen Stellen auf die JAXB-Spezifikation. Nachdem im nachfolgenden Kapitel zunächst allgemein auf die JAXB-Spezifikation eingegangen wird, soll danach anhand der Laufzeit-Nachrichtenprozessierung genauer erklärt werden, wie JAX-WS mit JAXB interagiert.

[2] Der JCP wurde 1998 zugehöriger Bestandteil der Java-Welt. Jede neue Spezifikation oder Referenzimplementierung durchläuft dabei einen standardisierten Prozess, bevor eine Freigabe erfolgt. [2, p. 33] Dabei können die Mitglieder des JCP den Prozess initiieren, indem sie einen Vorschlag auf eine Java-Erweiterung an das Executive Committee richten. Wenn das EC den Vorschlag annimmt, so wird diese Erweiterung als Java Specification Request in den Entwicklungsprozess aufgenommen [14].

2.2.2 Java Architecture for XML-Binding (JAXB)

Grundsätzlich beschreibt die JAXB-Spezifikation, wie Java-Klassen an ein XML-Schema gebunden werden. Dies beinhaltet sowohl die Serialisierung von einem Java-Objekt nach XML als auch in umgekehrter Form die Deserialisierung von XML zu einem Java-Objekt. [2, p. 34] Auch unabhängig des Java-Webservice-Bereichs wird JAXB-Funktionalität in anderen Entwicklungsbereichen wie etwa der Persistierung eines Java-Objektes in einer XML-Datei oder der Übertragung von XML-Dateien über ein Netzwerk angewandt. Daher bietet JAXB Funktionalitäten für alle Anwendungsfälle rund um die XML-Verarbeitung mittels Java.

Der JAX-WS Nachrichtenfluss inklusive der beteiligten JAXB API soll nachfolgend dargestellt und erläutert werden:

Abbildung 3 - JAX-WS Nachrichtenfluss

Nach der Ausführung eines parametrisierten Methodenaufrufs einer Java-Clientimplementierung und dem Durchlauf der Client-Stubs benötigt JAX-WS für das XML-Binding JAXB-Funktionalität. Zur Laufzeit bildet JAXB mittels des sogenannten Marshalling-Vorgangs das übermittelte Java-Objekt auf den korrespondierenden XML-Datentypen ab und pflanzt diesen in das erzeugte SOAP-Nachrichtengerüst ein. Daraufhin initiiert JAX-WS den Sendevorgang der SOAP-Request-Nachricht über das SOAP-Protokoll.

Nach dem Empfang der übermittelten SOAP-Request-Nachricht auf einer Java-basierten Dienstseite wird die Marshalling-Aufgabe erneut von JAX-WS an JAXB delegiert. Daraufhin bildet JAXB im Rahmen des Parse-Vorgangs den XML-Datentypen auf den gebundenen Java-Parameter ab. Die Dienstimplementierung wird mit dem übermittelten Methodenwert durchlaufen, und ein ermittelter Rückgabewert wird durch JAXB auf das SOAP-Response-Nachrichtenformat abgebildet und zurück an den Webservice-Client geschickt. Die JAXB-Komponente des Senders transformiert letztendlich die SOAP-Nachricht in den gebundenen Java-Methodenrückgabewert der Clientimplementierung.

3 Das Java Enterprise Edition Modul JAX-WS RI

Ab der Version 6 ist die JAX-WS Referenzimplementierung standardmäßig im Paket der Java Standard Edition enthalten, das wiederrum ein Bestandteil der jeweiligen Java Enterprise Edition ist [2, p. 36]. Nachdem ein Webservice auf einer Java SE 6+ stützenden Entwicklungsumgebung wie Eclipse unter der Verwendung von JAX-WS API-Funktionalitäten entwickelt wurde, muss dieser im Anschluss ebenfalls auf einem Applikationsserver bereitgestellt und ausgeführt werden. Exemplarisch wird an mehreren Stellen dieser Seminararbeit der Applikationsserver Oracle Glassfish V3.1.2 betrachtet, welcher sich im Idealfall ebenfalls auf die identische Java SE Version stützt. Glassfish hat innerhalb der Webcontainer-Komponente die Webservice-Engine Metro, die als Laufzeitumgebung für Webservices fungiert, standardmäßig eingebettet [2, p. 195].

Der gesamte Metro Webservice Stack setzt sich im Wesentlichen aus folgenden Bestandteilen zusammen [2, p. 34]:

- Webservice-Engine
- Java API for XML Webservices (JAX-WS RI)
- Java Architecture for XML Binding (JAXB RI)
- Diverse Webservice-Standard Implementierungen
- Webservice Interoperability Technologies (WSIT)

Die Webservice-Engine, die JAX-WS RI, die JAXB RI und ein Teil von weiteren Webservice-Standards sind bereits vollständig in der Java SE 6+ integriert. Der andere Teil existierender Webservice-Standard-Implementierungen und WSIT sind nicht Bestandteil von Java SE 6+ [2, pp. 33-36]. Diese zusätzlichen Metro-Funktionalitäten sind standardmäßig separat auf dem Glassfish V3+ Applikationsserver vorinstalliert. Dieser Bestandteil von Metro ist ein Subprojekt des Applikationservers Glassfish V3+, kann jedoch alternativ auch nachträglich in einen anderen Applikationsserver wie Apache Tomcat eingebunden werden [1, p. 420]. Obwohl nur ein Teil aller Metro-API's direkt in Java SE 6+ eingearbeitet sind, sind die extern angesiedelten API's wie etwa WSIT vollkommen in die Java-Umgebung integriert. Die nachfolgende Abbildung veranschaulicht die komplexen architektonischen Zusammenhänge:

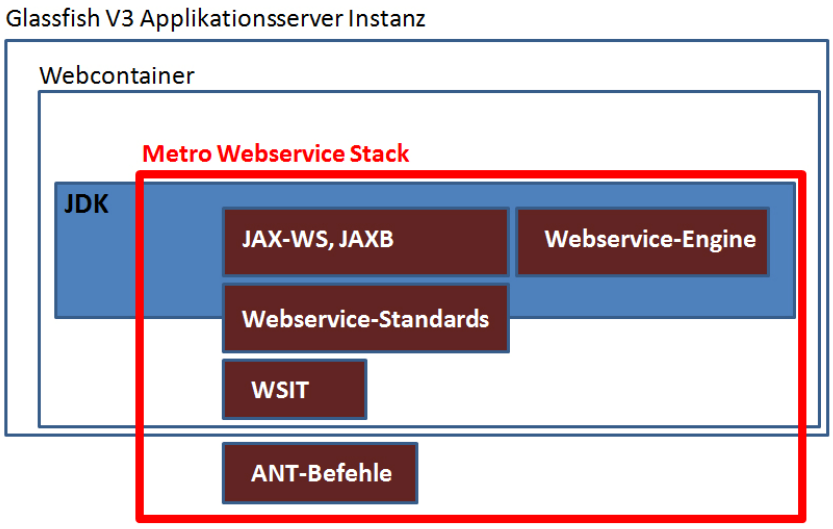

Abbildung 4 - Anatomie von Metro (sinngemäß übernommen aus [2, p. 36])

Aufgrund der Anatomie des Metro Webservice Stack ergeben sich zwischen den einzelnen APIs Versionsabhängigkeiten. Da der Release-Zyklus der Java SE Updateversionen anders getaktet ist als der Release-Zyklus der JAX-WS RI, kann anhand der Java SE Version nicht auf die JAX-WS RI Version geschlossen werden. Über den Befehl „wsimport – version“ kann die JAX-WS-Versionsnummer ermittelt werden. Es sei angemerkt, dass die höherwertigen WSIT-Funktionalitäten erst ab der Version 2.1 der JAX-WS-Referenzimplementierung enthalten sind, welche Bestandteil der Java SE 6 Update 4 ist. [2, p. 36] Falls bereits ein Enterprise Service Bus oder Applikationsserver in der IT-Systemlandschaft eines Unternehmens existiert, der eine ältere Java SE verwendet, mindestens jedoch J2SE 5.0, so besteht die Möglichkeit, die JAX-WS RI auf einen aktuelleren Stand nachzurüsten [2, p. 37].

4 Entwicklungsansätze eines Webservice

Zur Erstellung eines Webservice existieren die beiden Ansätze „Code-First“ bzw. in einer Java-basierten Entwicklungsumgebung „Java-First“ und „Contract-First“ bzw. „WSDL-First”. Beim Code-First-Ansatz wird zuerst die Implementierung des Dienstanbieters in einer sprachspezifischen Entwicklungsumgebung wie Java vorgenommen. Im Anschluss erstellt der Entwickler des Dienstes auf Basis der implementierten Java-Klassen eine WSDL-Datei, die einem Dienstnutzer zur Verfügung gestellt wird. Dieser kann innerhalb seiner sprachspezifischen Entwicklungsumgebung wie etwa .Net aus der WSDL-Datei einen sogenannten Dienstnutzer-Stub erzeugen. Bei dem Dienstnutzer-Stub handelt es sich beispielsweise um generierte C#-Klassen, über die der Entwickler des Dienstnutzers den Webservice ansprechen kann. Abschließend implementiert der Entwickler des Dienstnutzers unter der Verwendung der generierten Stub-Klassen den Webservice-Konsumenten.

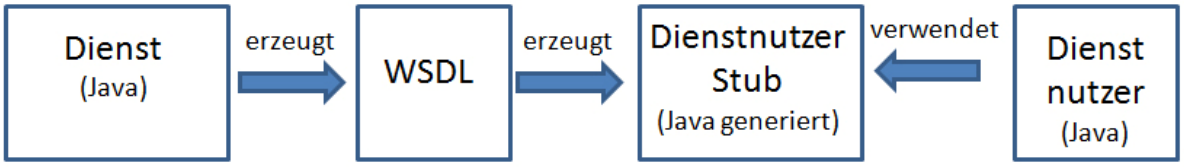

Abbildung 5 - Code-First-Ansatz (übernommen aus [2, p. 164])

Beim Contract-First-Ansatz hingegen ist die WSDL-Datei der Ausgangspunkt, aus der sowohl der Dienstanbieter, als auch der Dienstnutzer in deren sprachspezifischen Entwicklungsumgebung ihre Stubs erzeugen. Im Anschluss daran entwickelt der Entwickler des Dienstes die Anwendungslogik des Dienstes und der Entwickler des Dienstkonsumenten den Zugriff auf den Dienstnutzer-Stub.

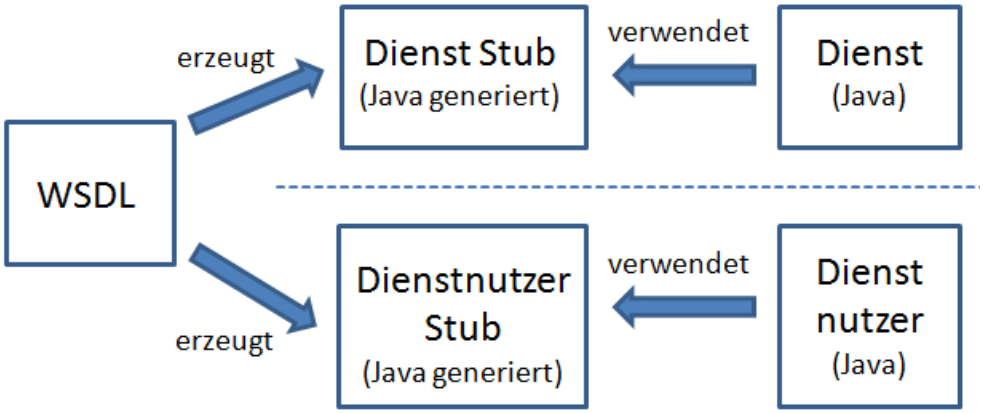

Abbildung 6 - Contract-First-Ansatz (übernommen aus [2, p. 154])

In der gängigen Praxis wird die initiale Erstellung der WSDL-Datei mittels des Code-First-Ansatzes vollzogen. Bei späteren Weiterentwicklungen erfolgt meist eine Wandlung in den Contract-First-Ansatz, d.h. die WSDL-Datei wird überarbeitet, beide Parteien generieren sich deren Stubs nach und passen gegebenenfalls deren Implementierungen an [2, p. 164].

Für die Generierung der Stubs gibt es mehrere Möglichkeiten:

1. Innerhalb des Kommandozeilenfensters kann das JDK-Tool „wsimport.exe“ genutzt werden (<JDK-HOME>/bin/wsimport.exe)
2. Nutzung des gleichnamigen Ant-Tasks „wsimport“, wofür jedoch ein ANT-Skript erstellt und ausgeführt werden muss. Dabei ist zu beachten, dass der „ws-import“-Task nicht im JDK enthalten ist, sondern nur Bestandteil der Metro-Installation [2, p. 155].

Gegenwärtig verwendete Entwicklungsumgebungen wie Eclipse oder Netbeans bieten zur Erstellung von Webservices für beide Ansätze einen dementsprechenden Wizard. Bei Eclipse kann innerhalb des Wizards zwischen „Bottom Up Java Bean Webservice“ für den Code-First-Ansatz und „Top Down Java Bean Webservice“ für den Contract-First-Ansatz gewählt werden. Diese Möglichkeit der Stub-Generierung wurde bewusst nicht in obiger Auflistung aufgeführt, da zumindest der Eclipse-Wizard nach eigener Beobachtung auf Basis des SOAP-Webservice-Stack Axis und nicht JAX-WS die Stubs generiert. Axis erzeugt nämlich eine vollkommen andere Stub-Klassenstruktur. Dennoch bleibt es dem Client überlassen, zwischen der JAX-WS- und AXIS-Generierung zu wählen. Falls auf der Dienstseite erweiterte Webservice-Technologien, wie etwa WSIT, verwendet werden, so ist jedoch die JAX-WS-Generierung aufgrund der sichergestellten Umsetzung der Standards zu präferieren.

5 Entwicklung eines Webservice

Anhand einer beispielhaften Entwicklung eines Webservice soll nachfolgend der Code-First-Ansatz verdeutlicht werden. Als zugrundeliegende Entwicklungsumgebung wurde Eclipse JEE Juno gewählt und für die Ausführung des Webservice eine separate lokale Installation des Applikationsservers Oracle Glassfish 3.1.2.

5.1 Konfiguration der Entwicklungsumgebung

Vor den eigentlichen Entwicklungsschritten muss Glassfish in Eclipse integriert werden, indem in der Server-Ansicht der Applikationsserver neu angelegt wird. Innerhalb des Setups wird das Auswählen eines Server-Adapters unter mehreren gelisteten Standardadaptern verlangt. Da der benötigte Oracle Glassfish Adapter in der Standardliste nicht enthalten ist, ist dieser nachträglich über den im gleichen Setupschritt platzierten Link „Download additional Server Adapters“ herunterzuladen. Unter den zum Download angebotenen Erweiterungen befindet sich u.a. ein Oracle Glassfish Bundle, das Server-Adapter zu mehreren Glassfish-Versionen umfasst. Nach einem erforderlichen Neustart von Eclipse ist bei erneuter Ausführung des „New Server“-Setups die Palette der Glassfish-Adapter in der Adapterliste enthalten. In Abbildung 7 sind die notwendigen Angaben dargestellt. Dort ebenfalls ersichtlich ist die festgelegte Konfiguration, dass die zu verwendende Server JRE der im Glassfish 3.1.2 hinterlegten Standardlaufzeitumgebung entsprechen soll. Optional wäre hier auch die Angabe einer anderen kompatiblen JRE Version möglich.

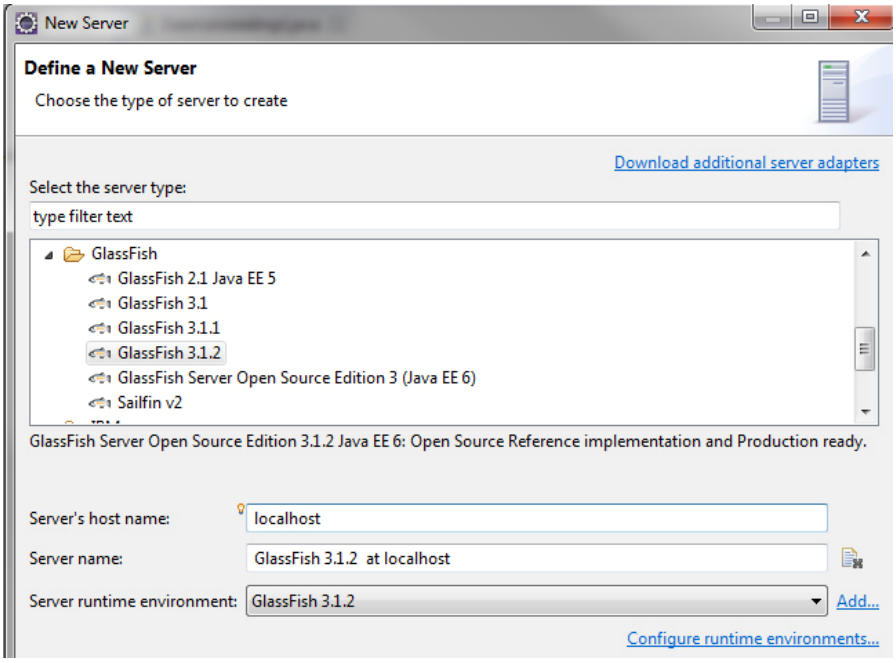

Abbildung 7 - Integration des Glassfish Applikationsserver in Eclipse

Nachdem abschließend eine einmalige Server-Authentifizierung hinterlegt wurde, ist die Integration abgeschlossen. Somit kann aus Eclipse heraus z.B. der Server gestartet bzw. gestoppt werden, und es können JEE-konforme EAR-Projekte auf dem Glassfish Applikationsserver bereitgestellt werden.

5.2 Entwicklung der Dienstimplementierung

Wie es der Code-First-Ansatz vorgibt, ist zu Beginn die Entwicklung des Dienstes in Form von einer Java-Klasse vorzunehmen. In Eclipse gibt es mehrere Java-Projekttypen, aus denen heraus ein Webservice erstellt werden kann: Die benötigten Java-Quellen können beispielsweise im Rahmen eines dynamischen Webprojektes oder eines Enterprise-Java-Bean-Projektes implementiert werden. Diese beiden Projekttypen sind einem EAR Applikation Projekt, das als Container fungiert, zuordenbar. Augenmerk soll hier bewusst auf eine Java Enterprise Anwendung im Stil von JEE 6 gelegt werden, die eine Geschäftslogikschicht in Form einer Enterprise Java Bean und eine Web-/Präsentationslogikschicht im Sinne eines ummantelnden Webservice beinhaltet. Bevor einzelne Implementierungsschritte detailliert erläutert werden, ist in der folgenden Abbildung zur Gesamtübersicht die finale Projektstruktur der Dienstimplementierung dargestellt:

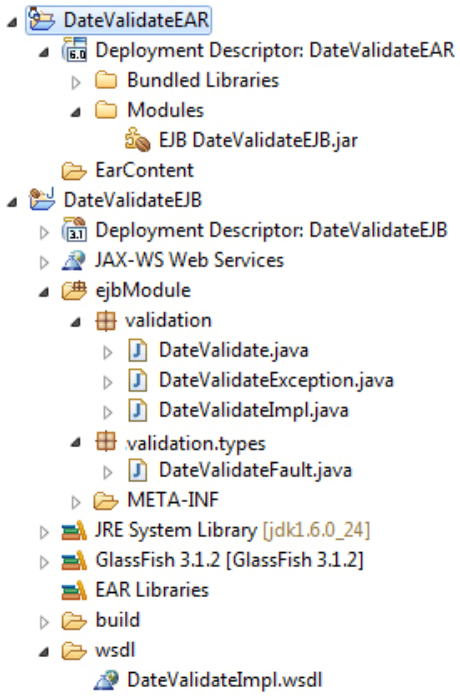

Abbildung 8 - Projektstruktur der Dienstimplementierung

Nach der Erzeugung des Java Bean Projekts wurde noch vor dem Beginn der Dienstimplementierung innerhalb des Pakets „validation“ ein Interface erstellt. An dieser Stelle soll zunächst auf ein paar JAX-WS-spezifische Aspekte hinsichtlich Interfaces eingegangen werden: Wenn ein Interface verwendet wird und die implementierende Klasse mittels der Annotation „@WebService(endpointInterface=<INTERFACE>)“ an dieses gekoppelt wird, so dient das Interface als alleinige Vorlage für die Generierung der Servicebeschreibung (WSDL-Datei). Falls kein Interface verwendet wird, so fungiert die Implementierungsklasse als Vorlage für die Servicebeschreibung. In der Literatur wird diesbezüglich unterschieden zwischen einer expliziten und einer impliziten Service-Endpunkt-Schnittstelle [4].

Bei der Verwendung eines Interfaces ist zu beachten, alle verwendeten webservicespezifischen Annotationen ausschließlich im Interface, nicht in der Implementierung und auch nicht verteilt in Interface und Implementierung zu definieren. Bei selbstdurchgeführten Tests dieser geschilderten Varianten konnte nämlich herausgefunden werden, dass beim Einsatz eines

Interface nur die Webservice-spezifischen Annotationen des Interfaces für die WSDL-Generierung berücksichtigt werden und die Webservice-spezifischen Annotationen der implementierenden Klasse missachtet werden. Diese „erzwungene“ und sinnvolle Trennung zwischen der Servicebeschreibung und der Anwendungslogik hat folgenden Vorteil:
Es ist von großer Bedeutung, einen Webservice für den Betrieb inklusive späteren Weiterentwicklungen bzw. Änderungen stabil zu halten. Ein Webservice-Client soll auch nach Änderungen eines Webservice die Garantie eines weiterhin funktionierenden Aufrufs mit Werten haben, die dieser laut „Vertrag“ erwartet. Somit gilt als Entwicklerrichtlinie, den Webservice stabil zu halten, indem bei der Weiterentwicklung die Servicebeschreibung unverändert bleibt. Das Hinzufügen neuer Operationen ist jedoch erlaubt. Da die Servicebeschreibung ein 1:1-Abbild des Interfaces und sie getrennt von der Anwendungslogik ist, hat der Entwickler es leicht, keine der Richtlinie nach „verbotenen“ Änderungen zu vollziehen, da er nur „erlaubte“ Modifizierungen in der Implementierung tätigt.

```
 9  @WebService(name="DateValidationService")
10  public interface DateValidate {
11
12      @WebMethod(operationName="validateSQLFormatDate")
13      @WebResult(name="realDate")
14      public boolean validateDate(@WebParam(name="sqlDate") String sqlDate) throws DateValidateException;
15
16  }
```

Listing 1 - Java-Interface mit Webservice-spezifischen Annotationen

Um eine Java-Klasse als einen Dienst zu kennzeichnen, ist sie und - falls eingesetzt - ihr Interface mit der Annotation @WebService zu versehen (Listing 1 Zeile 9). Alleine durch diese Annotation werden beim späteren Deployment-Vorgang auf den Glassfish Applikationsserver die benötigten Webservice-Artefakte, wie etwa die WSDL-Datei, automatisch generiert. Im Gegensatz zur benötigten @WebService-Annotation, die an eine Klasse gekoppelt ist, ist die Angabe der @WebMethod-Annotation (Listing 1 Zeile 12) angewandt auf eine Methode optional. Auch durch das Vernachlässigen der @WebMethod-Annotation zu einer Methode wird diese in eine Dienstoperation transformiert [2, p. 165].

Mithilfe von @SOAPBinding kann bestimmt werden, ob das SOAP-XML-Nachrichtenformat datenorientiert oder dokumentenorientiert aufgebaut ist. Bezogen auf die Nachrichtensemantik wurde für datenorientierte Webservices der SOAP Stil „RPC“ definiert. Im Gegensatz dazu wurde für fachliche Szenarien, in denen z.B. geschäftliche Informationen auf ein komplexes hierarchisches SOAP-XML-Nachrichtenformat abgebildet werden sollen, der SOAP Stil „DOCUMENT“ spezifiziert. In einem selbst aufgesetzten Test wurde einmal der hier betrachtete Webservice mit dem Stil *document* versehen und einmal mit *RPC*. Beim Vergleich des bereitgestellten WSDL-/XSD-Dateipaares der document-Variante mit dem Dateipaar der RPC-Variante konnte folgender Unterschied festgestellt werden:

- Bei *document* werden durch JAX-WS/JAXB alle technischen Informationen, wie spezielle XSD-Datentypen oder die Namespaces der Operationen, in die XSD-Datei verlagert, die in die WSDL-Datei eingebunden ist
- Bei *RPC* werden diese Informationen nicht in die XSD-Datei, sondern in die WSDL-Datei aufgenommen

Obwohl die Option besteht, diese beiden unterschiedlichen Einstellungen bzgl. des SOAP-Stils zu treffen, wird empfohlen, in jedem Fall das Nachrichtenverfahren *document* zu verwenden. Dies ist zurückzuführen auf eine weitverbreitetere Unterstützung in Webservice-Stack-Implementierungen [2, p. 102]. Somit wird bei dem Nachrichtenverfahren *document* eine höhere Interoperabilität erzielt. Im Listing 1 wurde die @SOAPBinding-Annotation bewusst weggelassen, da JAX-WS bei der WSDL-Generierung implizit den Default-Wert „DOCUMENT“ verwendet.

Durch die Eigenschaft „name“ bzw. „operationName“ bei den Annotationen @WebService, @WebMethod, @WebResult und @WebParam können der Webservice und die Operationen einschließlich derer Parameter und Rückgabewerte aussagekräftig beschrieben werden. Auch der Webservice-Client erhält dadurch innerhalb der Request-/Response-Struktur einer konkreten SOAP-Nachricht aussagekräftige Bezeichnungen.

Als nächster Vorgang ist im gleichen Paket eine Stateless Session Bean mit der Klassenbezeichnung „DateValidateImpl“ anzulegen:

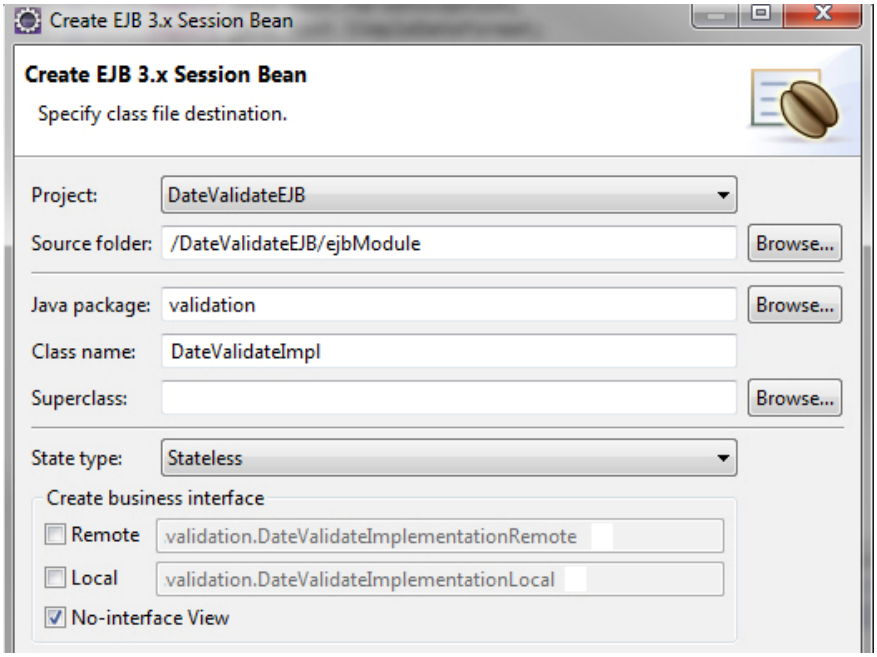

Abbildung 9 - Anlegen einer neuen EJB-Klasse

In einem der Setupschritte kann das zuvor erstellte Interface angegeben werden. Hierdurch wird von Eclipse nach Bestätigung des Wizards die neue EJB-Klasse mit den zu implementierenden Methoden aufgesetzt. Um die Klasse an das Interface zu binden, wird zusätzlich zu der „implements <Interface>“-Klausel (Listing 2 Zeile 17) die gesetzte Eigenschaft „endpointInterface“ der @WebService-Annotation benötigt (Listing 2 Zeile 15). Hier ist der Paketpfad festzulegen, in dem sich das Interface befindet, gefolgt von dem Interface-Namen.

Die Enterprise Java Bean soll eine Methode namens „validateDate“ beinhalten, die ein Datum auf SQL-Syntax validiert und einen booleschen Wert zurückliefert. Ein akzeptierter Wert entspricht dem Muster YYYY-MM-TT und ist gleichzeitig ein gültiges Datum. Das nachfolgende Listing 2 enthält den Java-Code der EJB als implementierende Klasse mit Ausnahme von Paketangabe, Imports und Getter-/ Setter-Methoden:

```
15 @WebService(endpointInterface = "validation.DateValidate")
16 @Stateless
17 public class DateValidateImpl implements DateValidate{
18
19     private String year = "";
20     private String month = "";
21     private String day = "";
22     private Date testDate = null;
23
24     @Override
25     public boolean validateDate(String sqlDate) throws DateValidateException{
26         try{
27             this.setYear(sqlDate.split("-")[0]);
28             this.setMonth(sqlDate.split("-")[1]);
29             this.setDay(sqlDate.split("-")[2]);
30
31             if(this.getYear().length() == 0 || this.getMonth().length() == 0 || this.getDay().length() == 0 ||
32                     this.getYear().length() > 4 || this.getMonth().length() > 2 || this.getDay().length() > 2){
33                 DateValidateFault faultDetail = new DateValidateFault();
34                 faultDetail.setdate(sqlDate);
35                 faultDetail.setErrorCode(123);
36                 throw new DateValidateException("Tag- Monats- oder Jahresanteil ist leer/zu lange", faultDetail);
37             }
38
39         }catch (ArrayIndexOutOfBoundsException e) {
40             DateValidateFault faultDetail = new DateValidateFault();
41             faultDetail.setdate(sqlDate);
42             faultDetail.setErrorCode(234);
43             throw new DateValidateException("Sie müssen das Pattern XXXX-XX-XX einhalten!", faultDetail);
44         }
45
46         String consolidatedDate = sqlDate.split("-")[0] + "-";
47
48         consolidatedDate += new String((sqlDate.split("-")[1].length() < 2) ? "0"
49         + sqlDate.split("-")[1] : sqlDate.split("-")[1]);
50
51         consolidatedDate += "-" + new String((sqlDate.split("-")[2].length() < 2) ? "0"
52         + sqlDate.split("-")[2] : sqlDate.split("-")[2]);
53
54
55         SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
56
57         try {
58           this.setTestDate(sdf.parse(consolidatedDate));
59         } catch (ParseException e) {
60           return false;
61         }
62
63         if (!sdf.format(testDate).equals(consolidatedDate)) {
64             return false;
65         } else return true;
66     }
95
96 }
97
```

Listing 2 – Anwendungslogik der EJB-Klasse und Webservice-spezifischen Annotationen

Da in diesem Kapitel der Fokus auf der grundsätzlichen Entwicklung eines Webservice unter der Verwendung des Code-First-Ansatzes liegt, soll der Algorithmus nur grob erläutert werden. Dieser enthält zwei aufeinanderfolgende Validierungskriterien, nach denen das Datum auf Richtigkeit geprüft wird:

1. Im ersten Prüfblock (Zeile 26-44) wird validiert, ob der übermittelte Wert dem Muster „1..4 Zeichen - 1..2 Zeichen - 1..2 Zeichen“ entspricht. Falls nicht, ist das angegebene Datum ungültig und der nächste Prüfschritt wird ausgelassen.

2. Im zweiten strikteren Prüfblock (Zeile 46-65) wird validiert, ob es sich bei dem übermittelten Wert um ein wahrheitsgemäßes Datum handelt. Hier werden auch unechte Datumsangaben, wie etwa der Wert „2013-02-29“ erkannt und als ungültig eingestuft.

Um auch auf das Thema Fehlerbehandlung bei Java-Webservices einzugehen, wurde die „validateDate“-Methode als Exception-werfende Methode implementiert (Zeile 25). Die selbstdefinierte Exception-Klasse „DateValidateException“ ist für alle Ausnahmen gedacht, die vom Dienst geworfen werden sollen, wenn das Datumformat „XXXX-XX-XX“ vom Aufrufer nicht eingehalten wird. Grundsätzlich bietet der Einsatz einer feingranularen Fehlerbehandlung auf der Seite der Dienstimplementierung für den Webservice-Client den Vorteil, dass dieser im Vorfeld über potentielle Fehlerszenarien beim Aufruf einer Webserviceoperation aufgeklärt ist. Damit können bereits zur Entwicklungsphase in der Clientimplementierung möglich auftretende Ausnahmen berücksichtigt werden und programmtechnisch auf diese reagiert werden. Allerdings steigt einhergehend mit einer feineren Fehlerbehandlung auch der Entwicklungsaufwand auf Webservice-Clientseite an: Die Clientimplementierung muss auf verschiedene Exception-Typen reagieren können.

JAX-WS bietet für die Definition von Java-Exceptions auf Dienstseite die @WebFault-Annotation (Listing 3, Zeile 7), mit der beim Deployment einer derlei gekennzeichneten Exception das WSDL-Element „fault“ einer Operation der Servicebeschreibung hinzugefügt wird. Durch das standardisierte WSDL-Fault-Element wird eine Plattformunabhängigkeit erzielt: Im Fehlerfall wird die, von Dienstseite geworfene Java-Exception auf eine SOAP-Fault-Nachricht abgebildet, die dem Client zugesandt wird. Anschließend bildet JAX-WS in Verbindung mit JAXB den SOAP-Fault wieder ab auf eine Java-Exception und innerhalb der Client-Implementierung kann dementsprechend darauf reagiert werden.

```
5  import validation.types.DateValidateFault;
6
7  @WebFault(targetNamespace="http://validation.types")
8  public class DateValidateException extends Exception{
9
10     private static final long serialVersionUID = 4342594276340594799L;
11
12     private DateValidateFault dateValidateFault;
13
14     public DateValidateException(String msg, DateValidateFault dateValidateFault)
15     {
16     super(msg);
17     this.dateValidateFault = dateValidateFault;
18     }
19     public DateValidateFault getFaultInfo(){
20         return this.dateValidateFault;
21     }
22 }
23
```

Listing 3 - Exception mit @WebFault-Annotation

Die Exception beinhaltet ein Objekt des komplexen Datentyps „DateValidateFault“ (Zeile 12). Dieser wurde separat erstellt, um Detailinformationen zu einem Fehler zu beschreiben. Anhand dieses komplexen Datentyps soll im Listing 4 exemplarisch gezeigt werden, wie eine Klasse mit speziellen Annotationen versehen werden muss, damit beim späteren Deployment automatisch durch JAX-WS in Verbindung mit JAXB ein Mapping auf einen korrespondierenden XSD-Datentypen vorgenommen wird. Der abgebildete XML-Datentyp wird dann in einer sogenannten XSD-Datei (XML-Schema) auf dem Applikationsserver gespeichert. Die ebenfalls nach dem Deployment auf den Applikationsserver gespeicherte WSDL-Datei bindet diese XSD-Datei ein.

```
10 @XmlAccessorType(XmlAccessType.PROPERTY)
11 @XmlRootElement(name = "DateValidateFault")
12 public class DateValidateFault {
13
14         private String date;
15         private int errorcode;
16
17         @XmlElement(name = "occure-date")
18         public String getdate() {
19             return this.date;
20         }
21         public void setdate(String date) {
22             this.date = date;
23         }
24
25         @XmlElement(name = "error-description")
26         public int getErrorCode() {
27             return errorcode;
28         }
29         public void setErrorCode(int errorcode) {
30             this.errorcode = errorcode;
31         }
32 }
```

Listing 4 - Java-Klasse, die einen Fehler vom Typ „DateValidate“ beschreibt

Eine mit der @XmlRootElement-Annotation versehene Java-Klasse wird von JAXB mittels des Marshalling-Vorgangs in ein Wurzelelement eines XML-Dokuments übertragen (Zeile 11). Ohne die Angabe dieser Annotation wird beim Marshalling-Versuch durch JAXB eine Exception geworfen. Bei einer allein mit dieser Annotation ausgestatteten Java-Klasse bildet JAXB für jedes Attribut und jedes Getter-/Setter-Methodenpaar ein XML-Element mit gleichem Bezeichner. [1, p. 426]

Möchte man feiner steuern, welche Attribute oder Getter- und Setter-Methoden von JAXB in das XML-Schema übernommen werden sollen, kann ergänzend auf Klassenebene die Annotation @XmlAccessorType verwendet werden (Zeile 10). Diese Annotation bringt die Optionen PUBLIC_MEMBER, FIELD, PROPERTY und NONE mit [5]:

- Bei der Standardoption PUBLIC_MEMBER werden alle Attribute und Getter-/Setter-Methoden mit dem Gültigkeitsbereich „public“ an das XML-Schema gebunden
- Bei FIELD wird jedes nicht statische Attribut in XSD abgebildet
- Bei PROPERTY wird für jedes Getter-/Setter-Methodenpaar ein XML-Element gebildet
- Mit der Option NONE werden alle Attribute und Getter-/Setter-Methoden unterdrückt, es sei denn, es wird die @XmlElement-Annotation auf einzelne Attribute oder Getter-/Setter-Methoden angewandt

Mit der @XmlElement-Annotation können einzelne Attribute oder Getter-/Setter-Methodenpaare mit dem Gültigkeitsbereich „private“, „package“ oder „protected“ explizit an XML-Elemente gebunden werden. Zudem wird diese Annotation häufig verwendet, um mit einer Reihe von Parametern das von Java in XML umgeformte Element mit zusätzlichen Eigenschaften auszustatten. Hier kann u.a. ein Attribut umbenannt werden oder als Pflichtfeld deklariert werden (Zeile 17 oder Zeile 25). [1, p. 426]

Die in Listing 4 dargestellte Java-Klasse wird durch JAXB in das nachfolgende XML Schema überführt:

```
- <xs:complexType name="dateValidateFault">
  - <xs:sequence>
      <xs:element name="error-description" type="xs:int"/>
      <xs:element name="occure-date" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
```

Listing 5 - Durch JAXB ereugte XSD-Datei (eingebunden in WSDL-Datei)

Generell sind trotz der im Web dokumentierten Abbildungsregeln von Java-Objekten nach XML-Elementen selbst verschiedenste Konstellationen von JAXB-bezogenen Annotationen auszutesten, um die Abbildungszusammenhänge zu verstehen. Es konnte beispielsweise erkannt werden, dass die Angabe von @XmlElement(required=true) nur bezogen auf komplexe Datentypen wie String oder Wrapper-Klassen von JAXB beim Marshalling-Vorgang wahrgenommen wird, bei primitiven Datentyp wie int jedoch nicht. So würde bei der Umformulierung des String-Attributes „date“ (Listing 4 Zeile 17) auf „required=true“ das erzeugte XSD-Element anstatt „minOccurs=0“ die Eigenschaft „minOccurs=1“ erhalten (Listing 5). Wenn aber in Zeile 26 das int-Attribut „errorcode“ mit „required=true bzw. false“ versehen wird, so missachtet JAXB diese Eigenschaft beim Marshalling-Vorgang. Falls dennoch ein optionales Zahlenelement benötigt wird, kann anstatt des primitiven Datentypen int die Wrapper-Klasse Integer verwendet werden.

Java-Deklaration	XSD-Schema
@XmlElement(required=false) private int test;	Missachtung der Eigenschaft “required” von JAXB: <xs:element name="test" type="xs:int"/>
@XmlElement(required=false) private Integer test2;	Umsetzung der Eigenschaft “required” von JAXB: <xs:element name="test2" type="xs:int" minOccurs="0"/>

5.3 Deployment

Wie bereits im vorherigen Kapitel 5.2 angedeutet, werden die benötigten Webservice-Artefakte wie etwa die WSDL-Datei bei dem Deployment-Vorgang auf den Glassfish Applikationsserver automatisch erzeugt. Um die Bereitstellung des EAR-Projektes aus Eclipse heraus anzutreiben, ist im Kontextmenü des angelegten Glassfish 3.1.2 Applikationsservers innerhalb der Eclipse-Server-View der „Add and Remove“-Dialog anzuwenden.

In der webbasierten Glassfish Admin Konsole kann daraufhin getestet werden, ob die Bereitstellung erfolgreich durchgeführt werden konnte und außerdem, ob der Applikationsserver das EAR-Projekt richtigerweise auch als Webservice interpretiert. Auf die Glassfish Admin Console kann über die URL „http://localhost:<festgelegter Glassfish Port>/common/index.jsf“ zugegriffen werden. Wie in der Abbildung 10 dargestellt, ist in der

tabellarischen Ansicht „Module und Komponenten“ zu prüfen, ob in der Spalte „Aktion“ der Hyperlink „End Point anzeigen“ erscheint. Der Grund für die Prüfung ist folgender:

Selbst wenn durch Eclipse keine syntaktischen Fehler in den inkludierten Modulen des EAR-Projektes festgestellt werden konnten und das EAR-Projekt bei den Anwendungen in der Admin Konsole gelistet wird, kann der Fall auftreten, dass Glassfish die im EAR-Projekt inkludierten Module nicht als Webservice interpretiert. Ein Grund dafür ist beispielsweise die falsche Verwendung der Webservice-spezifischen Annotationen in dem Entwicklungsprojekt. In diesem Fall würde Glassfish das (EJB-) Modul nicht der Webservice-Engine Metro zuordnen und somit würde diese auch keine WSDL-Datei generieren. Demzufolge würde der Hyperlink „End Point anzeigen“ nicht erscheinen. Deshalb kann nur von einem vollständig erfolgreichen Deployment des Webservices ausgegangen werden, wenn der Hyperlink angezeigt wird.

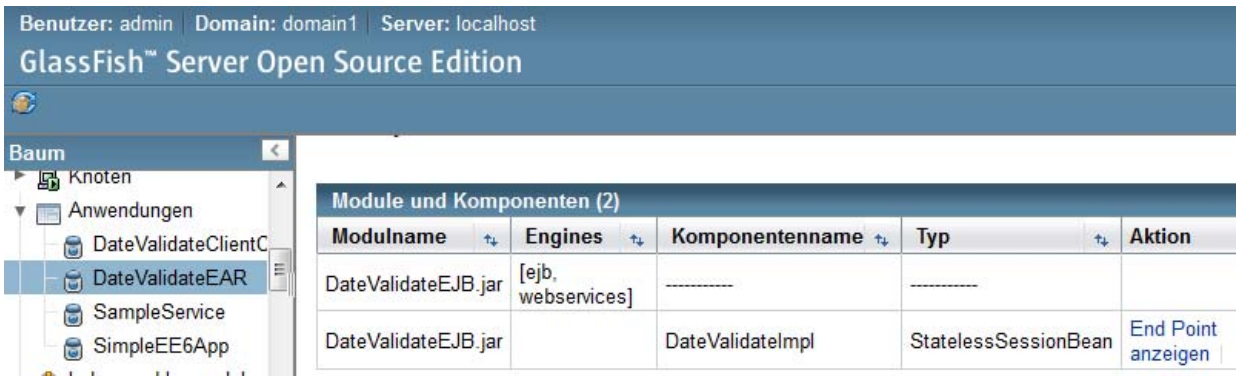

Abbildung 10 - Zugriff auf die generierte WSDL-Datei innerhalb der Glassfish Admin Konsole

Über den „End Point anzeigen“-Link wird auf einer Folgeseite auf den Server-Ablageort der erzeugten WSDL-Datei über eine URL verwiesen. Zusätzlich besteht die Möglichkeit, über den webbasierten Webservice-Tester der Admin Konsole einen erstmaligen Webservice-Aufruf zu testen. In der nachfolgenden Abbildung 11 ist der Zugriff auf die WSDL-Datei mittels Browser dargestellt. Hier ist im auskommentierten Kopfteil der WSDL-Datei erkennbar, dass der sich im Webcontainer des Applikationsservers befindliche Metro-Webservice-Stack die Datei erzeugt hat.

Abbildung 11 - Im Browser angezeigte WSDL-Datei

Mittels der vorgelegten WSDL-Datei kann kontrolliert werden, ob alle mit Webservice-Annotationen gekennzeichneten Java-Elemente korrekt auf XML-Elemente abgebildet wurden. Nachfolgend werden mögliche Kontrollen hinsichtlich der „DateValidate“-Webserviceentwicklung veranschaulicht:

1. Die mit @WebFault versehene Exception „DateValidationException“ wird auf das entsprechende WSDL-Element Fault transformiert.
2. Alle Eigenschaften der restriktiven Annotation @WebParam und @WebResult, die in dem Interface „DateValidate“ definiert sind, fließen in die entsprechenden WSDL-Elemente ein.

Zu beachten ist, dass datentypspezifische Informationen nicht direkt in der WSDL-Datei enthalten sein können, sondern auch in separaten XSD-Dateien, die mittels Import-Anweisung in die WSDL-Datei eingebettet sind (siehe Abbildung 11)[3].

Alternativ zu dem Webservice Tester der Glassfish Admin Konsole kann auch das OpenSource Tool SOAPUI verwendet werden, um Aufrufe eines Webservice Clients zu simulieren. Generell kann angeraten werden, strukturierte Tests mittels JUnit und nicht mit SOAPUI durchzuführen. Auch falls die ursprüngliche Testerstellung mit einem höheren Aufwand verbunden sein könnte, ergibt sich bezogen auf die spätere Webservice-Weiterentwicklung der Hauptvorteil, dass sich durch den hohen Automatisierungsgrad bei der Testdurchführung weniger Aufwand gegenüber der Testdurchführung mit SOAPUI ergibt. Deshalb wird im nachfolgenden Kapitel 5.4 anhand der „DateValidate“-Webserviceentwicklung detailliert auf das strukturierte Testen eines Webservices mit JUnit eingegangen. Dennoch soll verkürzt auf SOAPUI Bezug genommen werden, da hier die durch die Applikation automatisch erzeugten SOAP-Nachrichtengerüste eingesehen werden können. Auf diese Art kann der grundsätzliche SOAP-Nachrichtenaufbau der Anfragenachricht und der Antwortnachricht besser veranschaulicht werden.

[3] Prinzipiell bestünde bei der manuellen Erstellung einer WSDL-Datei wie beim Contract-First-Ansatz auch die Möglichkeit, Datentypen direkt innerhalb der WSDL-Datei zu definieren anstatt diese in Form einer XSD-Datei einzubetten [2, p. 137]. Aufgrund einer besseren Strukturierung ist jedoch die Einbettung von gesonderten XSD-Dateien zu bevorzugen.

Innerhalb der Applikation SOAPUI kann ein neues Projekt durch die Einbeziehung einer WSDL-Datei erstellt werden, das automatisch ein SOAP-Nachrichtengerüst erstellt. Dieses Request-Nachrichtengerüst kann wie in der Abbildung 12 mit einem beispielhaften Datumsstring als Anfrageparameter manuell ergänzt werden. Im Anschluss kann in SOAPUI der Request abgesetzt werden. Der synchrone Webservice-Aufruf liefert unter der Voraussetzung, dass ein valides Datum im geforderten Pattern angefragt wurde, eine positive SOAP-Antwortnachricht im Stil von Abbildung 13.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:val="http://validation
   <soapenv:Header/>
   <soapenv:Body>
      <val:validateSQLFormatDate>
         <sqlDate>1987-06-12</sqlDate>
      </val:validateSQLFormatDate>
   </soapenv:Body>
</soapenv:Envelope>
```

Abbildung 12 - SOAP-Request-Nachrichtengerüst

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <ns2:validateSQLFormatDateResponse xmlns:ns2="http://validation/" xmlns:ns3="http://validation.types">
         <realDate>true</realDate>
      </ns2:validateSQLFormatDateResponse>
   </S:Body>
</S:Envelope>
```

Abbildung 13 - SOAP-Response-Nachrichtengerüst

Unter der Angabe eines ungültigen Datums empfängt der simulierende Client eine SOAP-Fault-Nachricht analog der Abbildung 14. Wie hier dargestellt, ist das auf SOAP-Nachrichtenformat abgebildete Java-Objekt „DateValidateException“ in die Standardstruktur einer SOAP-Fault-Nachricht eingebunden. Wäre anstatt einer DateValidateException z.B. eine Standard-ArrayIndexOutOfBoundsException von der Dienstimplementierung geworfen worden, so würde der SOAP-Fault ausschließlich einen „faultcode“ und einen nicht aussagekräftigen „faultstring“ enthalten.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Body>
      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
         <faultcode>S:Server</faultcode>
         <faultstring>Sie müssen das Pattern XXXX-XX-XX einhalten!</faultstring>
         <detail>
            <ns3:DateValidateException xmlns:ns3="http://validation.types" xmlns:ns2="http://validation/">
               <date>-06-</date>
               <errorcode>234</errorcode>
            </ns3:DateValidateException>
         </detail>
      </S:Fault>
   </S:Body>
</S:Envelope>
```

Abbildung 14 - SOAP-Fault-Nachricht

5.4 Strukturiertes Testen eines Webservice mit JUnit

Anhand eines herkömmlichen Java-Projektes soll das Aufsetzen eines JUnit-Testszenarios veranschaulicht werden. Da die JUnit-Testklasse genauso, wie die später zu entwickelnde Clientimplementierung als Webservice-Aufrufer agiert, benötigt diese Zugriff auf den Webservice. Deshalb sind zunächst aus der WSDL-Datei die gebrauchten Dienstnutzer-Stub-Klassen zu generieren. Hierfür wurden zwei Möglichkeiten in Betracht gezogen:

1. Der Eclipse Wizard zum Erstellen eines neuen Web Service Clients
2. Die Erstellung eines Ant-Skriptes, welches intern das JAX-WS Programm „wsimport.exe“ anwendet

Da der Eclipse Wizard nur die Möglichkeit bietet, nicht mittels JAX-WS (implizit Metro) sondern unter Verwendung des Apache Axis, Axis 2- oder CXF Webservice Stacks aus der WSDL-Datei die Dienstnutzer-Stub-Klassen zu generieren, wurde letztendlich die Variante des Ant-Skriptes präferiert. Wie in der Java-Projektstruktur der Abbildung 15 sichtbar, wurde das Ant-Skript „wsdl2java.xml“ auf oberster Projektebene abgelegt. Anzumerken ist, dass die abgebildete Projektstruktur bereits die vollendente Fassung nach der Skriptausführung und der Testklassenentwicklung aufweist.

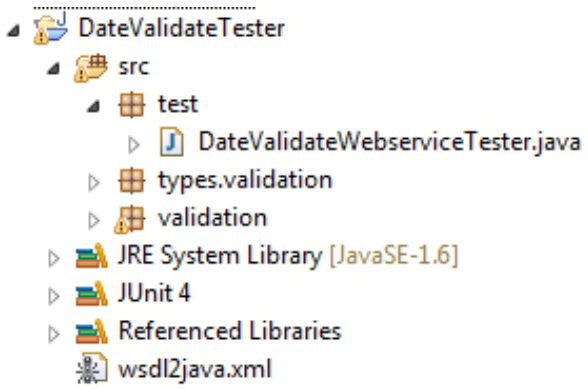

Abbildung 15 - Projektstruktur des JUnit-Tests

```
<project name="DateValidateConsumerJAX" default="generate-ws-client" basedir=".">
    <target name="generate-ws-client" >
        <exec executable="C:/Program Files/Java/jdk1.6.0_24/bin/wsimport">
        <arg line="-keep -s src -d build/classes -verbose http://tobyboy-pc:8080/DateValidateImplService/DateValidateImpl?wsdl"/>
        </exec>
    </target>
</project>
```

Abbildung 16 - Ant-Skript zur Generierung der Dienstnutzer-Stub-Klassen aus einer WSDL

Da der wsimport-Befehl nicht in der Standardpalette des Ant-Befehlssatzes enthalten ist, muss die ausführbare Befehlsimplementierung „wsimport.exe“ innerhalb des Ant-Skriptes eingebunden werden. Um in Ant auch nicht nativ enthaltene Befehle wie „wsimport“ verwenden zu können, existiert hierfür der „exec“-Ant-Task.

Mithilfe des Tasks (Zeile 3 der Abbildung 16) erfolgt die Einbettung des wsimport-Befehls. Die Befehlszeile in Zeile 4 beinhaltet folgende Funktionalität:

- Der „keep“-Parameter gibt an, dass die Dienstnutzer-Stubs auch als Java-Dateien generiert werden sollen
- Mit dem „s“-Parameter wird der „src“-Ordner als Ablageort für die generierten Java-Stub-Klassen festgelegt.
- Mit dem optionalen Parameter „-d“ werden zusätzlich .class-Dateien der Dienstnutzer-Stubs mitgeneriert.
- Der optionale „verbose“-Parameter gibt getätigte Compilervorgänge bei der Skriptausführung in der Konsole aus
- Als letzter Pflichtparameter muss entweder der URL- oder der lokale Festplattenablageort der WSDL-Datei als Quelle für die Generierung angegeben werden

Die eigentliche Ausführung des Ant-Skriptes erfolgt über das Windows-Kommandozeilentool. Für diesen Zweck ist zuerst zu dem Pfad zu navigieren, in dem sich der Ablageort des Ant-Skriptes befindet. Vorausgesetzt, dass das Building-Tool Ant auf dem lokalen Rechner installiert ist, kann das Skript im Anschluss mit „ant –f wsdl2java.xml“ ausgeführt werden. Der Generier-vorgang erstellt daraufhin in der Projektstruktur die Pakete „types.validation“ und „validation“ einschließlich deren Stub-Klassen.

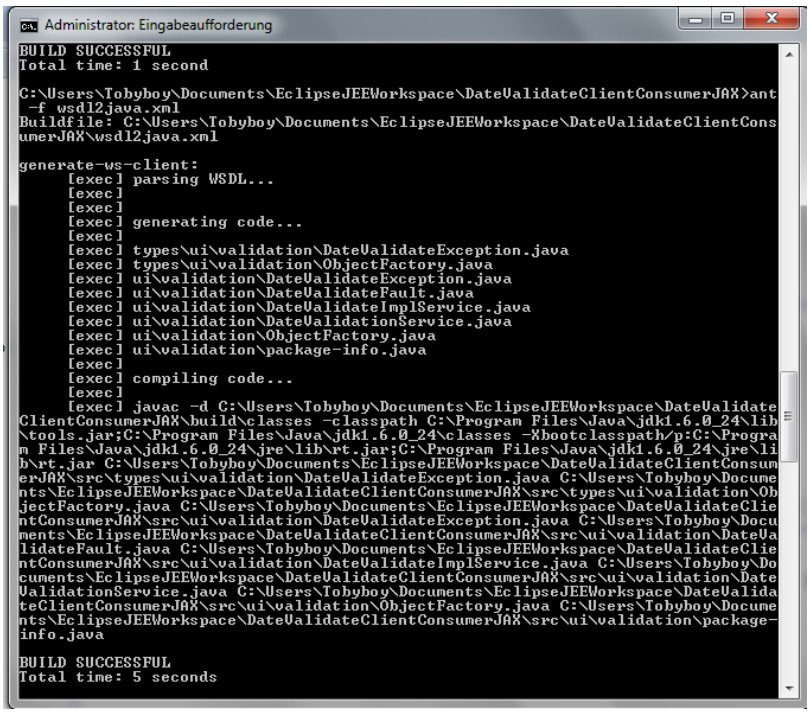

Abbildung 17 - Ausführung des Ant-Skripts zur Generierung der Dienstnutzer-Stubs

Als Ausgangsbasis der Testentwicklung wurde zunächst über eine Konfiguration der Build Path des Java Projektes um die JUnit4 Bibliothek erweitert. Auf dem nachfolgenden Listing 6 ist die erstellte JUnit-Testklasse aufgeführt. Ein besonderes Augenmerk liegt hierbei auf der „before“-Methode (Zeile 21): Diese beinhaltet die Instanziierung der Dienstnutzer-Stub-Klasse. Damit ist der Zugriff auf die vom Webservice veröffentlichten Weboperationen über Java-Methoden möglich. Aufgrund der @Before-Annotation wird die Instanziierung vor den Tests ausgeführt.

Die Testklasse wurde so konzipiert, dass diese zwischen drei verschiedenen Testfällen unterscheidet:

1. Testfall für gültige Testdaten
   -> Rückgabewert „true“ wird erwartet
2. Testfall für ungültige Testdaten, die nicht dem Muster „XXXX-XX-XX“ entsprechen
   -> Geworfene DateValidateException wird erwartet
3. Testfall für ungültige Testdaten, die trotz des richtigen Musters kein wahrheitsgemäßes Datum wiederspiegeln: Bsp: 2013-06-31, 2013-02-29
   -> Rückgabewert „false“ wird erwartet

```
 8 import validation.DateValidateException;
 9 import validation.DateValidateImplService;
10 import validation.DateValidationService;
11
12
13 public class DateValidateWebserviceTester {
14
15     boolean result = false;
16
17     DateValidateImplService service;
18     DateValidationService port;
19
20     @Before
21     public void before(){
22         service = new DateValidateImplService();
23         port = service.getDateValidateImplPort();
24
25     }
26
27     @Test
28     public final void testWithValidDates() {
29
30         try {
31
32             result = port.validateSQLFormatDate("2012-02-29");
33             assertTrue(result);
34
35             result = port.validateSQLFormatDate("2012-06-30");
36             assertTrue(result);
37
38             result = port.validateSQLFormatDate("2013-1-1");
39             assertTrue(result);
40
41         } catch (DateValidateException e) {}
42
43     }
44
45     @Test
46     public void testWithInvalidDateOtherStringMustThrowException() {
47         try {
48
49             result = port.validateSQLFormatDate("Test");
50             fail();
51
52         } catch (DateValidateException e) {}
53     }
54
55
56     @Test
57     public void testWithInvalidDatesLeapYearOrOverMonthRange() {
58         try {
59
60             result = port.validateSQLFormatDate("2013-06-31");
61             assertFalse(result);
62
63             result = port.validateSQLFormatDate("2013-02-29");
64             assertFalse(result);
65
66         } catch (DateValidateException e) {}
67     }
68 }
```

Listing 6 - JUnit-Testklasse

Es ist hinzuzufügen, dass alle auf dem Listing dargestellten Testdaten, die jeweils den drei Testfällen zugeordnet sind, nur einen Teilbereich aller möglichen Testszenarien abdecken.

Die Durchführung der Tests ist natürlich erst möglich, wenn der Webservice auf dem bereits laufenden Glassfish Applikationsserver bereitgestellt ist. Der Gesamttest ist gelungen, sobald alle inkludierten Tests erfolgreich durchlaufen sind.

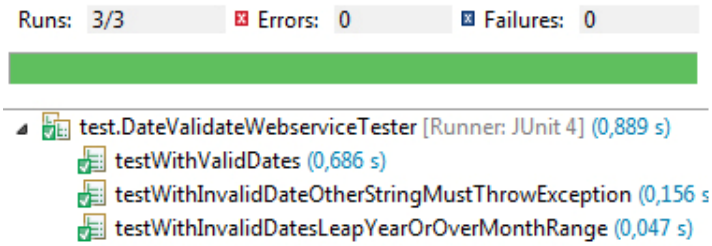

Abbildung 18 - Ergebnis der durchgeführten JUnit-Tests

5.5 Entwicklung des Webservice-Client

Für die Clientimplementierung wurde anstatt eines EJB-Projektes, das für den Dienst verwendet wurde, ein dynamisches Webprojekt gewählt. Der beabsichtigte Aufbau des Webprojekts entspricht einer JSP-Seite, die unter der Nutzung von Dienstnutzer-Stub-Klassen einen vom Benutzer ausgefüllten Formularwert an den Webservice übermittelt. Da es sich um ein synchrones Webservice-Szenario handelt, wird dem Client über den zurückgegebenen booleschen Wert der Webserviceoperation die Datumsgültigkeit mitgeteilt. Das dynamische Webprojekt soll ebenso, wie das EJB-Projekt bei der Dienstimplementierung, einem neuen EAR-Projekt zugeordnet werden. Dies ist prinzipiell nicht zwingend, da dynamische Webprojekte auch autark auf den Glassfish Applikationsserver bereitgestellt werden können.

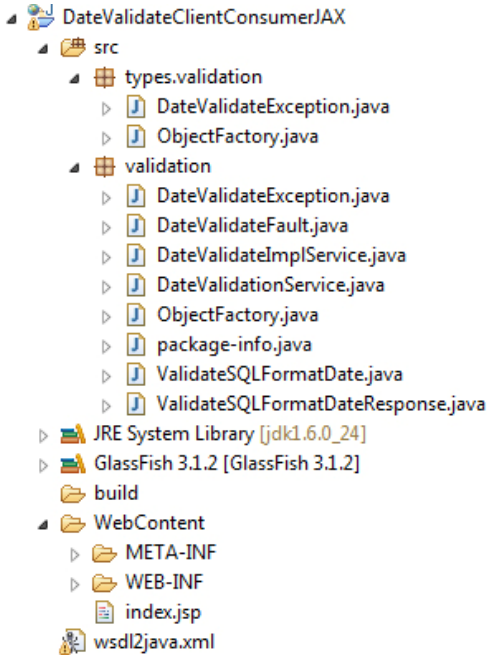

Abbildung 19 - Projektstruktur der Clientimplementierung

Zu Beginn der Entwicklung sind entsprechend der Schritte in Kapitel 5.4 die Dienstnutzer-Stub-Klassen zu generieren. Die Abbildung 19 zeigt die vollendete Projektstruktur.

Der Algorithmus der JSP-Seite sieht vor, ein ausgefülltes HTML-Formular via HTTP-POST an sich selbst zu verschicken. In den Zeilen 24 und 25 (Listing 7) wird der übermittelte Parameter aufgefangen und auf einen Nullwert geprüft. Das Quellcodesegment von Zeile 27-43 wird nur durchlaufen, wenn ein übermittelter String-Wert empfangen werden konnte. Über diesen Mechanismus wird dieser Teil beim erstmaligen Aufruf der JSP-Seite nicht ausgeführt.

Für den Zugriff auf den Webservice werden wie bei der JUnit-Testentwicklung die benötigten Dienstnutzer-Stub-Klassen importiert (Zeile 1-3) und der Webservice instanziiert (Zeile 28-29). Da es sich bei der aufgerufenen Webservicemethode „validateSQLFormatDate“ um eine Exception werfende Methode handelt, obliegt es dem Client alle potentiell auftretenden Ausnahmen abzufangen und darauf zu reagieren. Falls die Ausnahme „DateValidateException“ eintritt, wird die Fehlernachricht auf dem Browser angezeigt. Es ist auch möglich clientseitig aus der Fehlernachricht gezielt auf einzelne Attribute bzw. Getter-/Setter-Methoden wie z.B. „errorcode“ zuzugreifen.

```
1  <%@page import="validation.DateValidationService"%>
2  <%@ page import="validation.DateValidateImplService" %>
3  <%@ page import="validation.DateValidateException" %>
4  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
5      pageEncoding="ISO-8859-1"%>
6  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
7                        "http://www.w3.org/TR/html4/loose.dtd">
8  <html>
9    <head>
10         <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
11         <title>Datumsvalidierung</title>
12   </head>
13   <body>
14     <h1>Datumsvalidierung</h1>
15     <form action="index.jsp" method="post">
16             <table bgcolor="#f5f5f5" width="50%" height="20" cellspacing="2" cellpadding="2" border="
17                 <tr>
18                     <td>Datum im Format YYYY-MM-TT:</td>
19                     <td><input maxlength=10 type="text" name="date">
20                 </tr>
21             </table>
22         </form>
23     <%
24     String date = request.getParameter( "date" );
25     if(date != null)
26     {
27     try {
28         DateValidateImplService service = new DateValidateImplService();
29         DateValidationService port = service.getDateValidateImplPort();
30
31         %>
32         <table bgcolor="#f5f5f5" width="50%" height="20" cellspacing="2" cellpadding="2" border="0">
33                 <tr>
34                     <td>Korrektes Datum:</td>
35                     <td>
36                     <% out.println(port.validateSQLFormatDate(date)); %>
37                     </td>
38                 </tr>
39         </table>
40         <%
41     } catch (DateValidateException e) {
42         out.println("Fehlernachricht: "+ e.getMessage());
43     }
44     }
45         %>
46   </body>
47 </html>
```

Listing 7 - Quellcode der JSP-Seite

Nachfolgend ist die Ausführung der JSP-Seite im Browser sowohl in der Erfolgs- als auch in der Fehlervariante dargestellt:

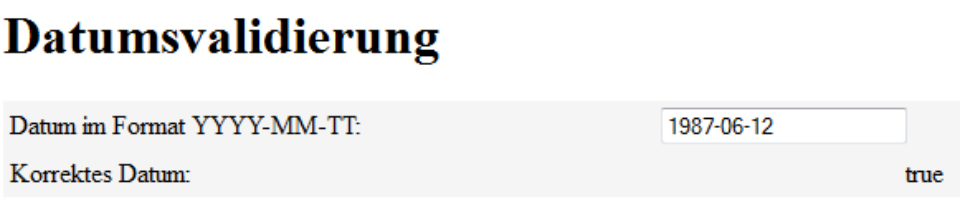

Abbildung 20 - Browseransicht bei erfolgreich ausgeführter Datumsprüfung nach JSP-Ausführung

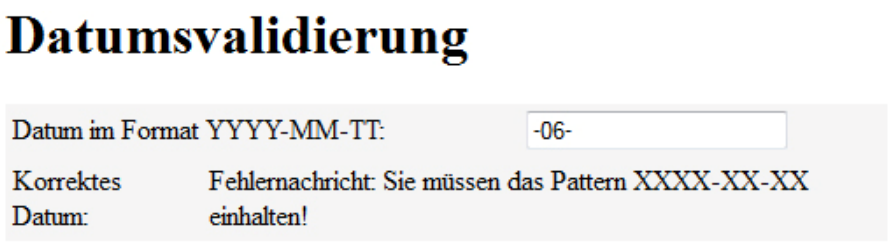

Abbildung 21 - Browseransicht bei ausgelöster Exception nach JSP-Ausführung

6 Webservice Interoperability Technologies

Im Rahmen einer gemeinsamen Initiative von Sun und Microsoft wurden mehrere ausgewählte WS-*-Standards zu der Dachspezifikation WSIT kombiniert. Ziel war die Schaffung von Kompatibilität zwischen der Java-Plattform und Microsofts Windows Communication Foundation bezogen auf serviceorientierte Architekturen [2, p. 34]. Gerade im Hinblick auf erweiterte Webservice-Technologien, die sich u.a. auf die Sicherheit, Nachrichtenübertragung oder Transaktionssicherheit von Webservices beziehen, sollte die Interoperabilität zwischen den beiden technologisch unterschiedlichen Plattformen sichergestellt werden [6]. WSIT umfasst u.a. die folgenden WS-*-Standards angeordnet nach unterschiedlichen QoS-Kriterien[4]:

Bootstrapping

- WSDL
- WS-MetadataExchange
- WS-Policy

Sicherheit

- WS-SecureConversation
- WS-Trust
- WS-Security
- WS-SecurityPolicy

Nachrichtentransfer

- SOAP
- WS-Addressing
- SOAP-MTOM

Nachrichtenübertragung

- WS-ReliableMessaging
- WS-Coordination
- WS-AtomicTransaction

[4] Des Weiteren beinhaltet WSIT als Grundstock für die obig aufgezählten Webservice-Standards die Kern-XML-Standards XML, XML Namespace, XML Infoset und XML Schema und noch andere WS-*-Standards [6].

Einhergehend mit der Formulierung der WSIT-Spezifikation wurde eine Referenzimplementierung unter dem gleichnamigen Projekt vollzogen. Die WSIT-Implementierung ist ein in den Webservice-Stack Metro eingebettetes Subsystem [6].

Um ein tiefergehendes Verständnis über WSIT zu erhalten, soll in den nachfolgenden zwei Kapiteln detailliert auf die Reliable-Messaging-Technologie eingegangen werden. Die Intension dabei ist nicht vorwiegend das genaue Betrachten der WS-ReliableMessaging-Spezifikation, sondern eher eine möglichst genaue Beleuchtung der technischen Implementierung sowie die praktische Anwendbarkeit für den Entwickler eines Webservices.

6.1 Reliable Messaging in der Theorie

In bestimmten Webservice-Szenarien kann die Anforderung einer garantierten Übertragung mehrerer einzelner SOAP-Nachrichten zu einem Webservice-Endpunkt bestehen. Dabei stellen diese unabhängig versendeten SOAP-Nachrichten eine logisch zusammengehörende Sequenz dar. Als optionalen Zusatz könnte auch verlangt werden, dass die SOAP-Nachrichten in der gleichen Reihenfolge bei dem Webservice-Endpunkt ankommen müssen, wie sie von dem Webservice-Client abgesendet wurden. Ein Anwendungsfall dafür ist das nachfolgende Beispiel:
Ein Sendersystem mit einer Datenführerschaft versorgt nächtlich ein Empfängersystem mit Informationen zu Aufträgen. Die Versendungsreihenfolge der SOAP-Nachrichten muss auf der Empfangsseite beibehalten werden, da der schreibende Webservice eine bestimmte Reihenfolge der DB-Insert-Operationen für jeweils eine Nachrichtensequenz beachten muss. Bei der Nichteinhaltung könnte beispielsweise der Fehlerfall auftreten, dass eine Schreiboperation, die einen Tupel einer Assoziationstabelle zeitlich vor einer anderen Schreiboperation absetzt, die einen referenzierten Tupel einer Stammdatentabelle in die Datenbank schreibt.

Zusammenfassend formuliert besteht also der Bedarf einer zuverlässigen Übertragung von zusammengehörenden, aber einzeln versendeten SOAP-Nachrichten zu einem Webservice-Endpunkt. Unter der Beachtung der optionalen Erweiterung muss jede Sequenz an SOAP-Nachrichten in der ursprünglichen Übertragungsreihenfolge beim Webservice-Endpunkt ankommen. Jedoch kann HTTP als standardmäßiges Übertragungsprotokoll für SOAP-Nachrichten erstens keine zuverlässige Übertragung einer SOAP-Nachricht sicherstellen und zweitens auch nicht die erweiterte Anforderung gewährleisten.

HTTP ist ein zustandsloses und verbindungsorientiertes Kommunikationsprotokoll. Innerhalb des OSI-Schichtenmodells ist das Kommunikationsprotokoll der Anwendungsschicht zugeordnet und benutzt wiederum standardmäßig für das Verbindungsmanagement das Transportprotokoll TCP der Transportschicht.

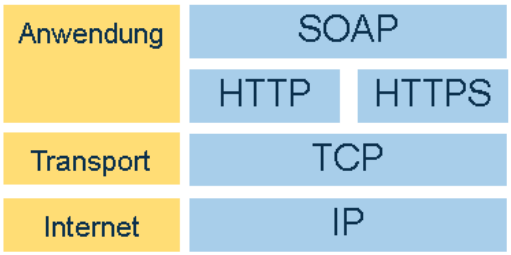

Abbildung 22 - SOAP-Protokollstack (sinngemäß übernommen [2, p. 92])

Für eine HTTP-Kommunikation wird mittels Drei-Wege-Handshake ein TCP-Verbindungsaufbau vollzogen. Die Abbildung 23 veranschaulicht, dass eine TCP-Verbindung standardmäßig nur über eine HTTP-Kommunikation hinweg aufrecht erhalten bleibt. Mit anderen Worten bleibt die Verbindung nur so lange bestehen, bis ein Nachrichtenpaar (HTTP-Request-/HTTP-Response-Nachricht) ausgetauscht wurde.

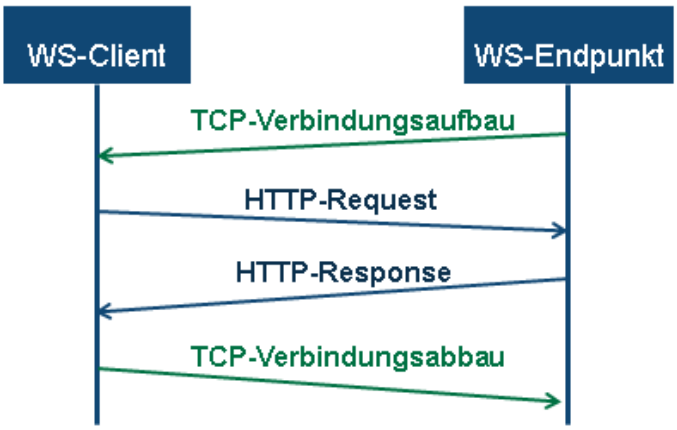

Abbildung 23 - HTTP-Kommunikation (sinngemäß übernommen aus [7, p. 249])

Nachfolgend soll erläutert werden, warum HTTP als Standard-Übertragungsprotokoll für SOAP-Nachrichten nicht in der Lage ist, den hier benötigten Grad an Quality-of-Service zu bewerkstelligen:

1. Das HTTP-Protokoll kann keine garantierte Übertragung von SOAP-Nachrichten sicherstellen.

   Der vorhin genannte Begriff „zustandslos“ wird deshalb mit HTTP assoziiert, weil sowohl der Sender als auch der Empfänger keine Statusinformationen zu einer Kommunikation verwalten [7, p. 248]. Zudem werden durch HTTP keine Mechanismen hinsichtlich Nachrichten-Quittierungsvarianten oder Varianten der Nachrichten-Übertragungswiederholung implementiert. Angenommen eine SOAP-Nachricht wird via HTTP-Request über mehrere Netzknoten übermittelt und es tritt die Fehlersituation auf, dass die Verbindung zum Netzwerk verloren geht. In diesem Fall bleiben beide Kommunikations-Endpunkte darüber uninformiert, dass die SOAP-Nachricht auf dem Weg zum Webservice verloren gegangen ist.

2. Das HTTP-Protokoll stellt keine Mechanismen bereit, eine einmalige Zustellung von mehreren SOAP-Nachrichten in der ursprünglichen Versandreihenfolge an ein Empfängersystem zu garantieren.

   Die Tatsache, dass bei HTTP keine Zustandsinformationen der Kommunikation gepflegt werden hat die folgende Konsequenz: Mehrere einzeln versandte SOAP-Nachrichten im Empfängersystem können nicht als eine zusammengehörende Sequenz interpretiert werden. Hierzu müssten auf Anwendungsebene eigene Mechanismen umgesetzt werden. Die Empfängerseite braucht vorerst Informationen über Sequenz- und Nachrichtenidentifizierung. Zusätzlich benötigt der Empfänger verwaltete Nachrichtenwarteschlangen, in denen korrelierte SOAP-Nachrichten zwischengepuffert werden können, bis die Nachrichtensequenz vollständig übermittelt wurde. Letztendlich muss eine Implementierung des Empfängersystems eine eventuelle Umsortierung vornehmen können.

Durch die Reliable Messaging Technologie wird die zuverlässige Übertragung von SOAP-Nachrichten zu einem Webservice-Endpunkt sichergestellt. Diese Technik garantiert die Auslieferung einer Sequenz von SOAP-Nachrichten wahlweise entweder nach der Zustellgarantie „Exactly Once“ oder „Exactly Once In Order“. Bei EO wird die Sequenz von SOAP-Nachrichten genau einmalig zugestellt und EOIO stellt diese Zustellung in der ursprünglichen Übertragungsreihenfolge sicher.

Im Folgenden soll die genaue Funktionsweise der RM-Technologie erläutert werden:
In Kapitel 3 wurde bereits die Webcontainer-Anatomie des Glassfish Applikationsservers beleuchtet. Unter Anderem wurde darauf hingewiesen, dass der Teil des Webservice-Stacks-Metro, der die WSIT-Komponenten enthält, nicht in dem verwendeten JDK enthalten ist, sondern bereits separat auf dem Glassfish Applikationsserver vorinstalliert ist. Wird die WSIT-Funktionalität wie etwa die Reliable-Messaging-Technologie für ein Webservice-Szenario gewünscht, so kann durch bestimmte programmatische bzw. konfigurierbare Ergänzungen (siehe Kapitel 6.2) sowohl in der Client- als auch in der Webserviceimplementierung diese aktiviert werden. Bei den richtig getätigten Ergänzungen binden somit die JAX-WS-Laufzeitumgebungen beider Glassfish Applikationsserver (Webservice Client und Webservice Endpunkt) benötigte Reliable-Messaging-Module ein. Die nachfolgende Abbildung 24 zeigt eine vereinfachte Darstellung eines asynchronen Nachrichtenaustauschs bei der Aktivierung von Reliable Messaging.

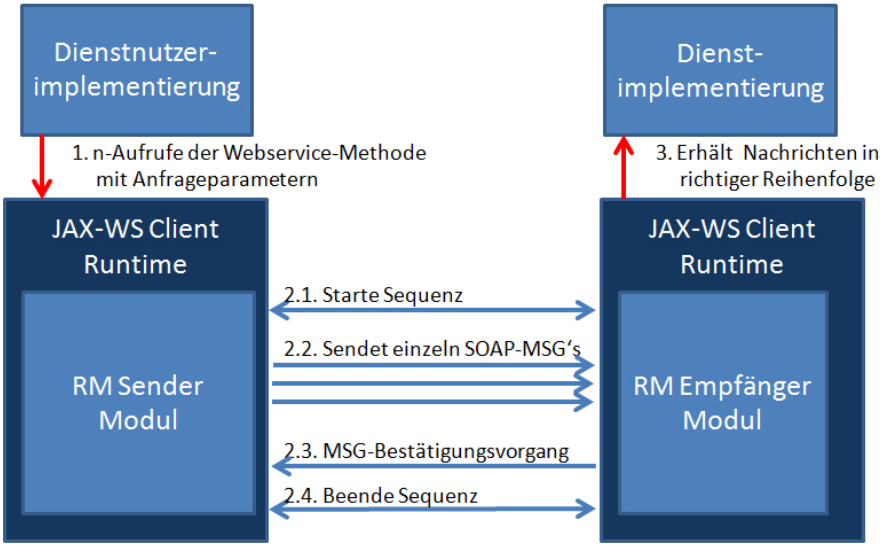

Abbildung 24 - Anatomie der JAX-WS-Runtime mit RM-Technologie (sinngemäß übernommen aus [4] und [8, p. 29])

Tritt die Fehlersituation auf, dass eine SOAP-Nachricht bei der Übertragung verloren geht, so initiiert das RM-Modul der JAX-WS Client Runtime eine wiederholte Übertragung. Als Grundvoraussetzung, dass eine Übertragungswiederholung nach einer bestimmten Strategie stattfinden kann, ist im Header einer jeden SOAP-Nachricht mit Reliable-Messaging –Erweiterung eine Sequenzidentifikationsnummer, die die aktuelle Sequenz identifiziert. Zusätzlich enthält der Header eine eindeutig identifizierbare Nachrichtennummer, die die Position der SOAP-Nachricht in der Sequenz angibt. [8, p. 28] Es soll bezogen auf die nachfolgende Abbildung 25 darauf eingegangen werden, dass bei der RM-Technologie als Übertragungswiederholungsstrategie ein positiv kumulatives Quittierungsverfahren angewandt wird. Innerhalb des Headers der versandten MSG 3 wird zusätzlich vom RM-Client die Information mit gesendet, dass es sich um die letzte SOAP-Nachricht der aktuellen Sequenz handelt. Daraufhin bestätigt der RM-Server alle bisher erhaltenen Nachrichten (positiv kumulativ). (abgeleitet aus [9, pp. 33-34]).

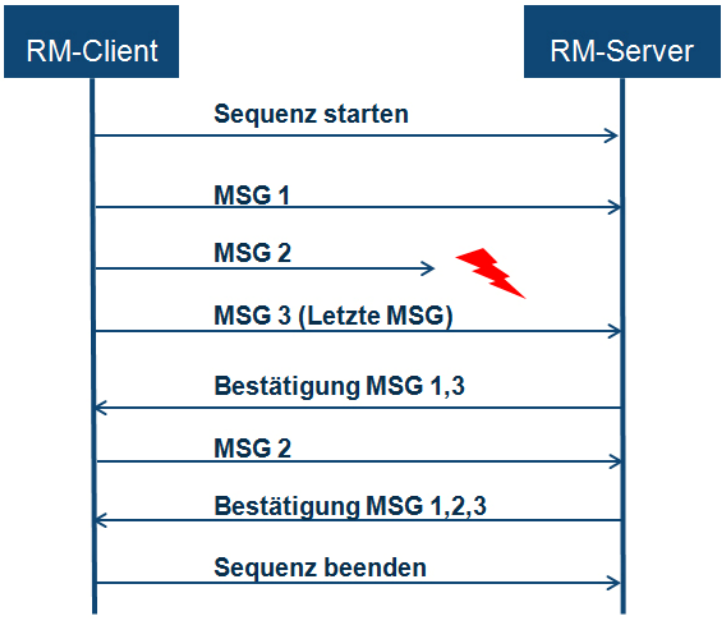

Abbildung 25 - RM-Nachrichtenfluss inklusive Übertragungswiederholung (sinngemäß übernommen aus [10])

Das RM-Modul des Clients hält sich lokale Kopien der SOAP-Nachrichten bis deren Quittierung vom RM-Modul des Servers eingetroffen ist. Falls EOIO aktiviert ist, puffert das Empfänger-RM-Modul ebenfalls SOAP-Nachrichten in seinem lokalen Speicher. Erhält nun das RM-Modul des Empfängers eine Sequenz von anders geordneten SOAP-Nachrichten als die ursprüngliche Übertragungsreihenfolge war, dann ordnet das RM-Modul des Empfängers die eingetroffenen SOAP-Nachrichten um in die korrekte Sequenz, bevor es diese an die darüber liegende Dienstimplementierungsschicht hochreicht. [8, p. 29] Die Neusortierung in die ursprüngliche Versendungsreihenfolge wird mithilfe der übermittelten Nachrichtennummern vollzogen.

6.2 Reliable Messaging in der Praxis

Für die praktische Umsetzung eines Webservice-Szenarios auf der RM-Technologie wurde anstatt Eclipse Juno als Entwicklungsumgebung bewusst NetBeans 7.2.1 gewählt. Laut Meinung der Autoren des Buches „Java Webservices in der Praxis“ Oliver Heuser und Andreas Holubek, die im Rahmen der Literatur sehr stark auf Metro und WSIT eingehen, integriert die Netbeans IDE die WS*-Standards durchgängiger und konsequenter als die Eclipse IDE. Diese Aussage wurde aufgrund von Beobachtungen von den Autoren getroffen und konnte im Rahmen eines persönlichen E-Mail-Schriftverkehrs mit Oliver Heuser bekräftigt werden.

Zunächst werden die zu tätigenden Schritte auf Dienstseite beschrieben. Da der Fokus darauf liegt, wie die RM-Funktionalität für einen Webservice aktiviert werden kann, wurde in Netbeans - innerhalb des „New Project“-Dialogs unter der Kategorie „Samples“ - das bereits aufgesetzte Webservice-Projekt namens „Calculator (Java EE 6)“ gewählt. Listing 8 zeigt die Anwendungslogik des CalculatorWS mit einer add-Webserviceoperation, die die Summe von zwei mitgegebenen Parametern zurückgibt.

```java
@WebService()
public class CalculatorWS {
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "i") int i, @WebParam(name = "j") int j) {
        int result = i + j;
        return result;
    }
}
```

Listing 8 - Anwendungslogik des Calculator-Webservice

Auf der Abbildung 26 ist ersichtlich, wo die Dialogbox für die erweiterten WSIT-Eigenschaften eines Webservice aufzufinden ist. Für alle hier konfigurierbaren WSIT-Einstellungen wird im Hintergrund durch Netbeans eine XML-Konfigurationsdatei generiert und innerhalb der Projektstruktur abgelegt.

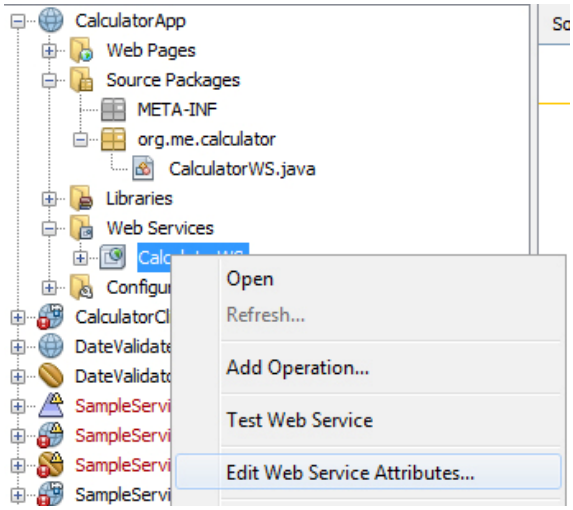

Abbildung 26 - Konfiguration der RM-Eigenschaft

Zunächst ist über die Checkbox „Reliable Message Delivery“ die RM-Funktionalität zu aktivieren:

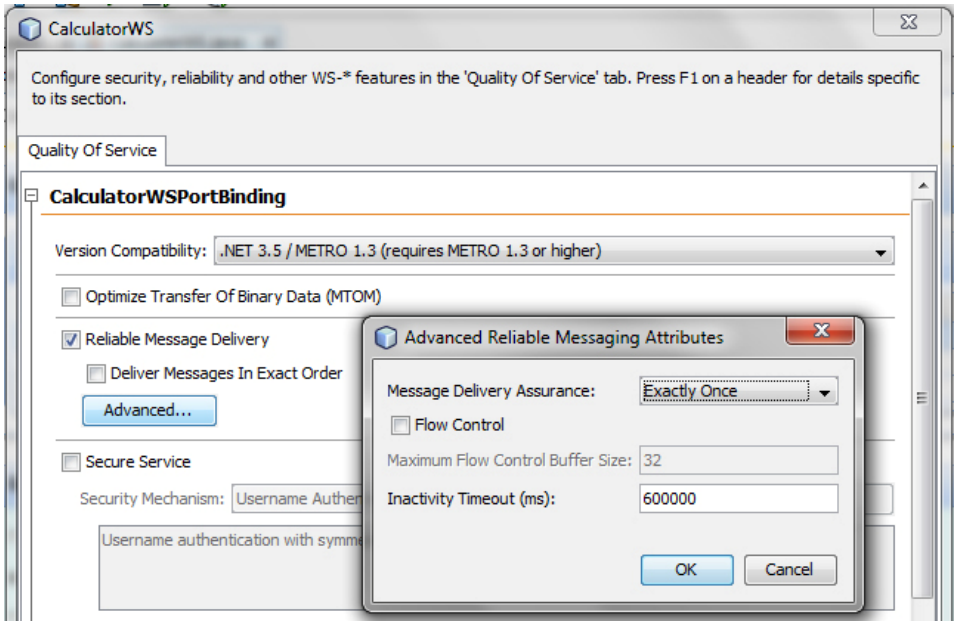

Abbildung 27 – Konfigurationsfenster für Reliable Messaging-Funktionalität

Bei der hier durchgeführten Webservice-Entwicklung soll nicht die EOIO-Zustellgarantie, sondern EO gewählt werden, weshalb die zweite Checkbox „Deliver Messages in Exact Order“ deselektiert bleibt. Neben EO besteht in den erweiterten RM-Einstellungen die Möglichkeit, die verglichen zu EO niedrigwertigere Zustellgarantie „At Least Once“ zu wählen. Dort können im Gegensatz zu EO Nachrichtenduplikate beim Webservice-Endpunkt ankommen. Dies ist deshalb möglich, weil bei „At Least Once“ das RM-Modul auf Empfängerseite keine Filterung von Nachrichtenduplikaten vornimmt, bevor es die Nachrichten an die Webservice-Anwendung hochreicht.

Zusätzlich können in den erweiterten RM-Einstellungen auch durch Feinjustierung die technischen Parameter des RM-Moduls „Maximum Flow Control Buffer Size“ und „Inactivity Timeout“ angepasst werden. Die „Flow-Control“-Checkbox ist in Kombination mit dem „Maximum Flow Control Buffer Size“-Feld zu betrachten: Im Falle der Selektierung der „Flow Control“-Checkbox ist in dem benachbarten Feld bereits der vorgeschlagene Standardwert 32 eingetragen. Dieser Wert kennzeichnet die maximale Anzahl an Nachrichten einer Sequenz, die das empfangende RM-Modul innerhalb des Wartezustandes bis zur Übertragungsvollendung maximal zwischenspeichert. Wenn die Anzahl an gepufferten Nachrichten den festgesetzten Schwellenwert erreicht, werden die restlich ankommenden Nachrichten der Sequenz ignoriert. Zusammenfassend wird also mit der Aktivierung der Flusskontrolle der im Feld „Maximum Flow Control Buffer Size“ festgesetzte Wert von dem Empfänger-RM-Modul verwendet. [11] „Inactivity Timeout“ legt den Zeitraum in Millisekunden fest, nachdem die RM-Session einer Nachrichtensequenz aufgrund von Inaktivität terminiert wird. Verringert man den ursprünglich vorgeschlagenen Wert von 600000 Millisekunden (= 10 Minuten), dann wird der Speicher einer RM-Sequenz wieder frühestmöglich freigegeben. [12]

Nach Bestätigung der getätigten Konfigurationen kann in der Ordner-Ansicht von Netbeans die automatische Erzeugung der WSIT-Konfigurationsdatei nachvollzogen werden.

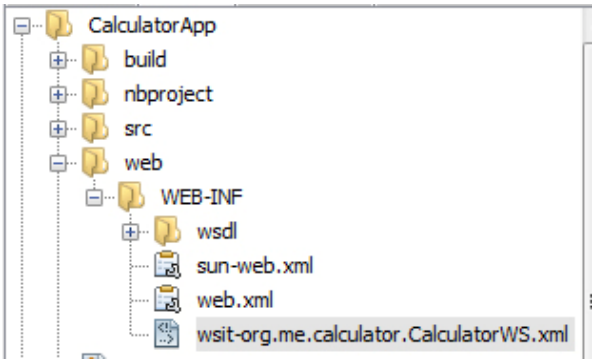

Abbildung 28 – Ablageort der WSIT-Konfigurationsdatei

```
<wsp:Policy wsu:Id="CalculatorWSPortBindingPolicy">
    <wsp:ExactlyOne>
        <wsp:All>
            <wsam:Addressing wsp:Optional="false"/>
            <wsrm:RMAssertion>
                <wsp:Policy></wsp:Policy>
            </wsrm:RMAssertion>
        </wsp:All>
    </wsp:ExactlyOne>
</wsp:Policy>
```

Abbildung 29 - Auszug der WSIT-Konfigurationsdatei

Wird im Folgeschritt die Bereitstellung auf den Glassfish Applikationsserver getätigt, so bindet dieser aufgrund der im Projekt vorzufindenden WSIT-Konfigurationsdatei das benötigte Metro-RM-Modul in die Webservice-Laufzeitumgebung ein. Beim Aufruf der publizierten WSDL via Browser ist sehen, dass die generierte WSDL-Datei um das Policy-Element im Kopfteil erweitert wurde:

```
<!--
    Published by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is M
-->
<!--
    Generated by JAX-WS RI at http://jax-ws.dev.java.net. RI's version is M
-->
<definitions targetNamespace="http://calculator.me.org/" name="CalculatorWSService">
  <wsp:Policy wsu:Id="CalculatorWSPortBindingPolicy">
    <wsrmp:RMAssertion>
      <wsp:Policy/>
    </wsrmp:RMAssertion>
    <wsam:Addressing/>
  </wsp:Policy>
  <types>
```

Abbildung 30 – Die in der WSDL-Datei eingebetteten WS-RM- und WS-Addressing-Richtlinien

Das Element „<wsp:Policy>“ spiegelt den WS-Policy-Standard wieder, welcher als erweiterter WSDL-Standard von W3C dafür spezifiziert wurde, um weitere Diensteigenschaften oder Richtlinien für einen Webservice zu beschreiben. Der WS-Policy-Standard ist ein Rahmenwerk, auf dem die einzelnen speziellen WS-*-Standards wie beispielsweise WS-RM aufbauen. [2, pp. 225,227]
Die in diesem Fall verwendeten Richtlinien für den Calculator-Webservice sind wie auf der Abbildung 30 ersichtlich, WS-RM (<wsrmp:RMAssertion>) und WS-Addressing (<wsam:Addressing/>). Dem Webservice-Client wird durch die Angabe der beiden Policies innerhalb der WSDL-Datei mitgeteilt, dass er die beiden Standards unterstützen muss, um den höherwertigen Reliable Messaging-Webservice ansprechen zu können. Neben WS-RM ist auch eine Unterstützung von WS-Addressing für den Client vorgeschrieben, weil WS-RM hinsichtlich des Standards (und somit auch technologisch) auf WS-Addressing aufsetzt. Über WS-Adressing kann ausgesagt werden, dass mit den ausgetauschten Nachrichten auch Routinginformationen übertragen werden.[5]

Nach der erfolgreichen Bereitstellung des Webservice auf den Glassfish Applikationsserver, wurde mit dem Open-Source Tool SOAPUI die „Reliable Messaging“-Fähigkeit des Webservice-Endpunktes getestet. Hierzu kann - nachdem die WSDL bei der Erstellung eines neuen SOAPUI-Projektes importiert wurde - in den erweiterten Einstellungen auch auf Clientseite die RM-Unterstützung aktiviert werden (Siehe Abbildung 31).

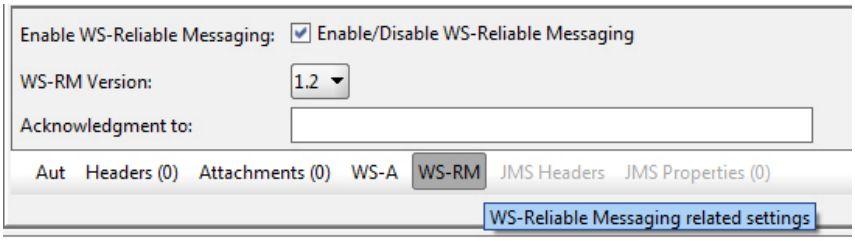

Abbildung 31 – Aktivierung der RM-Unterstützung für SOAPUI als simulierender Webservice-Client

[5] SOAP-Nachrichten, die in einem RM-Webservice-Szenario verschickt werden, beinhalten im SOAP-Header zusätzliche WS-RM- und WS-Addressing-Informationen (siehe Abbildung 32). Der Client muss ebenfalls diese Standards unterstützen, damit er diese erweiterten Informationen interpretieren und auch technologisch die notwendigen Aufgaben eines RM-Senders übernehmen kann (Übertragungswiederholung und Nachrichtenpufferung).

Wird in einem Folgeschritt eine SOAP-Anfrage-Nachricht an den Webservice gesendet, so erhält SOAPUI als Antwort eine Nachricht in der auf Abbildung 32 dargestellten Form. Der SOAP-Header beinhaltet alle relevanten Informationen. Besonders eingegangen werden soll auf das <AcknowledgementRange>-Tag, mit dem die kumulative Quittierung an den Client erfolgt. In diesem Fall wird dem Client übermittelt, dass auf Serverseite bisher die Nachrichten 1 bis 1 angekommen sind. Da SOAPUI standardmäßig pro Request eine Sequenz initiiert und nach dem Versand wieder schließt, besteht jede Sequenz nur aus einer SOAP-Nachricht. Angenommen in einer Sequenz wären mehrere zugehörige SOAP-Nachrichten, so wird ein bestätigter Nachrichtenbereich von beispielsweise 1 bis 3 an den Client verschickt.

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
   <S:Header>
      <Action S:mustUnderstand="1" xmlns="http://www.w3.org/2005/08/addressing">http://calculator.me.org/CalculatorWS/
      <MessageID xmlns="http://www.w3.org/2005/08/addressing">uuid:d9c8bd92-730a-4823-a33a-cb8186fa388f</MessageID>
      <RelatesTo xmlns="http://www.w3.org/2005/08/addressing">uuid:7903c9d0-902b-4a92-9d82-53dce3f68fd6</RelatesTo>
      <To xmlns="http://www.w3.org/2005/08/addressing">http://www.w3.org/2005/08/addressing/anonymous</To>
      <ns2:Sequence ns8:mustUnderstand="true" xmlns="http://www.w3.org/2005/08/addressing" xmlns:ns2="http://docs.oasi
         <ns2:Identifier>urn:soapui:e140fc10-fc71-4218-a40b-0994168lfla3</ns2:Identifier>
         <ns2:MessageNumber>1</ns2:MessageNumber>
      </ns2:Sequence>
      <ns2:AckRequested xmlns="http://www.w3.org/2005/08/addressing" xmlns:ns2="http://docs.oasis-open.org/ws-rx/wsrm/
         <ns2:Identifier>urn:soapui:e140fc10-fc71-4218-a40b-0994168lfla3</ns2:Identifier>
      </ns2:AckRequested>
      <ns2:SequenceAcknowledgement xmlns="http://www.w3.org/2005/08/addressing" xmlns:ns2="http://docs.oasis-open.org/
         <ns2:Identifier>uuid:e06837ad-6ed2-490c-9012-f04c37e2145d</ns2:Identifier>
         <ns2:AcknowledgementRange Upper="1" Lower="1"/>
      </ns2:SequenceAcknowledgement>
   </S:Header>
   <S:Body>
      <ns2:addResponse xmlns:ns2="http://calculator.me.org/">
         <return>8</return>
      </ns2:addResponse>
   </S:Body>
</S:Envelope>
```

Abbildung 32 – SOAP-Antwort-Nachricht mit zusätzlichen Informationen hinsichtlich WS-RM und WS-Addressing

Im letzten Teil der RM-Entwicklung soll veranschaulicht werden, welche programmatischen Ergänzungen in einer Java-basierten Clientimplementierung zu tätigen sind, damit der Webservice-Client an einer ReliableMessaging-Konversation teilnehmen kann. Nachdem in Netbeans unter der Kategorie „Java Web“ ein neues „Web Application“-Projekt angelegt wurde, müssen zunächst (wie auch in den Kapiteln 5.4 und 5.5 bei Eclipse) die Dienstnutzer-Stub-Klassen aus der WSDL-Datei generiert werden. Im Gegensatz zu Eclipse verwendet Netbeans beim „New Web Service Client“-Wizard JAX-WS anstatt Axis.

Die Implementierung des Clients kann entsprechend Kapitel 5.5 aufgesetzt werden. Der einzige Unterschied ist die Ergänzung um eine close-Anweisung, nachdem der letzte Webservice-Methodenaufruf stattfindet. Durch die close-Anweisung wird clientseitig die Nachrichtensequenz beendet. Hierzu ist der Webservice-Port nach dem letzten Zugriff durch die auf dem Listing 9 ersichtliche Anweisung zu schließen[6]. Nach dieser Ergänzung ist der Webservice-Client in der Lage, an einer RM-Kommunikation mit einem „Reliable Message„-fähigen Webservice teilzunehmen.

[6] Folgende Import-Anweisung wird für die Verwendung des Closeable-Interfaces benötigt:
import com.sun.xml.ws.Closeable;

```
org.me.calculator.client.CalculatorWS port = service.getCalculatorWSPort();

int result1 = port.add(1, 2);
int result2 = port.add(3, 4);
int result3 = port.add(5, 6);
int result4 = port.add(7, 6);

((Closeable)port).close();
```

Listing 9 - Sequenz clientseitig schließen mittels close-Anweisung (übernommen aus [8, pp. 43-44])

7 Fazit

Ein Kerngedanke der Java-Webservice-Technologie (JAX-WS in Verbindung mit JAXB), nämlich dem Entwickler die Komplexität der Webservice-Technologie zu verbergen, ist die von den Verantwortlichen des Java Community Prozesses richtige Zielsetzung. So ist es auf jeden Fall für den Entwickler von Java-basierten Webservices und auch von WS-Clients ein großer Vorteil, da er sich in erster Linie auf die Anwendungslogik fokussieren kann und sich nicht mit der Nachrichtenübertragung auf SOAP-Schicht auseinandersetzen muss. Betrachtet man nochmals exemplarisch den Code-First-Entwicklungsprozess, so übernimmt der Dienstentwickler nur einen kleinen Part an der gesamten Webservice-Entwicklung. Er braucht nur seine Java-Anwendungslogik an bestimmten Stellen mit Webservice-spezifischen Annotationen ausstatten. Die restlichen Entwicklungsschritte auf Dienstseite werden automatisch durch die Bereitstellung auf einen JEE6 implementierenden Applikationsserver wie Glassfish erledigt.

Der von der Java-Webservice-Technologie umgesetzte starke Kapselungsgrad einiger interner Mechanismen vor dem Anwendungsentwickler ist eine zweischneidige Angelegenheit. Bei kleinen Webserviceentwicklungen - ohne die Verwendung der höheren WSIT-Technologien mit den gängigen IDE's - funktioniert der Code-First-Ansatz prinzipiell sehr gut. Vorausgesetzt die Java-Dienstimplementierung ist syntaktisch korrekt und dazu sind auch Java-Dienstbeschreibung und Java-Datentypen richtig mit den dementsprechenden Annotationen versehen, so überführt der Glassfish-Applikationsserver diese mit vollster Zuverlässigkeit nach WSDL und XSD. Jedoch ergab sich aus eigenen Beobachtungen ein Kritikpunkt: In manchen Entwicklungssituationen funktionierte der Deployment-Vorgang nicht, obwohl die dementsprechende IDE keine syntaktischen Fehler monierte. Zudem hatten die in den Server-Logs protokollierten Fehlermeldungen oftmals gar keinen Bezug zu dem eigentlichen Fehlergrund. Dies alles erschwerte die Einarbeitung enorm und nahm sehr viel Zeit in Anspruch. Oftmals war nicht nachzuvollziehen, ob die Fehlerursache an einer falsch angewandten Annotation oder an einem Fehler in der Anwendungslogik lag. Es wäre an dieser Stelle wünschenswert, dass das JAX-WS- bzw. JAXB-spezifische Debugging stärker in der jeweiligen IDE verankert wäre, und der Entwickler bereits vor der Bereitstellung auf den Applikationsserver bei falscher Programmierung besser auf die richtigen Lösungsansätze aufmerksam gemacht werden würde.

Wird Augenmerk auf die WSIT-Technologien gelegt, kann definitiv bestätigt werden, dass der gesamte Funktionsumfang in Metro und somit im aktuellen Glassfish V 3.1.2 Applikationsserver enthalten ist. Trotzdem konnte anhand einer tiefergehenden Auseinandersetzung mit WS-AtomicTransaction und WS-ReliableMessaging folgendes festgestellt werden: Insgesamt wurde der

Support der WSIT-Themen bei der aktuellen Version von Metro, d.h. der zentralen Open-Source-Lösung für WSIT, in den letzten Jahren stark vernachlässigt. Dieser Eindruck wurde bestätigt bei intensiven Online-Recherchen anhand des aufgefundenen Dokumentationsmaterials. In erster Linie waren auf den offiziellen Homepages des Metro-Projektes, von Oracle und auf diversen Foren Artikel aus den Jahren 2007-2009 vorzufinden, die sogar Links beinhalteten, hinter denen keine Referenz mehr hinterlegt war. Diese Dokumentationen und auch Tutorials waren vorrangig abgestimmt auf ältere Versionen von Netbeans und Glassfish. Im Gegensatz dazu wurden auf Homepages anderer kommerzieller Applikationsserver-Anbieter wie IBM und SAP aktuellere Dokumentationen und Tutorials zu WS-AT und WS-RM gefunden. Diese beiden Technologien sind jedoch sehr stark innerhalb der Websphere- bzw. NetWeaver-Architektur verbaut, und somit war es nicht möglich, beschriebene Webservice-Szenarien nachzubilden. Viele der vorgefundenen Tutorials sind bereits in der Auslieferung des jeweiligen Applikationsservers in Form von ausführbaren „Samples“ enthalten. Die eigene Vermutung ist in diesem Zusammenhang, dass seit der Firmenübernahme von Sun durch Oracle im Jahr 2009 die Weiterentwicklung der WSIT-Themen in dem Open-Source-Projekt Metro strategisch gestoppt wurde und der Support inklusive der Dokumentation von dem Open Source Bereich abgezogen und verstärkt in den kommerziellen Bereich verlagert wurde. Die anderen Unternehmen wie SAP und IBM realisieren vereinzelt ausgesuchte WS-*-Standards in deren Applikationsservern unterschiedlich auf deren eigene Weise. So unterstützt beispielsweise IBM die WS-AtomicTransation-Technologie sehr durchgängig und SAP hingegen überhaupt nicht. Dabei setzt SAP voll auf die WS-ReliableMessaging-Technologie. Insgesamt wird durch die beschriebene historische Entwicklung der Zugriff auf aktuelle Dokumentationen nur Käufern der kommerziellen Applikationsserver zur Verfügung gestellt. Es bleibt zukünftig abzuwarten, ob die WSIT-Referenzimplementierung entweder im Metro-Projekt oder auch in einem anderen neuen Open-Source-Projekt wieder weiter angetrieben wird.

8 Literaturverzeichnis

[1] A. Goncalves, Beginning Java EE 6 Platform with GlassFish 3: From Novice to Professional, APress, 2009.

[2] O. Heuser und A. Holubek, Java Web Services in der Praxis, DPunkt.Verlag, 2010.

[3] „JAX-WS 2.2 Specification Chapter 1 Introduction," 10 12 2009. [Online]. Available: http://download.oracle.com/otn-pub/jcp/jaxws-2.2-mrel3-evalu-oth-JSpec/jaxws-2_2-mrel3-spec.pdf?AuthParam=1370075617_7ef887712d5cfc9504b690baeba4e869. [Zugriff am 01 06 2013].

[4] „IBM Websphere Infocenter Suchbegriff: SEI-basierte JAX-WS-Web-Services entwickeln," 10 01 2011. [Online]. Available: http://pic.dhe.ibm.com/infocenter/wasinfo/v7r0/index.jsp?topic=%2Fcom.ibm.websphere.express.doc%2Finfo%2Fexp%2Fae%2Ftwbs_devjaxwsendpt.html. [Zugriff am 28 05 2013].

[5] „Glassfish Metro - Project JAXB," [Online]. Available: https://jaxb.java.net/tutorial/section_6_2_5-Controlling-Element-Selection-XmlAccessorType-XmlTransient.html. [Zugriff am 31 05 2013].

[6] „Metro - Introduction," [Online]. Available: https://metro.java.net/guide/ch01.html#ahiak. [Zugriff am 30 05 2013].

[7] P. Mandl, A. Bakomenko und J. Weiß, Grundkurs Datenkommunikation - TCP/IP-basierte Kommunikation: Grundlagen, Konzepte und Standards, Vieweg + Teubner, 2008.

[8] „Oracle - WSIT Tutorial," 18 09 2007. [Online]. Available: http://docs.oracle.com/cd/E17802_01/webservices/webservices/reference/tutorials/wsit/doc/WSITTutorial.pdf. [Zugriff am 29 05 2013].

[9] P. Mandl, Vorlesungsskript Datenkommunikation: Transportschicht Grundlagen.

[10] „OASIS - PDF S.14," 02 02 2009. [Online]. Available: http://docs.oasis-open.org/ws-rx/wsrm/200702/wsrm-1.2-spec-os.pdf. [Zugriff am 28 05 2013].

[11] „Metro Projekt," [Online]. Available: https://metro.java.net/guide/ch10.html. [Zugriff am 29 05 2013].

[12] „Metro Projekt - Chapter 10. Using Reliable Messaging," [Online]. Available: https://metro.java.net/guide/ch10.html. [Zugriff am 29 05 2013].

[13] „Oracle - 5 Using Web Services Reliable Messaging," 2007. [Online]. Available: http://docs.oracle.com/cd/E21764_01/web.1111/e13734/rm.htm. [Zugriff am 28 05 2013].

[14] „Wikipedia - Java Community Process," [Online]. Available: http://de.wikipedia.org/wiki/Java_Community_Process. [Zugriff am 01 06 2013].

9 Abbildungsverzeichnis