INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Realisierung und Management zustandsbehafteter Web Service Resources mit Apache Muse

Michael Höber

Draft vom 22. September 2008

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Fortgeschrittenenpraktikum

Realisierung und Management zustandsbehafteter Web Service Resources mit Apache Muse

Michael Höber

Aufgabensteller:	Prof. Dr. Heinz-Gerd Hegering
Betreuer:	Dr. Michael Schiffers
Abgabetermin:	7. Juli 2077

Hiermit versichere ich, dass ich die vorliegende FoPra-Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 7. Juli 2077

. . . . . .
(Unterschrift des Kandidaten)

Abstract

Web Services werden immer beliebter und nach und nach mehr eingesetzt. So wurde ein Standard zum Management mittels Web Services von Oasis unter dem Namen WSDM - Web Services Distributed Management - spezifiziert. Diese Management-Spezifikation umfasst viele Teil-Spezifikationen, von denen eine gewissen Anzahl durch das Apache Muse Project in Java implementiert ist.

Die Arbeit umfasst die Analyse von Apache Muse sowie eine konkrete Implementierung und Anwendung eines Management-Szenarios.

Inhaltsverzeichnis

1 Einleitung

Web Services nehmen stetig an Verbreitung und möglichem Funktionsumfang zu. Von OASIS ([OAS08a]) wurde daher eine Spezifkation entwickelt, in der beschrieben wird, wie sich diese Web Services auch zum Management eignen. Diese Spezifikation wird als WSDM, Web Services Distributed Management, bezeichnet und in einzelne Teil-Spezifikationen unterteilt.

1.1 Aufgabenstellung

In dieser FoPra-Arbeit wird mit Hilfe von Apache Muse eine kleine Management-Umgebung auf Basis von Web Services entwickelt. Dazu werden zuerst einige Definitionen in Kapitel 2 erkläutert sowie auf grundlegende Konzepte eingegangen. Kapitel 3 befasst sich mit der Installation und Konfiguration der notwendigen Software. In Kapitel 4 wird auf die Funktionsweise von Apache Muse in Zusammenhang mit Apache Axis2 und Apache Tomcat eingegangen. Kapitel 5 dient uns zur Entwicklung folgendem konkretem Beispiels:
Ein Apache HTTP Web Server soll als Komponente angesehen werden und mittels den Web Services Distributed Management (WSDM) Möglichkeiten gemanaget werden. So soll der Server gestartet und beendet werden können. Außerdem soll natürlich vor Start der Port und der Servername konfiguriert werden können, auf dem der HTTP-Server lauscht. Vor dem Betrieb als auch während des Betriebes kann der aktuelle Servername, -Port und das Installationsverzeichnis abgefragt werden. Des weiteren kann der verwaltende Web Service auch mittels einem Subscribe-Notification Mechanismus die momentane Anzahl der Verbindungen des Web Servers, sowie im Falle eines Ausfalls, auch den Zustand des Web Servers an alle Subscriber übermitteln. Das Intervall in dem die Nachrichten gesendet werden, kann ebenfalls jederzeit über den Web Service angepasst werden.
Im Anschluss wird in Kapitel 6 auf Probleme von Apache Muse eingangen, sowie die Erfahrungen damit näher erläutert.
Diese Arbeit legt somit den Grundstein um Management mittels Web Services praktisch umzusetzen. Sogenannte Virtuelle Organisationen (VOs) in Grid-Umgebungen können als „managed objects“ betrachtet werden, so dass diese auch mit den Methoden des Web Services Distributed Managements verwaltet werden können.

2 Grundlegende Konzepte

Dieses Kapitel behandelt die grundlegenden Konzepte sowie Begriffsdefinitionen.

2.1 Definitionen

In dieser Arbeit bietet sich für unser Ziel konkret Apache Muse an. Doch bevor wir näher auf Muse eingehen, müssen einige notwendige Begriffe näher betrachtet werden.

2.1.1 Web Service

Abbildung 2.1: Web Service

Web Services wurden von der W3C definiert als Anwendungen, die mit Hilfe einer URI (Uniform Resource Identifier) identifiziert werden. [Wik08f] erklärt, dass die Schnittstellen von Web Services anhand von XML Ausdrücken definiert, beschrieben und im Netzwerk zur Verfügung gestellt werden.
Google bietet einen Web Service an, der dieselben Funktionen implementiert wie die Oberfläche, die ein menschlicher Nutzer im Browser sieht. Anwendungen brauchen jedoch keine grafische Darstellung der Suchmaschine, sondern nur deren Funktionalität. Dies bietet Google mit Hilfe eines Web Services an, somit ist ein Parsen der Google-Webseite nicht nötig, was eine Performanceverbesserung bietet und Traffic-Einsparungen ermöglicht.
Web Services unterstützen direkte Interaktionen mit anderen Anwendungen durch den Austausch von XML-basierten Nachrichten. Typischerweise wird das SOAP-Protokoll über HTTP benutzt, um Requests an einen Service an einer bestimmten URL zu senden, auf welcher der Web Service angesprochen wird und die Nachricht nach einer Bearbeitung als Response zurücksendet.
Web Services im Allgemeinen stellen Services zur Verfügung, die mit verschiedene Arten von Netzkomponenten wie zum Beispiel Clients, Servers, Anwendungen, Benutzer, Programme usw. interagieren. Daher ist es nötig, dass Web Services gut integriert sind, sowie auch gut skalierbar und einfach zu pflegen sind. Ebenso soll auch die Möglichkeit bestehen, dass Web Services selbst auch gemanaged werden können, damit sichergestellt werden kann, dass der von ihnen angebotene Service nicht abstürzt oder überlastet wird, damit die Performance in akzeptablem Bereich bleibt. Um diese Ziele zu erreichen, bietet das Web Management Lösungen für Web Services an, damit diese die Fähigkeit haben, Services-basierte Applications zu überwachen und kontrollieren.

2.1.2 SOAP

[Wik08e] beschreibt das Simple Object Access Protocol (SOAP) als ein Protokoll, das dazu dient, Daten zwischen Systemen auszutauschen und Remote Procedure Calls auszuführen. SOAP verwendet zur Übertragung die standardisierten Internet-Protokolle der Anwendungs- und Transportschicht. Zur Repräsentation wird XML verwendet.
SOAP wird üblicherweise als Header-Body-Pattern modelliert, welches versehen mit dem verwendeten Namensraum auch als SOAP-Envelope bezeichnet wird. Wie üblich sind im Header Metainformationen enthalten und im Body die Nutzdaten.
SOAP basiert auf dem Request-Reponse-Prinzip, das heißt, wenn einen Anfrage an zum Beispiel einen Web Service geschickt wird, so wird von diesem eine Antwort erwartet.

2.1.3 WSDL

Die Web Services Descripten Language (WSDL) definiert laut [Wik08g] eine plattform-, programmiersprachen- und protokollunabhängige XML-Spezifikation, um Web Services zu beschreiben und die Nachrichten, mit denen diese angesprochen werden können, zu definieren.
Mit Hilfe von WSDL werden die angebotenen Funktionen, Daten, Datentypen und Austauschprotokolle von Web Services definiert und beschrieben. Grundsätzlich werden die von außen zugänglichen Operationen und deren Parameter sowie Rückgabewerte erklärt. Ein WSDL Dokument beinhaltet folgende Angaben:

- Schnittstelle
- Zugansprotokoll und Details zum Deployment
- alle benötigten Informationen in maschinenlesbarem Format, um auf den Service zuzugreifen

Ein Web Service wird durch folgende sechs XML-Hauptelemente definiert:

- Datentypen (types)
  Definition der Datentypen, die zum Austausch der messages benutzt werden
- Nachrichten (messages)
  Abstrakte Definitionen der übertragenen Daten, bestehend aus mehreren logischen Teilen, von denen jedes mit einer Definition innerhalb eines Datentypes verknüpft ist.
- Port-Typen (portType)
  Eine Menge von abstrakten Arbeitsschritten (vier Typen von ausgetauschten Nachrichten):
  - One Way: Der Service bekommt eine Input-Message vom Client
  - Request-response: Der Service bekommt einen Request (Input-Messsage) vom Client und sendet daraufhin eine Response (Output-Message)
  - Solicit-response: Der Service sendet eine Message und erwartet eine Bestätigung vom Client.
  - Notification: Der Service sendet eine Output-Message.

- Bindung (binding)
  Bestimmt das konkrete Protokoll und Datenformat für die Arbeitsschritte und Nachrichten, die durch einen bestimmten Port-Typ gegeben sind.
- Ports (port)
  Spezifiziert eine Adresse für eine Bindung, also eine Kommunikationsschnittstelle, standardmäßig eine URI.
- Services
  Definieren, wie der Service erreichbar ist.

Anwendung findet WSDL häufig in Kombination mit SOAP und XML-Schema, um Web Services im Internet und lokalem Netzwerk anzubieten. Der Client, der einen Web Service benutzen möchte, muss WSDL lesen können, um die bereitgestellten Funktionen interpretieren zu können. Um eine in WSDL gelistete Funktion aufzurufen kann der Client SOAP verwenden.
Letztendlich dient WSDL der Spezifizierung der syntaktischen Elemente eines Web Services, also wie ein Client auf den entsprechenden Web Service zugreifen kann. Sollen darüber hinaus weitere semantische Spezifikationen gegeben werden, wie zum Beispiel Antwortzeit, Kosten eines Services, Security-Policies sowie auch zur Discovery (automatischen Auffindung von Services) nötige Parameter, muss auf eine Erweiterung von WSDL wie zum Beispiel WSDL-S, WSLA oder OWL-S zurückgegriffen werden. Diese genannten Beispiele sind weitaus mächtiger bei der Beschreibung, aber auch weit komplexer.

2.1.4 WS-Distributed Management (WSDM)

Oasis Web Service Distributed Management ist nach [OAS08a] ein Projekt des WSDM Technical Commitee, welchem Actional corporation, BEA Systems, BMC Software, Computer Associates, Dell, Fujitsu, HP, Hitachi, IBM, Novell, Oracle und TIBCO angehören. Ziel von WSDM ist ebenso wie auch bei WS-Management die Verwirklichung eines Web Service basierten Managements laut [VB06]. Im Gegensatz zu WSM bietet WSDM aber auch noch die Möglichkeit die Web Services selbst zu managen. WSDM beschreibt die Management Informationen und capabilties für bestimmte Geräte, Anwendungen und Komponenten mit Hilfe der Web Service Description Language (WSDL).
WSDM wurde als Standard von der Organization for the Advancement of Structured Information Standards (OASIS) genehmigt, was als weiterer Schritt für die Integration und Verbreitung von WSDM angesehen werden kann.
Das WSDM Technical Commitee hat zwei getrennte Spezifikationen definiert. Zum einen die Möglichkeit mit Hilfe von Web Services Management Funktionen durchzuführen, zum Anderen die Möglichkeit, die Web Services selbst zu managen. WSDM bietet Management Tools für vernetzte Systeme und Devices und bietet Mechanismen, die in ein bereits existierendes Netzwerk und eine Service Oriented Architecture (SOA) passen.
Der WSDM-Standard besteht aus zwei unterschiedlichen Standards.

- **Management using Web Services (MUWS)**
  Management mit Hilfe von Web-Services zum Managen von zum Beispiel Druckern, Routern und sonstigen Komponenten und Geräten.
- **Management of Web Services (MOWS)**
  Management von Web-Services, welche der Funktionalität des Netzwerkes dienen.

2.1.5 WS-Addressing

[Win04] beschreibt WS-Addressing als eine Spezifikation, die es Web Services ermöglicht, Adressinformationen auszutauschen. Diese besteht aus zwei Teilen: Zum einen eine Datenstruktur zur Übergabe einer Referenz zu einem Web Service EPR, zum anderen eine Menge von Adresseigenschaften von Nachrichten, die die Adressinformationen mit einer bestimmten Nachricht verknüpfen.
Da jede SOAP-Nachricht in ihrem Header zusätzliche Metainformationen über den Absender und den Empfänger der Antwort sowie Empfänger von etwaigen Fehlernachrichten enthält, erleichter WS-Addressing die Verwendung asynchroner Web Service Aufrufe. Damit ist es nun möglich, auf eine SOAP-Nachricht zu antworten, auch wenn die eigentliche HTTP-Nachricht bereits nicht mehr existiert, da eine eindeutige ID, sowie Referenzen zu vorangegangenen Nachrichten mittransportiert werden.

2.1.6 WS-Resource Framework (WSRF)

Das WS-Resource Framework (WSRF) umfasst nach [uI04] eine Sammlung modularer Einzelspezifikationen, die für den Zugriff mittels eines Web Services auf eine Resource (entweder eine Komponente oder selbst ein Web Service) ermöglichen. Um einen bestimmten Web Service ansprechen zu können, muss dieser mittels einer Endpoint Reference (EPR) eindeutig gekennzeichnet sein und an diesen gerichtete Anfragen laut der WS-Addressing-Spezifikation ([Win04]) annehmen. Das WSRF beinhaltet desweiteren folgende Spezifikationen.

- WS-ResourceProperties
  Die in [SG04b] beschriebenen WS-ResourceProperties (WS-RP) spezifizieren, wie Daten, die mit zustandsbehafteten Resourcen verbunden sind, über Web Services abgefragt und manipuliert werden können mit Hilfe von Operationen wie GetResourceProperty, UpdateResourceProperty und DeleteResourceProperty.

- WS-ResourceLifetime
  Laut [Wee04] dient die WS-ResourceLifetime-Teilspezifikation (WS-RL) zur zeitgesteuerten und sofortigen Auflösung von WS-Resourcen. Mit Hilfe der SetTerminationTime Methode kann ein beliebig in der Zukunft liegender Zeitpunkt zur Auflösung der WS-Resource definiert werden. Um etwaige Zeitdifferenzen abzugleichen kann mittels CurrentTime die aktuelle Server-Zeit abgerufen werden. Eine sofortige Deaktivierung und Löschung einer Resource bewirkt die destroy Methode.

- WS-ServiceGroup
  In [TM04] wird die WS-ServiceGroup-Spezifikation (WS-SG) beschrieben. Dabei werden Web Services zu ServiceGroups gruppiert. So wird erklärt, dass die Gruppierungskriterien mittels MembershipcontentRules festgelegt werden und durch Ansprechen der ServiceGroupRegistration-Schnittstelle verwaltet werden. Ein Web Service kann hierbei mehreren ServiceGroups angehören, welche jedoch auch in ihren Aufnahmekriterien eingeschränkt sein können.

- WS-BaseFaults
  Die WS-BaseFaults-Spezifikation (WS-BF) beschreibt nach [ST04] einen Mechanismus, um etwaige Fehlermeldungen zu generieren. Diese Meldungen genügen einem

XML-Schema, das eine Fehlerbeschreibung, einen Zeitstempel, einen Fehlercode und die Quelle enthält.

2.1.7 WS-Notification (WSN)

WS-Notification dient zur Übermittlung von Nachrichten außerhalb des request-response Prinzips. In WS-Notification wird laut [uI04] definiert, wie Nachrichten an einen Subscriber gesendet werden können.
Dabei kommen folgende Begriffe zum Einsatz:

- NotificationMessage
  Eine Nachricht, die zum Empfänger übermittelt werden soll, um diesen über ein bestimmtes Ereignis zu informieren.

- NotificationProducer
  Der Producer genieriert Messages beim Eintreten eines zu überwachenden Vorfalls.

- NotificationConsumer
  Der Consumer empfängt die Nachrichten eines Producers.

- Subscription
  Eine Registierung eines Consumers bei einem Producer, sowie die Beziehung untereinander und eventuell vereinbarte Topics.

- Subscriber
  Ein Subscriber führt Subscriptions von einem Consumer bei einem Producer durch.

- NotificationBroker
  Ein NotificationBroker dient als Zwischeninstanz zwischen Consumer und Producer. So kann es zum Beispiel nötig sein, bei einem bestimmten Vorfall verschiedene Consumer zu benachrichtigen, ohne dass der Producer mehr als eine Nachricht versendet. In diesem Fall sendet der Producer nur eine Nachricht an einen bei ihm registrierten Broker und dieser kümmert sich dann um den Versand aller Nachrichten an die Consumer. Somit ist ein Broker sowohl Producer als auch Consumer, je nach Sicht.

- Publisher
  Ein Publisher ist eine Instanz, die NotificationMessages generiert. Ein Publisher kann entweder die Situationen-entdeckende Instanz sein, die einen Broker informiert oder aber ein NotificationProducer selbst sein, der zusätzlich eine Liste der aktiven Subscriptions verwaltet und gegebenfalls benachrichtigt.

- PublisherRegistration
  Eine PublisherRegistration repräsentiert die Beziehung zwischen einem Publisher und dem dazugehörigen NotificationBroker.

- WS-BaseNotification
  In [SG04a] ist zu lesen, dass die WS-BaseNotification als Basis für den WS-Notification Standard dient. Prinzipiell muss sich ein Subscriber bei einem NotificationProducer anmelden, so dass dieser an einen bestimmten NotificationConsumer die angeforderten Nachrichten sendet. Laut der WS-BaseNotification Spezifikation ist auch der Einsatz

von Filtern möglich. Dies ist nötig, da grundsätzlich alle eintretenden Events an alle angemeldeten Consumer geschickt werden. Oft ist das unnötig, da nur bestimmte Nachrichten von bestimmten Consumern empfangen werden wollen. Daher ist es mit Hilfe von Filtern möglich diese näher zu spezifizieren.
WS-BaseNotification unterstützt auch automatisches Beenden eine Subscription. Dies ist möglich durch Setzen eines expliziten Datums oder aber auch durch relative Zeit-Intervalle. Sollte das Timeout verlängert werden müssen, so ist dies durch eine renew-Nachricht jederzeit möglich.

- WS-Topics
  Laut [Vam04] dienen WS-Topics dazu, Filter unnötig zu machen. Mit Hilfe von Topics werden Events kategorisiert und in hierarchischen Bäumen gespeichert. Subscriptions beziehen sich hierbei nicht mehr auf NotificationProducer, sondern auf Teilbäume beziehungsweise Pfade des Baumes eines Producers. Somit kann bereits bei der Subscription mit Hilfe von Topics näher definiert werden, über welche Events der Consumer benachrichtigt werden will.

- WS-BrokeredNotification
  Normalerweise sind NotificationProducer und Publisher in einer Instanz vereint. Durch Einsatz eines Brokers, können Producer und Publisher getrennt werden, wie in [DC04] beschrieben wird. Dies hat zur Folge, dass der Producer nun einzig für das Verteilen von Nachrichten, sowie die Aufgaben des Subscribens übernimmt, wohingegen der Publisher sich um das Entdecken von bestimmten Situation und Vorfällen kümmert. Ein Broker kann einem bestimmten Publisher zugewiesen sein, aber auch mehreren Publishern dienen. Ebenso verteilt der Broker dann die Nachrichten an einen oder auch mehrere Consumer.

2.1.8 WS-MetadataExchange

Mittels WS-MetadataExchange ist es laut [Oas08b] möglich, Informationen und insbesondere Metadata eines EPRs abzufragen und zwischen Web Services auszutauschen.

2.2 Erläuterung der Software

Nachdem nun alle Definitionen geklärt sind, wird im folgenden Abschnitt näher auf die verwendete Software eingegangen.

2.2.1 Apache Muse

Abbildung 2.2: Apache Muse

Das *Muse Project* von Apache ([Apa08a]), welches derzeit in Version 2.2.0 vorliegt, ist eine Java-basierte Implementierung der WSDM WS-ResourceFramework und WS-BaseNotification, dient also zum Management in einer Web Services Distributed Management Umgebung. In diesem Framework können Web Service Interfaces für Management-Zwecke entwickelt werden, ohne dass sich der Entwickler mit den Spezifikationen näher auseinander setzen muss. Mit Muse entwickelte Anwendungen können in Apache Axis2 und OSGi Umgebungen eingesetzt werden.
Apache Muse bietet hierzu folgenden Funktionsumfang:

- Implementierung der WSRF 1.2, WSN 1.3, WSDM 1.1 und WS-Mex Port Typen
- Standalone Implementierung des WSDM Event Formats 1.1
- Kompatibilität mit WS-Addressing 1.0 und SOAP 1.2
- WSDL-to-Jave Code Generierungs-Tool
- Generierung von WS-Resource- und Client-seitigem Code anhand von WSDL

Da die Entwicklung von WSDL Dokumenten kompliziert ist, haben die Muse Entwickler versucht, den Entwicklungs-Prozess eines WSDL Dokuments zu vereinfachen. Dafür wird ein WSDL Template angeboten, das bereits eine Vielzahl von Properties und Operationen (wie zum Beispiel WS-ResourceProperties, WS-MetadataExchange, WSN NotificationProducer, WSDM MUWS Identity, Description und OperationalStatus) beinhaltet, die nur noch angepasst, sowie um mögliche weitere Properties und Funktionen ergänzt werden müssen. Dies führt dazu, dass der Zeitaufwand der WSDL-Dokument-Entwicklung deutlich reduziert wird, so dass früher mit der eigentlichen Code-Implementierung angefangen werden kann.
Anhand dieses WSDL-Templates wird mittels wsdl2jave der benötigte Code-Skeleton generiert sowie relevante Dateien angelegt. Muse erzeugt dabei eine fest vorgegebene Dateistruktur. So befinden sich zum Beispiel in „/WEB-INF/classes/router-entries“ die XML-Files der EPRs, in „/WEB-INF/classes/wsdl“ die WSDL und XML Schema Files der Resource und in „/WEB-INF/classes/muse.xml“ der Muse Deployment Descriptor.
Der Deployment Descriptor (muse.xml) wird zum Starten, Laden, Konfigurieren und Supporten der Resource Types benötigt und bei Initialisierung der jeweiligen WS Resource

eingelesen. Darin sind <resource-type/> Elemente enthalten, die zum Beschreiben einer Resource nötig sind, zum Beispiel, welche Capabilities angeboten werden oder aber auch Variablen-Bindungen.
Standardmäßig sind in den Muse Resources folgende Capabilities als Klassen implementiert:

- WSRP ResourceProperties
- WSMEX MetadataExchange
- WSN NotificationProducer
- WSDM MUWS Identity
- WSDM MUWS Description
- WSDM MUWS OperationalStatus
- MyCapability (um weitere spezifische Capabilities zu implementieren)

2.2.2 Apache Axis 2

Abbildung 2.3: Apache Axis2

Apache Axis (Apache eXtensible Interaction System) ist eine SOAP-Implementierung und stellt laut [Wik08b] den Nachfolger des hausinternen Apache SOAP dar. Axis wurde mit der Zielsetzung auf höhere Geschwindigkeit, Flexibilität, Komponentenorientierung, Abstraktion des Transportframeworks sowie Unterstützung von WSDL entwickelt. Apache Axis setzt auf einen SAX-Parser im Gegensatz zu Apache SOAP, der auf einen DOM-Parser aufbaut.
Axis2 bietet uns also eine Umgebung an, die SOAP-Nachrichten interpretieren und generieren kann, so dass wir mit den Web Services, die mittels Muse erarbeitet werden, kommunizieren können.
Axis2 wird dazu meist als Java-Servlet innerhalb eines Servlet Containers betrieben. Dafür bietet sich insbesondere der hausinterne *Apache Tomcat* an, auf den im Folgenden näher eingegangen wird.
Axis2 unterstützt die Standards SOAP 1.1, 1.2 und WSDL 1.1 der W3C. Es wird auch SAAJ 1.1 (SOAP with Attachments API for Java) von Sun Microsystems unterstützt.

2.2.3 Apache Tomcat

Abbildung 2.4: Apache Tomcat

Apache Tomcat stellt laut [Wik08d] eine Umgebung zur Ausführung von Java-Code auf Webservern bereit.
Tomcat ist ein in Java implementierter Servlet-Container, der mittels dem JSP-Compiler Jasper auch JavaServer Pages in Servlets übersetzen und ausführen kann. Damit Tomcat per http angesprochen werden kann, beinhaltet er ebenso einen kompletten HTTP-Server, der jedoch meist nur während der Entwicklung direkt angesprochen wird und in einer „Live-Umgebung“ oft einen Apache HTTP Web Server davor geschalten bekommt.
Tomcat ist ein Gemeinschatfsprojekt von der Apache Software Foundation und Sun Microsystems, das aus Apaches JServ-Servlet-Container und der dazugehörigen Referenz-Implementierung durch Sun entstand. Diese erste Version hatte bereits 3.0 als Versionsnummer. Aktuell seit 08.08.2008 ist Version 6.0.18, die auch in dieser Arbeit verwendet wird. Diese Version implementiert die Spezifikationen Servlet 2.5 und JSP 2.1.
Tomcat besteht eigentlich aus dem Servlet-Container Catalina und dem Connector Coyote, der für die Abarbeitung diverser HTTP-Anfragen zuständig ist. Durch diese Trennung kann Catalina auf einem anderen Host betrieben werden, als der zugehörige Web Server, was mit Hilfe einer implementieren Loadbalancer-Funktion sogar dazu führt, dass einem einzelnen Web Server mehrere Servlet-Container zur Verfügung gestellt werden können.

2.2.4 Apache Ant

Abbildung 2.5: Apache Ant

Apache Ant („Another Neat Tool“, „Noch ein nettes Werkzeug“ bzw. „Ameise: ein kleines Programm, das Großes leistet“) ist nach [Wik08a] eine Java-Anwendung zum automatisierten

Erzeugen von ausführbarem Code aus einem Quelltext.
Ant benötigt zur Ausführung eine XML-Datei, standardmäßig build.xml. Diese Datei enthält ein Project-Tag, welches unterteilt ist in mehrere Targets. Einzelne Targets können gezielt aufgerufen werden und Abhängigkeiten zu anderen Targets haben, die dann entsprechend abgearbeitet werden. Jedes dieser Targets ist dann wiederum unterteilt in ein oder meist mehrere Tasks, also den eigentlichen Aufgaben, die zur Kompilierung des Projekts oder einem Teil davon notwendig sind.

2.2.5 Apache HTTP Web Server

Abbildung 2.6: Apache Web Server

Der *Apache HTTP Web Server*, auch oft als httpd bezeichnet, ist laut [Wik08c] der wohl am häufigsten im Internet eingesetzte Web Server. Er ist derzeit in Version 2.2.9 verfügbar und sowohl unter Windows als auch Linux sowie unter vielen weiteren Betriebssystemen einsetzbar. Er unterstützt mittlerweile Multiprocessing zur gleichzeitigen Abarbeitung von Client-Anfragen und ist durch seine modulare Bauweise umfangreich erweiterbar.
Der Apache HTTP Web Server wird in dieser Arbeit jedoch nur als zu managende Komponente verwendet. Er wird also nicht im direkten Produktiveinsatz verwendet.

2.3 Zustandsbehaftigkeit

Da wir in dieser Arbeit konkret das Management zustandsbehafteter Web Service Resources mit Apache Muse betrachten, gilt es natürlich, diese Begriffe näher zu erläutern um darzustellen, was konkret erreicht werden soll.
Zustandshaftigkeit bezeichnet dabei, dass der Web Service selbst über seinen Zustand bescheid weiß. Das bedeutet, er kann von sich selbst Informationen abfragen und diese verarbeiten.

2.4 Management mit Hilfe von Web Services

Als Management bezeichnet man die „Gesamtheit der Maßnahmen, die einen effektiven und effizienten, an den Zielen und Diensten eines Unternehmens ausgerichteten Einsatz eines vernetzten Systems samt seiner Anwendungen sicherstellen.“, wie in [Heg07] zu lesen ist.
Man unterscheidet dabei laut [Heg07] unter folgenden Teilaspekten:

- Konfigurationsmanagement
  Darunter fallen zum Beispiel die Neuinstallation von Hardware oder Software, sowie Anpassungen und Updates. Ebenso gehört in diese Kategorie die Maßnahmen bei einer Topologieänderung. Weitere Einstellungen wie Funktions-, Berechtigungs-, Last-, Protokoll-, Dienstgüte- und Anschlussparameter gehören zu diesem Teilaspekt.

- Fehlermanagement
  Zu diesem Teilaspekt gehört das Überwachen von Netz- und Systemzustand, sowie die Entgegennahme und Verarbeitung von Alarmen. Fehlerursachen müssen ermittelt werden, zum Beispiel mittels Log-Files, und nach Möglichkeit behoben werden.

- Leistungsmanagement
  Dieser Aspekt dient zur Bestimmung von Dienstgüteparametern und Überwachung auf Leistungsengpässen, sowie Durchführung von Messungen und Auswertung von History-Log-Files.

- Abrechnungsmanagement
  In diesen Teilaspekt wird das Namens- und Adressmanagement eingeordnet, sowie die Erfassung von Verbrauchsdaten und Zuordnung von Kosten zu Konten. Die Verwaltung von Kontingenten und Kundenprofilen fällt auch in diesen Bereich.

- Sicherheitsmanagement
  Bedrohungsanalysen und Festlegen sowie Durchführen von Sicherheitspolicys fallen in diesen Bereich. Autorisierung und Authentisierung sowie Zertifizierung gehören ebenso in diesen Teilaspekt. Des weiteren fällt in diesen Bereich die Überwachung von Angriffen.

Diese Management-Aufgaben sollen nun mittels umfangreichen Web Services, die sich gegenseitig ergänzen, erledigt werden.

2.5 Apache Muse

Die WSDM-Spezifikation wird konkret von Apache Muse in einer Java-Umgebung implementiert. Daher bietet es sich an, darauf aufzusetzen, da grundlegende Implementierungen bereits vorgenommen worden sind und die Spezifikationen somit eingehalten werden.
Apache Muse bietet also ein Java-Framework, das es uns ermöglicht, zustandsbehaftete Web Services zu entwickeln, die dem Management mit Hilfe von Web Services dienen und der WSDM-, WSRF-, und WSN-Spezifikation gerecht werden.

3 Installation und Konfiguration

Wir haben uns im letzten Kapitel mit den nötigen Definition und Konzepten auseinandergesetzt, so dass wir nun die verwendete Software näher anschauen. Im Folgenden wird erläutert, welche Software nötig ist, wo man sie herbekommt und wie man diese korrekt installiert sowie die Ersteinrichtung vornimmt. Damit wird das Grundgerüst gebildet, auf dem nachher gearbeitet wird.

3.1 Installation von Apache Muse

Als erstes wird die Installation von Apache Muse 2.2.0 erläutert. Unter [Apa08b] findet sich dazu auch ein englisch-sprachiges Tutorial, an das sich die folgenden Erläuterungen weitgehend halten.

Download und entpacken

Zuerst ist es selbstverständlich nötig, Apache Muse downzuloaden. Die jeweils aktuellsten Versionen der Binaries und des Quellcodes finden sich auf der Apache Web Services Homepage unter http://ws.apache.org/muse/download.html.
Nach einem erfolgreichen Download müssen die Archive entpackt werden. Im Folgenden betrachten wir nur das Binary-Archiv. Dies erfolgt durch den Befehl gzip -d muse-2.2.0-bin.tar.gz. Daraufhin enthalten wir ein unkomprimiertes Archiv, welches nun mit Hilfe von tar xvf muse-2.2.0-bin.tar extrahiert wird. Unter Windows kann natürlich einfach die Zip-Datei downgeloadet werden und mit Windows-üblichen Mitteln entpackt werden. Dies erzeugt nun einen Ordner muse-2.2.0-bin, in dem das Apache Muse Projekt vorzufinden ist. Unter Linux empfiehlt sich ein Entpacken nach /usr/local, unter Windows nach C:\Programme. Der Muse-Installationsordner enthält folgende Dateistruktur:

- /bin
  Enthält die Scripte wsdl2java und wsdlmerge für Windows und Unix.
- /docs
  Hier befindet sich eine ausführliche Dokumentation.
- /lib
  Enthält Bibliotheken, die bei der Code-Generierung verwendet werden.
- /lib/axis2
  Enthält eine Axis2 Umgebung.
- /lib/axis2-osgi
  Hier befindet sich die OSGi Umgebung, die wir jedoch nicht verwenden.

- /lib/common
  Drittanbieterbibliotheken von Apache Xerces, Apache Xalan und WSDL4J.
- /lib/eclipse-osgi
  Enthält Teile des Eclipse Equinox Projekts, das eine minimalisierte OSGi-Umgebung bietet. Auch das verwenden wir nicht.
- /modules
  Hier befindet sich das Herz von Muse, also die Runtime-Jars sowie die Implementierung der verwendeten Tools.
- /modules/axis2
  Spezielle Module für ein Axis2 Deployment.
- /modules/core
  Jar-Archive, die jede Muse Anwendung verwendet.
- /modules/mini
  Module für die Mini-SOAP engine. Wird in dieser Arbeit nicht verwendet.
- /modules/osgi
  OSGi-spezifische Module. Ebenso nicht verwendet.
- /modules/tools
  Alle Tools, die bei der Code-Generierung verwendet werden.
- /modules/ws-fx-api
  Hier finden sich die Java Interfaces der WS-Spezifikationen.
- /modules/ws-fx-impl
  Hier sind die dazugehörigen Java Implementierungen der WS-Spezifikationen, auf denen Muse-Anwendungen aufsetzen.
- /samples
  Einige grundlegende Beispiele sind hier zu finden.

Nachdem wir nun die Installation von Muse genauer angeschaut haben, können wir beginnen, Muse für unsere Ansprüche zu konfigurieren.

3.2 Konfiguration von Apache Muse

Damit man auf die im /bin Ordner beiliegenden Scripte zugreifen kann, empfiehlt es sich, diese dem Path hinzuzufügen. In dieser Arbeit wurde der Pfad /usr/local/muse-2.0.0-bin unter Linux, sowie C:\Programme\muse-2.0.0-bin unter Windows als Installationsverzeichnis gewählt. Unter Linux kann mittels

```
export PATH="$PATH:/usr/local/muse-2.2.0-bin/bin"
```

der Pfad nur für die aktuelle Session erweitert werden. Soll der Path jedoch dauerhaft erweitert werden, was sich empfiehlt, muss die Datei .profile im Home-Directory geändert werden. Dies geschieht durch Hinzufügen folgender Zeile mittels einem beliebigen Texteditor.

```
export PATH=$PATH:/usr/local/muse-2.2.0-bin/bin
```

Unter Windows wird der Pfad unter Systemsteuerung -> System -> Erweitert -> Umgebungsvariablen angepasst. In diesem Fenster kann nun die Systemvariable Path erweitert werden durch Anhängen des expliziten Installationspfades des Apache Muse. Zum Beispiel C:\Programme\muse-2.0.0-bin\bin . Es ist kein Neustart erforderlich, jedoch müssen bereits geöffnete Kommandozeilenfenster geschlossen und neu geöffnet werden.
Unter Linux ist als nächstes zu beachten, dass die Shellscripte ausführbar sind. Dazu muss man in das Installationsverzeichnis wechseln und wieter in das dort liegende /bin Directory. Der Befehl

```
chmod uog+x *.sh
```

gewährleistet die nötigen Rechte. Beim Einsatz von Windows sind die Batch-Dateien automatisch ausführbar.
Apache Muse ist hiermit fertig konfiguriert und für den Produktiveinsatz bereit. Im nächsten Kapitel wird noch näher auf die Funktionsweise eingegangen.

3.3 Installation und Konfiguration von Apache Tomcat

Apache Muse bringt bereits alle notwenigen Apache Axis2 Komponenten mit out-of-the-box, sodass Axis2 nicht extra installiert werden muss. Somit können wir uns als nächstes dem Apache Tomcat zuwenden.
Apache Tomcat ist notwendig, damit die Web Services die mittels Apache Muse erstellt und in ein Apache Axis2 Container deployed wurden, auch erreicht werden können und darauf zugegriffen werden kann. Zunächst ist natürlich sowohl unter Linux als auch Windows ein Download von der offiziellen Tomcat Seite http://tomcat.apache.org/ notwendig, sowie eine nachfolgende Extraktion des Archives von nöten. Unter Windows empfiehlt sich eine Installation nach C:\Programme, unter Linux nach /usr/local/
Wir sollten als nächstes einen Manager konfigurieren. Dies erfolgt durch Editieren der Datei conf/tomcat-users.xml im Tomcat-Installationsverzeichnis. In diesem Fall legen wir einen Benutzer mit dem Namen manager und dem Passwort s3cret an. Die Datei schaut dann wie folgt aus.

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
<role rolename="manager"/>
<user username="manager" password="s3cret" roles="manager"/>
</tomcat-users>
```

Es wurde sowohl unter Windows als auch Linux mit der Version 6.0.18 gearbeitet.
Besonders zu beachten ist jedoch folgende Inkompatibilität: Muse und Axis2 ist es relativ egal, mit welcher Java-Version die Java-Klassen generiert und kompiliert werden. Tomcat als Standalone-Server ist es ebenso egal. So wurde bei der Entwicklung der Arbeit bereits mit der aktuellen Java 1.6 Version mit aktuellstem Update gearbeitet. Erst bei Zugriff auf die WS Resources warf Tomcat immer nur 500 Internal Server errors und zwar ohne Begründung und völlig unverständlich. Erst nach langer Recherche ist das Problem zu Tage gekommen. In Muse generierte Anwendungen mit einer Axis2 Umgebung laufen nicht in einem Tomcat,

wenn dieser mit Java 1.6 arbeitet! Erst ein Downgrade auf Java 1.5 mit dazu aktuellstem Update konnte das Problem beheben! Dieses Problem trat sowohl unter Windows als auch Linux auf.
Daher ist für die Nachvollziehung dieser Arbeit dringend ein Downgrade auf Java 1.5 zu empfehlen, und zwar sowohl als Laufzeitumgebung als auch als JDK.
Der /bin Ordner von Tomcat sollte mit äquivalenten Mitteln wie auch der /bin Order von Muse der Path Variablen hinzugefügt werden. Auch müssen unter Linux die Shellscripte eventuell ausführbar gemacht werden.

3.4 Installation von Apache Ant

Unter Linux ist Apache Ant einfach über yast zu installieren und in den meisten Fällen sowieso schon standardmäßig korrekt installiert.
Windows Anwender müssen jedoch Apache Ant von der Homepage downloaden. Die aktuellste Version findet man unter http://ant.apache.org/bindownload.cgi etwas weiter unten. Das Archiv muss selbstverständlich ebenso wie die anderen Programme entpackt werden. Auch hier bietet sich als Installationsordner C:\Programme an. Das /bin Verzeichnis sollte natürlich auch zum Pfad hinzugefügt werden. Unter Linux wird dies bei der Installation per yast automatisch erledigt.
Ant benötigt keine weitere Konfiguration, sondern ist direkt nach Installation und der Pfad-Anpassung unter Windows einsatzfähig.

3.5 Installation des Apache HTTP Web Servers

In einer Windows Umgebung muss der HTTP Server zuerst von http://httpd.apache.org/ als MSI (Microsoft Installer) downgeloadet werden und selbsterklärend installiert werden. Unter Windows empfiehlt sich eine Installation nach C:\Programme. Unter Linux muss die aktuelle Version als Quelltext von http://httpd.apache.org/ downgeloadet werden und entpackt werden. Anschließend wird im entpackten Ordner

```
./configure -prefix=/usr/local/apache2
```

aufgerufen, gefolgt von

```
make
```

und

```
make install
```

Danach ist die Installation abgeschlossen und der Web Server ist unter /usr/local/apache2 zu finden. Dieses Verzeichnis müssen wir später als installDir_Linux konfigurieren bei der Erstellung unseres Web Services.
Unter beiden Betriebssystemen ist zu beachten, dass der HTTP Server als Dienst installiert wird, was jedoch standardmäßig der Fall ist. Außerdem ist noch wichtig, dass die httpd.conf nicht schreibgesschützt ist, da wir später diese Datei abändern. Zu beachten gilt überdies hinaus auch, dass einmalig in der httpd.conf die Zeile

```
#ServerName www.example.com:80
```

abgeändert wird, so dass das führende #-Symbol entfernt wird!

4 Anwendung der Apache Software

Da wir nun das System vollständig installiert haben, können wir als nächstes die Software näher betrachten.

4.1 Beschreibung und Funktionsweise von Apache Muse

Apache Muse bietet grundlegend folgende Implementierung und Funktionen.

- Implementierung des Web Service Resource Frameworks (WSRF) 1.1
- Implementierung der Web Service Notification (WSN) 1.3 Spezifikation
- Implementierung der Web Service Distirbuted Management (WSDM) 1.1 Spezifikation
- Implementierung der WS-MetadataExchange (WS-Mex) port types
- Implementierung der WSDM Event Format 1.1 Spezifikation
- Kompatibilität mit WS-Addressing 1.0 und SOAP 1.2
- Deployment Funktion in Axis2 und OSGi Umgebungen
- Viele Utility-APIs für die gebräuchlichsten Funktionen, nämlich ResourceProperties, Service Groups, Relationsships, Publish-Subscribe Szenarios und Resource Selbstüberprüfung.
- Eine Möglichkeit, den State einer WS Resource auch nach einem Shutdown des Host-Systems (also in unserem Fall Tomcat) zu behalten.
- Ein WSDL2Java-Tool, das Server- und Client-seitigen Java-Skeleton-Code aus WSDL-Dokumenten generiert.

Der Entwickler kann sich also die Vorarbeit sparen, sämtliche Standards und Spezifikationen zu implementieren. Er kann direkt mit der Entwicklung des tatächlichen WSDL-Files beginnen und seine Funktionen definieren, um anschließend innerhalb der Skeletons die Methoden und Logik zu implementieren.

4.2 wsdl2java

4.2.1 Kurze Einführung

WSDL2Java dient zur Code-Generierung anhand eines WSDL-Dokuments. Dazu schauen wir uns zunächst die Funktionsweise von wsdl2java näher an.
Mit Hilfe dieses Programms ist es möglich anhand des zu bearbeitenden WSDL-Files die

Server-seitigen Skeletons und die Client-seitigen Proxies zu erstellen. Falls wsdl2java ohne Parameter gestartet wird, wird nur eine kleine Hilfe angeboten um die grundlegenden Funktionen zu erklären.

```
Usage: wsdl2java.[bat|sh] PLATFORM [CONTAINER] -wsdl FILE [OPTIONS]

The following arguments are required:
  -wsdl FILE            The WSDL definition file to analyze

  PLATFORM              Must be one of the following:
    -j2ee               Create a J2EE project
    -osgi               Create a OSGi project
    -proxy              Create a Proxy project

  CONTAINER             Specify one of the following for J2EE or OSGi:
    axis2               Create a Axis2 1.1 container
    mini                Create a Mini SOAP Engine container

The following arguments are optional:
  -output DIR           Specify an output direcgtory
  -overwrite            Overwrite files that exist
  -help                 Display this message
  -helpmore             Display more advanced help message
```

Die wichtigsten Parameter sind im folgenden näher erläutert.

- **-wsdl** deklariert die zu bearbeitende WSDL-Datei.
- **-j2ee** erstellt ein J2EE Projekt in einem WAR-File (Web ARchive File)
- **-osgi** erstellt ein OSGi Projekt in einem OSGi Bundle JAR-File (Java ARchive File)
- **-proxy** erstellt ein Proxy Projekt, also ein JAR-File, das die clientseitigen Klassen enthält.
- **axis2** dient zum Erstellen eines Axis2 1.1 Containers, um die Muse Umgebung zu hosten. Der Axis2 Container wird in dieser Arbeit verwendet.
- **mini** dient zu Verwendung der Muse Mini Servlet Engine, der einen deutlich reduzierten Funktionsumfang im Vergleich zum Axis2 Container besitzt.
- **-output** definiert das Ziel-Verzeichnis. Sollte dieser Parameter weggelassen werden, so wird das aktuelle Verzeichnis verwendet.
- **-overwrite** überschreibt existierende Dateien ohne nachzufragen.
- **-verbose** gibt ausführliches Feedback der durchgeführten Umwandlung an.
- **-help** zeigt oben dargestellt Hilfe an.
- **-helpmore** zeigt die ausführliche Hilfe an, wie sie auch unter [Apa08c] zu finden ist.

4.2.2 Übersicht der Architektur

Folgenden Funktionsumfang bietet wsdl2java, wie in [Apa08c] zu lesen ist.

- **Capability Extrahierung**
  - Anhand des Namespaces können Capabilities gruppiert werden. Dadurch kann wsdl2java herausfinden, welche Properties (aus dem Properties Dokument) und welche Operations (basierend auf dem Namespace der Input Elemente der Nachricht) zu einem gemeinsamen Namespace gehören.
  - Für jede Capability werden eigene Klassen, die die nötigen Operationen und Properties implementieren, generiert.
- **Deployment Artefakt Generierung**
  - Anhand dem WSDL-File müssen die Muse Deployment Artifakte generiert werden. Jeder Muse Endpoint muss einen Muse Deployment Descriptor besitzen, welcher in der Regel aus dem Inhalt des WSDL-Files gefolgert werden kann.
- **Platform spezifische Artefakt Generierung**
  - Jede Platform benutzt spezifische Dateien und besitzt eine spezifische Dateistruktur. Daher versucht wsdl2java die nötige Dateistruktur automatisch zu erstellen sowie hilfreiche default values, die beim Deployment der Artikate nötig sind, automatisch anzulegen.
  - Aus einem WSDL-File können verschiedene Projekte für verschiedene Umgebungen generiert werden.
- **Das Java Projekt**
  - Bei jeder Durchführung wird automatisch die notwendige Verzeichnisstruktur und die darin enthaltenen Klassen und Packages angelegt.
  - Ein ant-Script wird erstellt, um alles zu kompilieren und in ein WAR-File zu bundeln.
  - Alle Platform-spezifischen Dateien werden generiert, die nötig sind, um ein fertiges deployfähiges WAR-File zu bauen.

Diese Ziele werden durch eine Aufteilung des wsdl2java-Algorithmus in drei Bereiche erreicht.

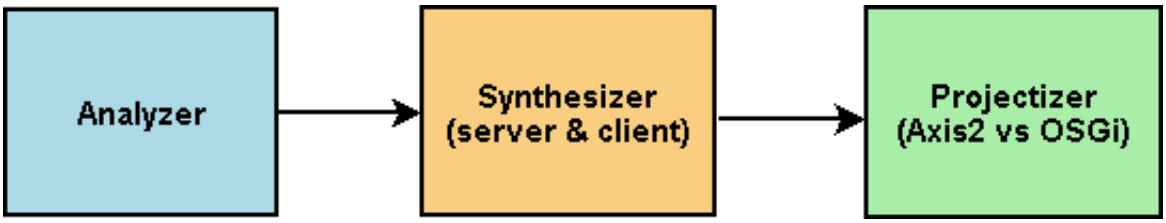

Abbildung 4.1: wsdl2java Komponenten

Im Folgenden sind die drei Bereiche und deren Funktionen näher erläutert.

4.2.3 Analyzer

Der Analyzer produziert eine Map anhand jeder Capability, die im WSDL-File gefunden wurde. Diese Mape stellt dar, welche Capability URIs zu welchem Capability Object gehören.

Capability Extraction

Die Capability Extraktion kann folgendermaßen zusammengefasst werden.

- Einlesen der WSDL-Datei mittels wsdl4j.
- Ermitteln des Services, des zugehörigen Ports und durchlaufen der portTypes zum gesuchten Port.
- Iterierung über jede Operation in diesem portType. Die namespace URI der Input Message wird als namespace für die Operation verwendet. Danach wird die zugehörige Capability in der URI-Capability-Map gesucht und die Informationen über die Operation Data der Capability hinzugefügt.
- Jetzt werden die ResourceProperties gesucht. Zunächst wird dazu das zugehörige Element in den Schemata gesucht und anschließend über die Liste der Elemente iteriert und jedes Element als Property zur gegebenen Capability (basierend auf der Namespace URI des Elements) hinzugefügt.
- Nun haben wir eine Map generiert, die zu jeder URI die Capability Objekte abbildet.

Eine Capability wird nur dann angelegt, wenn eine zugehörige Operation oder Property gefunden wurde. Daher ist es nicht möglich mit Hilfe eines WSDL Descriptors leere Capabilities (also Capabilities die weder Operationen noch Properties enthalten) anzulegen. Falls jedoch ein Muse Deployment Descriptor verwendet wird während der Code Generierung und falls eine Capability definiert ist, die nicht standarmäßig vorhanden ist und nicht im WSDL-Dokument auftaucht, dann wird eine leere Capability während der Code Erstellung angelegt. Diese leeren Capabilities sind hilfreich für serverseitige Aufgaben, die nicht durch Standard-Capabilites abgedeckt werden.
Die Code Generierung berücksichtigt außerdem die Implementierung der Standard Capabilities. Das bedeutet, dass, wenn wsdl2java Elemente findet, die durch Standard built-in

Capabilities bereits abgedeckt sind, der Muse Descriptor die built-in Muse Klasse als Standardimplementierung referenziert. Daher kann der Benutzer sich auf seine eigenen benutzerdefinierten Capability Implementierungen konzentrieren und wsdl2java kümmert sich darum, dass die notwendigen anderen Capabilities hinzugefügt werden.

WS-Notification Producer Support

Normalerweise wird anhand eines WSDL-Dokuments ein Server-seitiges Projekt angelegt, das genau einen resource-type im zugehören Muse Deployment Descriptor besitzt. Jedoch müssen WS-Notification Producers einen zweiten resource-type besitzen, der sich um die Anmeldungen für Benachrichtigungen kümmert. Anstelle, dass die Benutzer dazu gezwungen werden, einen Deployment Descriptor für den WS-NotificationProducer anzulegen, kümmert sich wsdl2jave automatisch darum, die nötigen Artifakte hinzuzufügen und erweitert den generierten Deployment Descriptor um einen Resource-Type, der den Subscription Manager implementiert.

4.2.4 Synthesizer

Der Synthesizer bearbeitet nun die Map der Capabilities, die vom Analyzer angelegt wurde. Er generiert den notwendigen Skeleton Code.

Server Stub Code Gernerierung

Die Skeleton-Klassen auf Serverseite bestehen aus Interfaces, die von einer Unterklasse von Capability (oder WSCapability) implementiert werden und grundlegende Methoden und deren Signaturen definieren.

Client Proxy Code Generierung

Hier wird eine einzige Client-seitige Klasse angelegt, die alle Operationen und Methoden der Capabilities enthält. Diese Klasse konvertiert lokale Requests in Remote Aufrufe.
Wenn ein Deployment Descriptor zur Generierung verwendet wird, wird für jeden resource-type im Descriptor ein jeweiliger Proxy angelegt.

4.2.5 Projectizer

Als letzte logische Komponente kommt nun der Projectizer zum Einsatz. Dieser nimmt sich die Maps aus den vorherigen zwei Vorgängen und entwirft das nötige Projekt Layout für die Ziel-Platform. Der Projectizer ist verantwortlich dafür, dass eine korrekte Verzeichnisstruktur und dazugehörige Scripte je nach Ziel-Platform angelegt werden.
Da es mehrere verschiedene Zielplatformen gibt, gibt es auch verschiedene Projectizer.

J2EE Projectizer Family

Die komplizierteste Platform ist die J2EE Platform. In der aktuellen Version unterstützt wsdl2java zwei verschiedene Engines, nämlich Axis2 und Muse Mini Soap Servlet. Der J2EE Projectizer legt nur ein deploybares WAR-File an, das in einen J2EE Container deployed werden muss, wie zum Beispiel Apache Tomcat.

OSGi Projectizer Family

Im Gegensatz zur J2EE Platform kann die OSGi Platform einfach self-contained sein. Der OSGi Projectizer generiert alles notwendige, um den Endpoint direkt zu referenzieren. Der Projectizer verwendet das Equinox Framework als OSGi Implementierung. Die aktuelle Version von wsdl2java unterstützt beim OSGi Projectizer genauso wie beim J2EE Projectizer zwei Engines, nämlich Axis2 und Muse Mini Soap Servlet. Diese Engines dienen als Isolations Layer, die Muse von der Umgebung trennt.
Der OSGi Projectizer legt die notwenigen Descriptor Dateien an und kopiert alles nötige, um das gesamte Projekt direkt in einer OSGI Umgebung ausführen zu können.

Wie oben beschrieben untersützten diese beiden Projectizer folgende Engines.

Axis2 Engine Das Axis 2 Engine Projekt beinhaltet zwei Verzeichnisse (WebContent und JavaSource) sowie eine build.xml Datei (für Ant) und eine .overwrite Datei. Außerdem werden die nötigen Descriptors (muse.xml und services.xml) angelegt. Durch den Aufruf von ant wird das zugehörige WAR-File generiert.

Muse Mini Servlet Engine Obwohl Axis2 bereits eine robuste Platform anbietet, um Muse Endpoints zu hosten, hat die Muse Community auch die sogenannte Muse Mini Servlet Engine implementiert. Diese Engine ist deutlich weniger komplex als Axis2, und für viele Benutzer absolut ausreichend. In seltenen Fällen, wie beim Einsatz von J2ME, also auf mobilen Devices, kann Axis2 gar nicht eingesetzt werden, so dass dort immer auf die Muse Mini Servlet Engine zurückgegriffen werden muss.

Proxy Projectizer

Der Proxy Projectizer generiert Java-Code, mit dem alle Porperties und Operationen des zuvor generierten „eigentlichen“ Projekts angesprochen werden können. Damit kann mit den Endpoints der Services innerhalb eines Java-Projects interagiert werden.

4.3 Beschreibung und Funktionsweise von Apache Axis2

Apache Axis2 dient in unserem Fall als Container für die Muse-Anwendungen. Axis2 empfängt SOAP-Requests und sendet SOAP-Responses zurück. Außerdem ist er in unserem Fall für WS-Addressing zuständig, damit die Web Services überhaupt angesprochen werden können. Das bedeutet, alle SOAP-Nachrichten gehen von Axis2 aus beziehungsweise zu Axis2 hin. Axis2 hostet die Web Services und verwaltet deren Beschreibung und EPRs. Die Interpretation von SOAP-Nachrichten liegt also allein bei Axis2. Dazu werden die SOAP-Messages aufgeteilt in ihre ursprünglichen Teile, nämlich dem Header und dem Body. Zuerst wird natürlich der Header interpretiert und daraus die Ziel-URI ausgelesen, also der EPR des Web Services. Anschließend können im Body die Nutzdaten ausgelesen werden und inpretiert werden. Die Nutzdaten werden dann an die zuständigen, also den per EPR vom Header, erreichbaren Web Service weitergereicht.

4.4 Beschreibung und Funktionsweise von Apache Tomcat

Apache Tomcat ist ein Serlet-Container für beliebige Java-Servlets unter anderem eben auch Axis2-Servlets. Tomcat bietet in unserem Fall die Möglichkeit Verbindungen entgegenzunehmen und an die jeweiligen Servlets weiterzuleiten. In unserem Fall handelt Tomcat die TCP-Verbindungen von (externen) Clients. Die Verbindungen werden dabei bereits explizit an ein Servlet gerichtet, so dass Tomcat die Nutzdaten direkt an das jeweilige Servlet weitergeben kann. In unserem Fall wird es später pro Web Service je ein Servlet geben. Das bedeutet, wir können durch Servlet Adressierung auch bereits impliziert einen Web Service adressieren. Nachdem nun Tomcat die Nutzdaten, in unserem Fall also SOAP-Messages, an das jeweilige angesprochene Servlet weitergegeben hat, ist die Aufgabe von Tomcat auf der Eingangsseite beendet. Da unser Servlet ja jeweils eine Axis2-Umgebung darstellt mit einem zugehörigen Web Service, können die weitergereichten SOAP-Nachrichten interpretiert werden.
Tomcat kommt natürlich auch zum Einsatz, wenn unser Servlet, also Axis2, eine SOAP-Nachricht versenden will. Dann muss Tomcat diese Nachricht in ein normales IP-Paket verpacken und an die gewünschte Ziel-Adresse versenden.

5 Entwicklung des Management Szenarios

In den vorangegangenen Kapiteln haben wir uns nun Schritt für Schritt an die Materie herangetastet. Nun geht es weiter mit der eigentlichen Implementierung des Anfangs vorgestellten Szenarios. Wir werden diese Implementierung schrittweise vornehmen. Diese Schritte sind dabei grob unterteilt in folgende Teile: Zuerst entwickeln wir das WSDL-Dokument unseres Haupt Web Servcies, der für WSDM zuständig ist, sowie das WSRF verfügbar macht und als WSN-Producer dient. Danach implementieren wir die eigentliche Logik in Java Klassen. Gemeinsam mit dem WSDL-Dokument erstellen wir danach ein Axis2 Servlet.
Damit unsere per WSN gepublisheten Nachrichten auch entgegengenommen werden können, entwickeln wir anschließend noch einen winzigen zweiten Web Service, der nur dazu dient, die Nachrichten anzunehmen und zu interpretieren.
Daraufhin werden wir beide Servlets in eine Tomcat Umgebung einsetzen wo wir sie dann mittels einem kleinen GUI, das wir entwickeln, testen werden.
Doch beginnen wir nun von Anfang an, nämlich mit der Entwicklung des WSDL-Dokuments für unseren Haupt Web Service.
Dazu erstellen wir uns zuerst ein Arbeitsverzeichnis, in dem wir all unsere Implementierungen abspeichern und generieren. In dieser Arbeit wird c:\myMuse unter Windows verwendet sowie ∼/myMuse unter Linux, im folgenden wird darauf mit „workDir“ referenziert. Da wir im Verlauf der Arbeit zwei Web Services entwickeln werden, brauchen wir auch zwei Unterverzeichnisse unter workDir. Unseren Haupt Web Service entwickeln wir also unter workDir/FoPra/ und den zweiten kleinen Web Service unter workDir/Consumer/. Doch dazu später mehr.
Während der ganzen Arbeit ist zu beachten, dass unter Linux die nötigen Rechte für den Benutzer gesetzt sind, da ansonsten auf Grund Zugriffsbeschränkungen der Web Service nicht auf die benötigten Dateien zugreifen kann und eventuell auch keine Rechte hat den Apache Server zu starten und zu stoppen. Auch der Tomcat Server kann nicht von einem normalen Nutzer gestartet und beendet werden. Die Rechtevergabe ist somit vorab zu klären. Zur Not kann man als root arbeiten, aber darauf sollte man grundsätzlich verzichten und eine Rechteverwaltung bevorzugen! Unter Windows XP kann der normale Benutzer alle von uns durchgeführten Operationen erledigen ohne in irgendeiner Weise eingeschränkt zu sein. Unter Windows Vista werden die Server durch das UAC beschränkt, so dass auch hier eventuell eine Rechtevergabe notwendig ist.

5.1 Entwicklung des WSDL-Dokuments für den Haupt Web Service

Da wir Management mit Hilfe von Web Services betreiben wollen, ist es natürlich notwendig die zugehörigen Services zu erstellen. Apache Muse bietet dazu ein Template an, das bereits alle Properties und Operations beinhaltet, die WSRF, WSN und WSDM definieren. Dieses WSDL-Template wird von Apache auf deren Muse-Homepage zum Download unter

http://ws.apache.org/muse/docs/2.2.0/tutorial/artifacts/muse-template.wsdl angeboten. Apache Muse empfiehlt dringend, die eigenen WSDL-Dokumente auf diesem Template aufzubauen und das Template als Grundlage zu verwenden. In dieser Arbeit erweitern wir dieses Template Stück für Stück um auf unseren gewünschten Funktionsumfang zu kommen. Das Template speichern wir unter workDir/FoPra/wsdl/muse-template.wsdl ab und nennen es um in FoPra.wsdl.
Das WSDL-Dokument setzt sich dabei aus folgenden Teilen zusammen:

- <wsdl:definitions> ist das Wurzelelement, in dem per Attribute alle Namespaces definiert werden, sowie der Name der WS Resource.
  - <wsdl:types> hierunter befinden sich mehrere <xsd:schema> Elemente, die die jeweiligen benötigten xsd-Schemata referenzieren. Außerdem befinden sich hier die zentralen <xsd:schema> Elemente, unter denen sich die Properties befinden.
  - <wsdl:message> listet die möglichen SOAP-Nachrichten auf. So gibt es hier die von Muse standardmäßig implementierten Requests, Responses und Faults.
  - <wsdl:portType> hier werden die von der WS Resource unterstützten Operationen gelistet und SOAP-Actions an wsa:actions gebunden.
    - <wsdl:operation> Je ein Operation Element enthält ein <wsdl:input> sowie ein <wsdl:output> Element und gegebenenfalls mehrere <wsdl:fault> Elemente. In jedem dieser Elemente wird dabei auf eine wsa:action referenziert.
  - <wsdl:binding> enthält mehrere Operation-Elemente.
    Prinzipiell dient das <wsdl:binding> Element dazu, bestimmte SOAP-Actions an die zugehörigen SOAP-Nachrichten zu binden.
    - <wsdl:operation> Jedes Operation Element bindet SOAP-Nachrichten an die zugehörigen <wsdl-soap:operation>
  - <wsdl:servic> Hier befindet sich vor allem die Adresse, unter der dem WSDL-Dokument zugehörige Service erreichbar ist, also der EPR.

Wir sehen also, dass das WSDL-Dokument nur der Beschreibung des Web Services dient und in keinster Weise für die Implementierung und Funktionsweise dessen verantwortlich ist.
Nachdem wir nun einen kleinen Blick auf das WSDL-Dokument geworfen haben, das Muse als Template beiliegt, können wir anfangen, das WSDL-Dokument zu bearbeiten. Dazu bieten sich viele verschiedene Editoren an. Einen grafischen WSDL-Editor, wie es zum Beispiel als Plugin für Eclipse gibt, benötigen wir jedoch nicht, da wir uns direkt mit dem Code beschäftigen. Das Plugin „Improve WSDL Viewer“ für Eclipse (zu finden per Update Manager unter http://www.improve-technologies.com/alpha/updates/site.xml) bietet sowohl die Möglichkeit ein WSDL-Dokument grafisch darzustellen, als auch direkt den Source-Code zu bearbeiten. Unter Windows bietet sich auch das Open Source Tool Notepad++ an, das WSDL-Dokumente mittels Syntax-Highlighting sehr ansprechend darstellen kann.
Nun können wir das FoPra.wsdl Dokument öffnen und bearbeiten. Wir ändern zuerst grundlegende Sachen ab, wie den Namespace sowie den Namen der Resource.
Dazu ändern wir in Zeile 2 den im Template angegebenen Namespace ab in den von uns gewünschten Namespace:

```
<wsdl:definitions targetNamespace="http://ws.apache.org/muse/fopra/httpd"
```

Natürlich muss dann auch der tns (ThisNameSpace) in Zeile 3 in den äquivalenten Namespace umgeändert werden, nämlich

```
xmlns:tns="http://ws.apache.org/muse/fopra/httpd"
```

Eine weitere Namespaceanpassung benötigen wir in Zeile 64, 65, ab wo die eigentlichen ResourceProperties definiert werden.

```
<xsd:schema elementFormDefault="qualified"
  targetNamespace="http://ws.apache.org/muse/fopra/httpd">
```

Nun haben wir die individuellen Namespaces angepasst, jedoch muss dies auch noch für jede einzelne Operation erledigt werden. Das bedeutet, am besten wird per „Suchen und ersetzen“ diese Aufgabe per Editor erledigt.
Als nächstes müssen wir natürlich den EPR an unsere Wünsche anpassen. Ganz unten im WSDL-Template gibt es den Tag <wsdl:service>, diesen editieren wir wie folgt:

```
<wsdl:service name="WsResourceService">
 <wsdl:port name="WsResourcePort" binding="tns:WsResourceBinding">
  <wsdl-soap:address
    location="http://localhost:8080/httpd/services/http_server" />
 </wsdl:port>
</wsdl:service>
```

Hierbei ist zu beachten, dass die Adresse keine Zeichen enthält, mit denen Java nicht klarkommt. So darf zum Beispiel http-server nicht als Adresse genommen werden, da es keine gültige Java Klasse ist! Nun wäre das WSDL-Dokument bereits einsatzbereit und definiert alle von Muse implementierten Spezifikationen. Jedoch ist es noch ohne genauere Funktionen, die wollen wir nun erarbeiten. Im aktuellen Zustand beschreibt das WSDL-Dokument folgende Funktionen:

- WS-ResourceProperties
- WS-MetadataExchange
- WSN NotificationProducer
- WSDM MUWS Identity
- WSDM MUWS Description (mit Caption, Description und Version Properties)
- WSDM MUWS OperationalStatus
- Properties

Wir möchten jedoch in unserem Beispiel den Apache HTTP Web Server per WSDM managen, also müssen wir mittels dem WSDL-Dokument noch einige weitere Funktionen beschreiben. So ist es natürlich obligatorisch, dass der Server gestartet und beendet werden kann, also benötigen wir auf jeden Fall die Funktionen start und stop. Außerdem benötigen wir noch die Elemente, die wir später mittels den ResourceProperties anpassen möchten.
Diese Definitionen müssen wir nun in unserem WSDL-Dokument einbauen. Dazu bearbeiten wir nun das FoPra.wsdl Dokument und erweitern es Stück für Stück.

Zuerst fügen wir grundlegende Funktionen hinzu im <xsd:schema> Element unseres eigenen Namespaces, an die Stelle, an der auch schon ServerName und MessageInterval gelistet sind. Wir benötigen hier weitere <xsd:element> Elemente, nämlich:

```
<xsd:element name="Port" type="xsd:integer"/>
<xsd:element name="ServerName" type="xsd:string"/>
<xsd:element name="installDir" type="xsd:string" />
<xsd:element name="Start"/>
<xsd:element name="StartResponse"/>
<xsd:element name="Stop"/>
<xsd:element name="StopResponse"/>
```

Hiermit haben wir nun sowohl die benutzerspezifischen Properties als auch Funktionen als „vorhanden“ definiert. Damit sind wir aber noch nicht fertig, wie benötigen auch Faults, die im Falle eines Fehlers zurückgesendet werden. Dazu erweitern wir nochmals unser Haupt <xsd:schema> Element um folgende Elemente:

```
<xsd:element name="StartFailedFault">
 <xsd:complexType>
  <xsd:complexContent>
   <xsd:extension base="wsrf-bf:BaseFaultType" />
  </xsd:complexContent>
 </xsd:complexType>
</xsd:element>
<xsd:element name="StopFailedFault">
 <xsd:complexType>
  <xsd:complexContent>
   <xsd:extension base="wsrf-bf:BaseFaultType" />
  </xsd:complexContent>
 </xsd:complexType>
</xsd:element>
```

Nun definieren wir noch einiges innerhalb des <xsd:element name=“WsResourceProperties“> Elements, dem darunterliegenden <complexType> und darunterbefindlichen <xsd:sequenz>. Hier werden alle Properties definiert, auf die wir später zugreifen können.

```
<xsd:element ref="tns:Port"/>
<xsd:element ref="tns:installDir"/>
```

Wir definieren hier die Properties Port und installDir als Elemente in „thisNamespace“ = „tns“ und CurrentTime sowie TerminationTime dem Namespace der ResourceLifetime zugehörig. Servername und MessageInterval sind bereits vom Template implementiert als Standard der Spezifikation.
Als nächstes müssen wir die akzeptierten SOAP Messages definieren. Alle akzeptierten SOAP-Nachrichten werden in getrennten <xsd:messages> Elementen abgelegt. Um unsere definierten Properties abzufragen oder zu ändern, sind keine weiteren Messages notwendig, da dies bereits mittels den Get-/SetResourcePropertiesRequests und -Responses möglich ist, die standardmäßig bereits definiert sind, da sie zu jeder Muse-Management-Umgebung gehören.

Jedoch unsere individuellen Requests, nämlich das Starten und Stoppen des Apache HTTP Web Servers erfordert eine explizite Definition. Daher erstellen wir folgende Knoten:

```
<wsdl:message name="StartRequest">
 <wsdl:part name="StartRequest" element="tns:Start"/>
</wsdl:message>
<wsdl:message name="StartResponse">
 <wsdl:part type="xsd:anyType"/>
</wsdl:message>
<wsdl:message name="StartFailedFault">
 <wsdl:part name="StartFailedFault" element="tns:StartFailedFault"/>
</wsdl:message>
<wsdl:message name="StopRequest">
 <wsdl:part name="StopRequest" element="tns:Stop"/>
</wsdl:message>
<wsdl:message name="StopResponse">
 <wsdl:part type="xsd:anyType"/>
</wsdl:message>
<wsdl:message name="StopFailedFault">
 <wsdl:part name="StopFailedFault" element="tns:StopFailedFault"/>
</wsdl:message>
```

Wir sehen hier die jeweiligen Namen der Messages. So gibt es für jede Funktion sowohl eine Request-Message, als auch Response- und Fault-Definition. Des weiteren ist zu sehen, dass innerhalb des <wsdl:part> Elements auf Operationen verwiesen wird. Dieser Zusammenhang wird aber explizit noch in den Bindings definiert, zu denen wir uns als nächstes wenden.
Betrachten wir nun den <wsdl:binding> Knoten genauer. Darunter befinden sich mehrere <wsdl:operation> Elemente, die zu jedem Request, die zugehörige Operation, also Funktion, sowie Response und Faults definieren. Legen wir also nun unsere beiden Operationen an, nämlich Start und Stop.

```
<wsdl:operation name="Start">
 <wsdl-soap:operation soapAction="Start" />
 <wsdl:input name="StartRequest">
  <wsdl-soap:body
    use="encoded"
    encodingSytle="http://schemas.xmlsoap.org/soap/encoding/"/>
 </wsdl:input>
 <wsdl:output name="StartResponse">
  <wsdl-soap:body
    use="encoded"
    encodingSytle="http://schemas.xmlsoap.org/soap/encoding/"/>
 </wsdl:output>
 <wsdl:fault name="StartFailedFault">
  <wsdl-soap:fault
    use="encoded"
    name="StartFailedFault"/>
 </wsdl:fault>
```

```
 <wsdl:fault name="ResourceUnknownFault">
  <wsdl-soap:fault
    use="encoded"
    name="ResourceUnknownFault" />
 </wsdl:fault>
 <wsdl:fault name="ResourceUnavailableFault">
  <wsdl-soap:fault
    use="encoded"
    name="ResourceUnavailableFault" />
 </wsdl:fault>
</wsdl:operation>
```

Damit ist die Start-Operation definiert und die Bindings festgelegt. Insbesondere ist zu sehen, dass jede Operation einige StandardFaults binden muss, so muss natürlich immer ein ResourceUnkownFault sowie ein ResourceUnavailableFault gebunden werden, im im selbsterklärenden Falle respondet werden. Analog zur Start Operation muss nun noch die Stop-Operation definiert werden.
Da wir nun alle Bindungen definiert haben, kommt noch die letzte Aufgabe, nämlich die portTypes zu definieren. Innerhalb des portType Knotens finden wir mehrere <wsdl:operation> Elemente, die jeweils zu einer Operation die wsa:Actions definieren, also die ganz zu Anfang definierten Start, StartResponse sowie Stop und StopResponse Funktionen und die dazugehörigen Input-Messages und Output-Messages und Faults festlegen. Dazu fügen wir innerhalb des <portType> Knotens zwei weitere <wsdl:operation> Elemente ein mit folgendem Inhalt:

```
<wsdl:operation name="Start">
 <wsdl:input wsa:Action="http://ws.apache.org/muse/fopra/httpd/Start"
   name="StartRequest" message="tns:StartRequest" />
 <wsdl:output wsa:Action="http://ws.apache.org/muse/fopra/httpd/StartResponse"
   name="StartResponse" message="tns:StartResponse" />
 <wsdl:fault name="StartFailedFault"
             message="tns:StartFailedFault" />
 <wsdl:fault name="ResourceUnkownFault"
             message="tns:ResourceUnknownFault"/>
 <wsdl:fault name="ResourceUnavailableFault"
             message="tns:ResourceUnavailableFault"/>
</wsdl:operation>
```

Analog zum Start-Knoten muss natürlich auch ein Stop-Knoten angelegt werden.
Hier schließt sich also der Kreis.
Wir haben bis jetzt zuerst die Funktionen und Properties definiert, sowie die nötigen Bindungen von Operationen an SOAP-Nachrichten vorgenommen, um als letzes diese alle zu verknüpfen.
Mit diesem WSDL-Dokument könnten wir nun schon einen Web Service beschreiben, der innerhalb der MUWS-Spezifikation einen Apache HTTP Web Server starten und stoppen kann, sowie den Servernamen, Port und Installationsverzeichnis bearbeiten kann. Das Muse Template beschreibt darüber hinaus bereits alle für WSN nötigen Operationen und Messages, so dass wir hierfür nichts mehr ergänzen müssen.

5.2 Erstellung der Java Klassen

Da wir nun das WSDL-Dokument fertig bearbeitet haben und den gewünschten Funktionsumfang unseres Web Services beschrieben haben, können wir uns daran machen, die Hauptfunktion von Apache Muse einzusetzen.
Apache Muse bietet, wie weiter oben bereits erläutert, mit Hilfe des Tools wsdl2java, Java Code aus einem WSDL-Dokument zu generieren.
Bevor wir jedoch mit wsdl2java arbeiten, müssen sämtliche xsd-Dokumente, auf die das WSDL-File referenziert in unser workDir/FoPra/wsdl Verzeichnis kopiert werden. Diese Dokumente findet man entweder direkt bei OASIS bei den WSDM Spezifikationen oder aber auch querverteilt in den Unterordnern des sample-Verzeichnises von Muse. Sollte man wsdl2java ohne die notwendigen Dokumente aufrufen, so erscheint eine Meldung, welches Dokument noch fehlt. Allerdings wird immer nur das nächstfehlende genannt, so dass es sehr zeitaufwendig sein kann, diese einzeln zusammenzusuchen.
Aber wenn wir alle Dokumente nun im workDir/FoPra/wsdl Ordner besitzen, können wir zurück in unser workdir/FoPra Verzeichnis wechseln und dort folgenden Befehl aufrufen:

```
wsdl2java -j2ee axis2 -wsdl .\wsdl\fopra.wsdl
```

Wir generieren also ein Axis2 Projekt, das auf einer J2EE Umgebung basiert. Dabei wird in unserem workDir/FoPra folgende Verzeichnisstruktur angelegt:

- /JavaSource
  In diesem Ordner sind die Java Quelltexte enthalten, die zu den benutzerspezifischen Capabilities und deren Funktionen und Properties gehören. Dabei werden zu jeder Capability zwei Klassen angelegt, die sich in einem Package befinden, das genauso heißt, wie der zugehörige Namespace URI. Zu jeder Capability werden zwei Klassen angelegt. Nämlich IMyCapability.java, ein Interface, das von MyCapability.java implementiert wird. Hierin befindet sich die Logik dieser Capability.

- /WebContent
  Dieser Ordner beinhaltet das Web Archiv, das wir nachher in einen J2EE-Server deployen werden. Darin enthalten sind zugehörige Muse-spezifische Klassen sowie WSDL-Dokumente, router-entries und muse.xml, das zur Konfiguration dient.

- build.xml
  Dies ist das Ant-Build-File, das die Java-Sources aus /JavaSource kompiliert und in das /WebContent/WEB-INF/lib Verzeichnis kopiert. Anschließend packt es das gesamte /WebContent Verzeichnis in ein WAR File, das in einen J2EE Server deployt werden kann.

- /.overwrite
  Diese Datei dient dazu, die Dateien zu markieren, die von wsdl2java überschrieben werden dürfen, falls es nach einer Änderung des WSDL-Dokuments nochmals aufgerufen wird. Denn hierbei sollen natürlich die eventuell bereits abgeänderten Java Sources nicht überschrieben werden.

5.3 Implementierung der Java Klassen

5.3.1 Allgemeine Beschreibung der Java Klassen

Bis jetzt wurde das allgemeine Konstrukt generiert, nun fehlt natürlich noch die logische Implementierung der Java Klassen. Dazu schauen wir uns im workDir/FoPra/JavaSources Verzeichnis genauer um. Hier finden wir einen Package-Pfad und am Ende dessen die Java Klassen, die für unser Projekt vonnöten sind. Wie oben bereits erläutert gehört zu jeder Capability eine Java Klasse MyCapability.java und ein dazugehöriges Interface IMyCapability.java. Wir müssen nun jeder Klasse die von uns gewünschte Logik hinzufügen. Am besten eignet sich zur Bearbeitung der Java-Dateien Eclipse, in dem das Package komplett importiert wird, oder aber auch jeder andere Editor.
Werfen wir nun also einen Blick auf MyCapability.java. Dort sehen wir natürlich zuerst, dass diese Klasse von AbstractWsResourceCapability erbt, also auch auf deren Methoden zugreifen kann. Doch betrachten wir die hier implementierten Methoden genauer. Unsere Klasse besitzt bereits die Methode initialize(). Dies ist eine der vier wichtigsten Methoden jeder Capability.

- Initialisierung: Die inititialize() Methode wird bei jeder Initialisierung der Capability aufgerufen. Hier sollten also alle notwendigen Schritte aussgeführt werden, damit die Resource die Capability verwenden kann.

- Post-Initialisierung: Die Methode initializeCompleted() wird aufgerufen, sobald alle anderen Capabilities ebenfalls die initialize() Methode ausgeführt haben. Unsere Capability kann sich also ab Aufruf von initializeCompleted() sicher sein, dass sie bereits auf andere Capabilites zugreifen kann. Nach Ausführen von initializeCompleted() ist die Capability fertig einsatzbereit.

- Pre-Shutdown: Wird die Methode prepareShutdown() aufgerufen, hat die Capability die letzte Möglichkeit, auf andere Capabilities zuzugreifen. Es ist sozusagen der last call, bevor andere Capabilites den stabilen und zugriffsbereiten Zustand verlassen und eventuell nicht mehr erreichbar sind.

- Shutdown: Die ultimativ zuletzt aufgerufene Methode shutdown() lässt die Capability alle Shutdown-Tasks ausführen, die sie selbst noch benötigt. Hier kann sie nicht mehr auf andere Capabilities zugreifen. Nach Durchführen dieser Methode gilt die Capability als beendet.

Diese vier Methoden müssen nicht zwangsweise von jeder Capability implementiert werden, da sie, falls sie nicht implementiert sind, einfach auf der Oberklasse aufgerufen werden. Sollte jedoch eine explizite Implementierung gewünscht sein (wie in unserem Fall bei initialize(), so muss (!) zuerst die jeweilige Methode der Oberklasse mittels super.xxx() aufgerufen werden. Außerdem sehen wir in unserer Java-Klasse alle Properties als private Variablen sowie dazugehörige Setter und Getter Methoden. Die von uns explizit definierten Funktionen Start und Stop (wir erinnern uns an das WSDL-File) werden in der Java-Klasse als Methoden implementiert, die jedoch bis jetzt nur RuntimeException werfen und noch keine Logik besitzen. Erst hier beginnt nun die eigentliche Implementierung unseres Funktionsumfangs. Bis jetzt haben wir nur definiert, was unser Web Service am Ende können soll, und wie auf dieses Können zugegriffen werden kann. Nun beginnen wir mit der tatsächlichen Logik. Und zwar

überlegen wir uns zuerst die grundlegende Funktion unseres Web Services.
Wir starten damit, dass wir das Werfen einer Exception bei der initialize() Methode auskommentieren. Ab jetzt könnte dieser Web Service bereits eingesetzt werden. Jedoch würde er per WSDL-File Funktionen definieren, die noch keinerlei Funktion haben, außer dass sie existent sind.

5.3.2 Entwicklung der grundlegenden Funktionalität

Bevor wir jedoch die Java-Klasse weiterentwickeln, müssen wir noch eine Kleinigkeit in einer anderen Datei ergänzen. Unser Web Service muss natürlich wissen, in welchem Verzeichnis der zu managende HTTP Web Server installiert ist. Dazu bearbeiten wir die Datei workDir/FoPra/WebContent/WEB-INF/classes/muse.xml. Darin suchen wir das <capability> Element, das unsere benutzerspezifische Capability repräsentiert, also in der Description auf MyCapability verweist aus der <capability>-Liste. Als letztes Kindelement fügen wir diesem Element folgende Kinder hinzu:

```
<init-param>
 <param-name>installDir_Windwows</param-name>
 <param-value>C:\\Programme\\apache-2.2</param-value>
 <param-name>installDir_Linux</param-name>
 <param-value>/usr/local/apache2</param-value>
</init-param>
```

Dabei ist natürlich zu beachten, dass die richtigen Installationspfade angegeben werden. Unter Windows ist darüber hinaus noch unbedingt zu beachten, dass die Backslashes \ escaped werden müssen durch einen weiteren Backslash. Diese beiden Parameter sind die einzigen, die unsere WS Resource für diese Capability aus externen Angaben benötigt. Alles andere wird während der Laufzeit initialisiert.
Doch nun zurück zu unserer MyCapability.java Datei. Alle weiteren Funktionalitäten implementieren wir darin.
Natürlich muss als erstes bei der Initialisierung, nach dem Aufruf von super.initialize() auf der Oberklasse, der zum Betriebssystem gehörige Wert des Installationsverzeichnises des HTTP Web Servers, den wir gerade in der muse.xml Datei definiert haben, in den Web Service eingelesen werden. Damit wird sichergestellt, dass das Installationsverzeichnis unbedingt zu jeder Zeit bekannt ist.

```
if (isWindows()) {
 setInstallDir(getInitializationParameter("installDir_Windows"));
}
else {
 setInstallDir(getInitializationParameter("installDir_Linux"));
}
```

Damit haben wir nun das Installationsverzeichnis festgelegt durch Einlesen des jeweiligen Wertes aus der muse.xml-Datei und setzen mittels der setInstallDir-Methode, die - wie viele andere Methoden auch - von wsdl2java in unserer Java-Skeleton-Klasse angelegt wurde.
Da unsere Resource sowohl unter Windows, als und Linux funktionieren soll, müssen wir

viele Befehle individualisieren, wie auch gerade schon das Konfigurieren des Installations-Verzeichnises. Um je nach Betriebssystem den richtigen Befehl auszuwählen, implementieren wir zunächste eine Methode, die uns unter Windows true zurückliefert, ansonsten false. Diese Methode können wir zum Beispiel ganz unten in unserer MyCapability-Klasse definieren.

```
private boolean isWindows() {
 if (System.getProperty("os.name").toLowerCase().contains("windows")) {
  return true;
 }
 else {
  return false;
 }
}
```

Wir erinnern uns, dass unser Web Service einen Apache HTTP Web Server managen soll und insbesondere diesen starten und stoppen soll. Daher ist es natürlich notwendig, dass wir uns zeurst um diese beiden Methoden kümmern. Wir implementieren also die start() Methode wie folgt:

```
public void start() throws StartFailedFault {
 try {
  Runtime.getRuntime().exec(getInstallDir() + "/bin/httpd -k start");
  shouldBeUp = true;
  getLog().info("Start OK.");
 }
 catch (IOException error) {
  throw new StartFailedFault("Start Error.");
 }
}
```

Durch Aufruf dieser Methode wird nun der Apache HTTP Web Server mit allen zuvor gesetzten Variablen/Properties als Dienst gestartet.
Als nächstes programmieren wir die stop() Methode analog dazu wie folgt

```
public void stop() throws StopFailedFault {
 try {
  Runtime.getRuntime().exec(getInstallDir() + "/bin/httpd -k stop");
  shouldBeUp = false;
  getLog().info("Stop OK.");
 }
 catch (IOException error) {
  throw new StopFailedFault("Stop Error.");
 }
}
```

Hiermit ist es möglich den Dienst zu beenden und den Apache HTTP Server sauber herunterzufahren. Natürlich müssen wir noch als globale Variable shouldBeUp deklarieren und initialisieren in der Nähe der anderen globalen Variablen. Diese Variable wird nur je nach Aufruf von start und stop gesetzt. Konkret verwendet wird sie erst später, wenn wir uns um WSN kümmern.

```
private boolean shouldBeUp = false;
```

Wir verwenden also die Java-interne Runtime Umgebung, um einen Befehl auf der Konsole auszugeben. Sollte dieser fehlschlagen, wird die jeweilige benutzerspezifische Exception geworfen.
Und genau diese benutzerspezifischen Exceptions müssen wir nun als Java Klassen implementieren. Da wir StopFailedFault und StartFailedFault verwenden, müssen wir zwei neue Java Klassen anlegen. Diese müssen im selben Verzeichnis liegen, wie auch unsere MyCapability.java.
Da diese benutzerspezifischen Faults jeweils von org.apache.muse.ws.resource.basefaults.BaseFault erben müssen, ergibt sich Stück für Stück folgende Implementierung.

```
package org.apache.ws.muse.fopra.httpd;
import javax.xml.namespace.QName;
import org.w3c.dom.Element;
import org.apache.muse.ws.resource.basefaults.BaseFault;
public class StartFailedFault extends BaseFault
{
 public static final QName START_FAILED_QNAME =
                 new QName(MyCapability.NAMESPACE_URI,
                     "StartFailedFault", MyCapability.PREFIX);
 public StartFailedFault(Element arg0)
 {
  super(arg0);
 }
 public StartFailedFault(String arg1)
 {
  super(START_FAILED_QNAME, arg1);
 }
 public StartFailedFault(String arg1, Throwable arg2)
 {
  super(START_FAILED_QNAME, arg1, arg2);
 }
 public StartFailedFault(Throwable arg1)
 {
  super(START_FAILED_QNAME, arg1);
 }
}
```

Natürlich wird analog dazu auch eine Klasse für das StopFailedFault benötigt.
Man sieht hier deutlich, woran die Muse Entwickler für die Zukunft noch arbeiten müssen: Warum kann nicht auch automatisch per wsdl2java die Java-Klassen der Exceptions (also hier Faults) angelegt werden? Mehr als eine Anpassung an die super-Klasse ist ja nicht zu machen.
Damit diese beiden Faults nun von unserere Hauptklasse geworfen werden können, müssen sie natürlich von dieser noch importiert werden:

```
import org.apache.ws.muse.fopra.httpd.StartFailedFault;
import org.apache.ws.muse.fopra.httpd.StopFailedFault;
```

Hiermit ist nun die Implementierung der Funktionalität des Startens und Stoppens abgeschlossen.

5.3.3 Entwicklung der Methoden zur Verwaltung der Properties

Damit wir die httpd.conf Datei verwalten können und in dieser Konfigurationsdatei die von uns gemanageten Werte anpassen können, brauchen wir natürlich Zugriff auf diese. Und natürlich wollen wir bereits bei Initialisierung unseres Web Services die Datei einlesen. Wir benötigen dazu aber primär die Möglichkeit, die httpd.conf Datei zu lesen und die dort eingetragenen Werte zu interpretieren. Also entwickeln wir eine Methode, die die komplette Datei einliest und alle Kommentare und sonstige unnützen Sachen entfernt, um danach alle Variablen und deren Werte in einer HashMap abzulegen. Folgende Methode fügen wir unserer MyCapability.java Datei hinzu:

```
private Map getConf(String file) throws SoapFault {
 BufferedReader reader;
 try {
  reader = new BufferedReader(new FileReader(file));
 }
 catch (Exception error) {
  throw new SoapFault("Error reading Config-File!", error);
 }

 String line;
 Map conf = new HashMap();
 try {
  while ((line = reader.readLine()) != null) {
   line = line.trim();
   if (line.length() == 0 || line.charAt(0) == '#' || line.charAt(0) == '<') {
    continue;
   }
   int space = line.indexOf(" ");
   String name = line.substring(0, space);
   String value = line.substring(space+1);
   conf.put(name,value);
   reader.close();
  }
 }
 catch (Exception error) {
  throw new SoapFault("Error reading Config-File!", error);
 }
 return conf;
}
```

Diese Methode legt nun eine Hashmap mit allen Variablennamen und deren Werte an Hand der httpd.conf Datei an und liefert diese Map zurück. Natürlich muss diese Methode nun gleich bei Initialisierung der Capability aufgerufen werden. Daher erweitern wir unsere initialize()-Methode mit folgendem Code:

```
Map conf;
try {
 if (isWindows()) {
  conf = getConf(_installDir + "\\conf\\httpd.conf");
 }
 else {
  conf = getConf(_installDir + "/conf/httpd.conf");
 }
}
catch (IOException error) {
throw new SoapFault("Error reading Config-File!", error);
}
```

Nun können wir je nach Betriebssystem mit unterschiedlichen Mitteln auf die Map zugreifen und dort direkt unseren ServerNamen und den Port auslesen und im Web Service in unserer Capability setzen.

```
_ServerName = ((String)conf.get("ServerName"));
_Port = (Integer.valueOf((String)conf.get("Listen")));
```

Damit unsere neue Methode funktioniert, müssen wir noch folgende Sachen importieren:

```
import java.util.Map;
import java.util.HashMap;
import java.io.IOException;
import java.io.BufferedReader;
import java.io.FileReader;
```

Als nächstes steht nun an, dass wir natürlich auch die Werte in der httpd.conf Datei ändern können, sollten sie mittels SetResourceProperties Methoden in der Resource geändert werden. Dazu entwickeln wir eine weitere Methode edit, die folgendermaßen aufgebaut ist:

```
private boolean edit(String name, String value) {
 File file;
 File file_new;
 if (isWindows()) {
  file = new File(_installDir + "\\conf\\httpd.conf");
  file_new = new File(_installDir + "\\conf\\httpd_new.conf");
 }
 else {
  file = new File(_installDir + "/conf/httpd.conf");
  file_new = new File(_installDir + "/conf/httpd_new.conf");
 }
 BufferedReader reader = null;
 FileWriter fwriter = null;
 FileReader freader = null;
 String line = null;
 try {
  freader = new FileReader(file);
```

```
  fwriter = new FileWriter(file_new);
  reader = new BufferedReader(freader);
 }
 catch (Exception error) {
  getLog().info("Editing httpd.conf failed!");
 }
 boolean found = false;
 while(true) {
  try {
   line = reader.readLine();
  }
  catch (Exception error) {
  getLog().info("Editing httpd.conf failed!");
 }
 if (line == null) {
  break;
 }
 if(line.startsWith(name)) {
   try {
    fwriter.write(name + " " + value + "\n");
   }
   catch (Exception error) {
    getLog().info("Editing httpd.conf failed!");
   }
  }
  else  {
   try {
    fwriter.write(line.toString() + "\n");
   }
   catch (Exception error) {
    getLog().info("Editing httpd.conf failed!");
   }
   continue;
  }
 }
 try {
  reader.close();
  freader.close();
  file.delete();
  fwriter.close();
  file_new.renameTo(file);
 }
 catch (Exception error) {
  getLog().info("Editing httpd.conf failed!");
 }
 return found
}
```

Beim Aufruf dieser Methode wird die zu ändernde Variable und der neue Wert dieser Variable übergeben. Danach wird die httpd.conf Datei geöffnet und nach der Variable gesucht. Zeile für Zeile wird in eine neue temporäre Datei geschrieben. Dabei werden ungeänderte Zeilen übernommen, nur beim Auftreten der gesuchten Variable wird eine neue Zeile generiert. Am Ende wird die ursprüngliche httpd.conf gelöscht und durch die temporäre Datei ersetzt.
Damit der Code funktioniert müssen wieder weitere Pakete importiert werden:

```
import java.io.File;
import java.io.FileWriter;
```

In jeder Setter Methode muss als nächstes noch der Aufruf der edit(..) Methode implementiert werden. Dazu bearbeiten wir als erstes die setServerName Methode und erweitern sie um folgenden Code:

```
if (edit("ServerName", param0)) {
 _ServerName = param0;
}
```

Bei der setPort Methode wird ähnliches ergänzt:

```
if (edit("Listen", param0 + "")) {
 _Port = param0;
}
```

Nun können wir sowohl ServerName als auch den Port in der httpd.conf Datei mittels unserem Web Service auslesen und abändern. Unser Apache HTTP Web Server kann also nun bereits in gewissem Umfange gemanaget werden.
Eine Änderung der Properties, die den Server betreffen, können zwar nach dem Start immer noch im Web Service und somit auch in der httpd.conf geändert werden, jedoch treten diese selbstverständlich erst nach einem Stop und erneutem Start des Servers in Kraft.
Es ist uns möglich den Port und Servername aus der Konfiguration komplett zu verwalten. Ein Starten und Stoppen ist auch möglich. Wir verwenden also mitlerweile Methodiken aus dem WSRF sowie aus WSDM. Als nächstes steht noch eine weitere Funkionalität an, die wir implementieren wollen, nämlich ein Beispiel für WSN.
Unser WSDL-File hat dazu aus dem Template bereits die benötigten Methoden beschrieben und unser MyCapability.java erbt ja bekanntlich von AbstractWsResourceCapability, so dass wir uns nun daran machen können, uns die Funktionalität eines Producers zu überlegen.

5.3.4 Entwicklung der WSN Funktionalität

Unsere WS Resource kann also nun bereits im Rahmen des WSRF verwaltet werden und Management im Sinne des WSDM betreiben. Als nächstes wollen wir noch einen Subscribe-Publish Mechanismus innerhalb der WSN Spezifikation implementieren. Dazu müssen wir wieder unser MyCapability.java File bearbeiten und dort einige Sachen ergänzen.
Zuerst müssen wir natürlich die Topics initialisieren, die unsere WS Resource verwaltet. Dazu deklarieren und initialisieren wir eine weitere Variable pro Topic. Wir wollen einen Topic, der, wenn der zu managende HTTP Web Server abgestürzt ist, oder außerhalb des Web Services beendet wurde, im Takt des MessageIntervals, Nachrichten published und damit alle Subscriber über das unerwartete Herunterfahren benachrichtigt. Unser zweiter Topic dient zum Benachrichtigen im MessageInterval-Takt über die Anzahl der derzeitigen Verbindungen zum HTTP Web Server.

```
private static final QName _TOPIC_ServerDown = new QName("ServerDownTopic");
private static final QName _TOPIC_Connections = new QName("ConnectionsTopic");
```

Des weiteren ist es nötig, dass die Property MessageInterval grundsätzlich initialisiert ist. Wir setzen also in unsere initialize() Methode an die Stelle, wo wir auch die Properties Port und ServerName initialisieren zusätzlich folgendes Code-Schnipsel:

```
_MessageInterval = 10:
```

Wir setzen diesen Wert auf 10 Sekunden. Der Wert kann jederzeit, natürlich unabhängig davon, ob unser gemanageter Apache HTTP Web Server läuft oder nicht, mittels setResourceProperties Methodiken geändert werden. Jedoch dazu später mehr bei der Entwicklung des Testers.
Als nächstes implementieren wir nun die initializeCompleted() Methode, die, wie wir bereits wissen, aufgerufen wird, sobald die WS Resource davon ausgehen kann, dass alle anderen Capabilities auch initialisiert sind, und vor allem, sie selbst auch initialisiert ist.
Da ein Publish Mechanismus natürlich erst dann starten soll, wenn alle Initialisierungen abgeschlossen sind, bietet sich an, diese Funktionalität eben innerhalb der initializeCompleted() Methode zu starten.

```
public void initializeCompleted() throws SoapFault
{
  super.initializeCompleted();
}
```

Selbstverständlich muss wiederum als erstes die initializeCompleted() Methode der Oberklasse aufgerufen werden, bevor wir Capability-spezifische Funktionalitäten implementieren. Zuerst müssen wir nun alle WSN-spezifischen Klassen sowie weitere verwendete Klassen importieren.

```
import org.apache.muse.ws.notification.NotificationProducer;
import org.apache.muse.ws.notification.WsnConstants;
import org.apache.muse.util.xml.XmlUtils;
import org.w3c.dom.Element;
import java.io.InputStream;
```

Wir erweitern nun Schritt für Schritt unsere initializeCompleted() Methode und holen uns zuerst mittels

```
final NotificationProducer wsn =
     (NotificationProducer)getResource().getCapability(WsnConstants.PRODUCER_URI);
```

den WSN Notification Producer, mittels dem wir unsere Nachrichten verteilen wollen. Anschließend fügen wir unsere Topics diesem hinzu.

```
wsn.addTopic(_TOPIC_ServerDown);
wsn.addtopic(_TOPIC_Connections);
```

Nun haben wir schon das Grundgerüst für zwei Topics fertig erstellt. Man könnte sich bereits für diese beiden anmelden. Jedoch haben sie noch keine Funktion, so dass bis jetzt noch keine Nachrichten verteilt werden.

Da die beiden Topics jeweils unabhängig voneinander und vor allem unabhängig von anderen Funktionsaufrufen in der WS Resource sein sollen, bietet sich an, für jeden Topic einen Thread zu erstellen.
Wir entwerfen daher zuerst einen Thread, der sich um die Überprüfung des Serverstatus kümmert und im Falle, dass der Server down ist, ohne, dass wir ihn über unseren Web Service deaktiviert haben, Nachrichten im Abstand des MessageIntervals an alle Subscriber versendet.

```
Thread ServerCheckerThread = new Thread() {
 public void run() {
  QName messageName = new QName(NAMESPACE_URI, "ServerDownMessage", PREFIX);
  String command[];
  if (isWindwos())
   command = new String[]{"cmd",  "/c",
                "netstat -an | find /C /I \"0.0.0.0:" + getPort() + " \""};
  else
   command = new String[]{"/bin/sh",  "-c",
                "netstat -an | grep -c \":::" + getPort() + " \""};
  while (true) {
   try {
    int interval = getMessageInterval();
    Thread.currentThread().sleep(interval * 1000);
    Process proc =  Runtime.getRuntime().exec(command);
    InputStream input = proc.getInputStream();
    StringBuffer buffer = new StringBuffer();
    int pos;
    while ((pos = input.read()) != -1) {
     buffer.append((char)pos + "");
    }
    if (proc != null) proc.destroy();
    if (input != null) input.close();
    int listen = (Integer.valueOf(buffer.toString().trim()));
    if (listen == 0 && shouldBeUp == true) {
     String message = "Der Server " + getServerName() + " ist down!";
     Element payload = XmlUtils.createElement(messageName, message);
     wsn.publish(_TOPIC_ServerDown, payload);
    }
   }
   catch (Throwable error) {
    error.printStackTrace();
   }
  }
 }
};
```

Dieser Thread erstellt zuerst einen messageName, den wir später zum publishen der Nachricht benötigen. Danach wird ein Aufruf für die Java Runtime Umgebung definiert, der je

nach Windows oder Linux Umgebungen anders lauten muss, da wir eine andere Shell und andere Kommandozeilenfunktionen haben. Die eigentliche Funktionalität finden wir innerhalb der while Schleife, die sich darum kümmert, den Aufruf durchzuführen und die Rückgabe auszuwerten. So werden hier alle auf unserem konfigurierten Port lauschenden Sockets auf Localhost gezählt. Falls der Server keinen Socket mehr offen hat und zugleich die Variable shoudlBeUp (wir erinnern uns an weiter oben), auf true gesetzt ist, geht der Web Service davon aus, dass der Server ein Problem hat und generiert eine Nachricht, die anschließend per publish Methode an alle Subscriber verteilt wird.
Der zweite Thread kümmert sich um unseren zweiten Topic und sieht wie folgt aus:

```
Thread ConnectionsThread = new Thread() {
 public void run() {
  QName messageName = new QName(NAMESPACE_URI, "ConnectionsMessage", PREFIX);
  String[] command;
  String ip = "";
  try {
   ip = InetAddress.getLocalHost().getHostAddress();
  }
  catch (UnknownHostException error) {
   error.printStackTrace();
  }
  if (isWindows()) {
   command = new String[]{"cmd",  "/c",
                "netstat -an | find /C /I \"'"' + ip + ":" + getPort() + " \""};
  }
  else {
   command = new String[]{"/bin/sh",  "-c",
                "netstat -an | grep -c \"0 ::1:" + getPort() + " \""};
  }
  while (true) {
   try {
    int interval = getMessageInterval();
    Thread.currentThread().sleep(interval * 1000);
    Process proc =  Runtime.getRuntime().exec(command);
    InputStream input = proc.getInputStream();
    StringBuffer buffer = new StringBuffer();
    int pos;
    while ((pos = input.read()) != -1) {
     buffer.append((char)pos + "");
    }
    if (proc != null) proc.destroy();
    if (input != null) input.close();
    int connections = (Integer.valueOf(buffer.toString().trim()));
    if (connections == -1) connections = 0;
    String message = "Der Server " + getServerName() +
                     " hat aktuell " + connections + " Verbindungen.";
    Element payload = XmlUtils.createElement(messageName, message);
```

```
    wsn.publish(_TOPIC_Connections, payload);
   }
   catch (Throwable error) {
    error.printStackTrace();
   }
  }
 }
};
```

Wir definieren wieder zuerst einen messageName sowie ein aufzurufendes Kommando, je nach Betriebssystem verschieden. Die while Schleife kümmert sich wieder um den unendlichen Ablauf. Wir lesen diesesmal über die Konsole die aktuell auf unseren Rechner zu unserem konfigurierten Port verbundenen Gegenstellen aus und zählen diese. Das Ergebnis wird dann im Takt von messageInterval gepublishet an alle Subscriber.
Selbstverständlich fehlen wieder einige Pakete, die wir per Imports importieren:

```
import java.net.InedAddress;
import java.net.UnknownHostException;
```

Am ende unserere initializeCompleted() Methode müssen wir nun noch die beiden Threads starten:

```
ServerCheckerThread.start();
ConnectionsThread.start();
```

Damit sind nun beide Topics fertig implementiert und funktionsbereit. Um die Subscribe- und Publish-internen Mechanismen kümmert sich Muse.
Wir haben nun die Logik unseres Producer Web Services fertig implementiert. Er ist somit voll einsatzfähig.

5.4 Kompilierung des Projekts

Nun sind wir fertig mit der Erstellung unseres Projektes und können das Web Archiv generieren. Dies erfolgt einfach durch den Aufruf von *ant* in unserem workDir/FoPra Verzeichnis. Ant führt dabei die Befehle in der build.xml aus und erstellt am Ende nach erfolgreicher Kompilierung ein WAR-Archiv. Dieses beinhaltet nun unsere Muse-Umgebung mit den von uns entwickelten Capabillities innerhalb einer Axis2 Umgebung und hat daher eine stolze Größe von über 11 Megabyte.
Dieses Paket kann nun in jeder J2EE-Servlet Umgebung eingesetzt werden, jedoch machen wir das erst später und entwickeln zunächst noch unseren zweiten Web Service.

5.5 Entwicklung des WSDL-Dokuments für den WSN-Consumer

Als nächstes müssen wir nun einen Web Service entwickeln, an den Messages per WSN gepublishet werden können. Dieser Web Service muss also nichts anderes können, als die Nachrichten annehmen und interpretieren. Folgendes simples WSDL-Dokument beschreibt den benötigten Umfang:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="http://ws.apache.org/muse/fopra/wsnconsumer"
 xmlns:tns="http://ws.apache.org/muse/fopra/wsnconsumer"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:wsa="http://www.w3.org/2005/08/addressing"
 xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
 xmlns:wsdl-soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:wsnt="http://docs.oasis-open.org/wsn/b-2"
 xmlns:wsntw="http://docs.oasis-open.org/wsn/bw-2"
 xmlns:wsrf-r="http://docs.oasis-open.org/wsrf/r-2"
 name="WSNConsumer">
 <wsdl:types>
  <xsd:schema elementFormDefault="qualified"
       targetNamespace="http://www.w3.org/2005/08/addressing">
   <xsd:include schemaLocation="WS-Addressing-2005_08.xsd" />
  </xsd:schema>
  <xsd:schema elementFormDefault="qualified"
       targetNamespace="http://docs.oasis-open.org/wsn/b-2">
   <xsd:include schemaLocation="WS-BaseNotification-1_3.xsd" />
  </xsd:schema>
  <xsd:schema elementFormDefault="qualified"
       targetNamespace="http://docs.oasis-open.org/wsrf/r-2">
   <xsd:include schemaLocation="WS-Resource-1_2.xsd" />
 </xsd:schema>
 </wsdl:types>

 <wsdl:message name="Notify">
  <wsdl:part name="Notify" element="wsnt:Notify"/>
 </wsdl:message>

 <wsdl:portType name="WSNConsumerPortType">
  <wsdl:operation name="Notify">
  <wsdl:input
    wsa:Action="http://docs.oasis-open.org/wsn/bw-2/NotificationConsumer/NotifyRequest"
        message="tns:Notify" />
  </wsdl:operation>
 </wsdl:portType>

 <wsdl:binding name="WSNConsumerBinding" type="tns:WSNConsumerPortType">
  <wsdl-soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <wsdl:operation name="Notify">
   <wsdl-soap:operation soapAction="Notify" />
   <wsdl:input>
    <wsdl-soap:body use="encoded"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
```

```
   </wsdl:input>
  </wsdl:operation>
 </wsdl:binding>

 <wsdl:service name="WSNConsumer">
  <wsdl:port name="WSNConsumerPortType" binding="tns:WSNConsumerBinding">
   <wsdl-soap:address location="http://localhost:8080/consumer/services/wsnconsumer" />
  </wsdl:port>
 </wsdl:service>
</wsdl:definitions>
```

Dieses WSDL-Dokument basiert nicht auf dem Muse-Template, da es im Prinzip keinen Funktionsumfang benötigt. Wir setzen direkt auf Muse beiliegenden XSD-Schemata, die die jeweiligen Spezifikationen definieren. Wir sehen anhand des types-Elements, dass dieser Web Service nur auf WS-Addressing, WS-Resource und WS-BaseNotifiaction setzt, also genau den Funktionsumfang, den dieser kleine Web Service am Ende beherrschen soll. Die einzige Operation, die er unterstützt ist Notify, was genau dann von einem anderen Web Service aufgerufen wird, wenn eine Nachricht eingeht. Wir sehen, dass die Notify-Message auf ein Element im wsnt-Namespace verlinkt ist, also direkt von der Base-Notifiaction Spezifikation abgedeckt werden kann.
Erreichbar ist der Web Service unter der URI http://localhost:8080/consumer/services/wsnconsumer.
Dieses WSDL-Dokument speichern wir ab unter workdir/Consumer/wsdl/Consumer.wsdl.

5.6 Generierung und Entwicklung der Java-Klassen

Nachdem wir nun unseren zweiten Web Service beschrieben haben, wechseln wir in das Verzeichnis workDir/Consumer und lassen Muse wieder seine Hauptarbeit erledigen, indem wir wsdl2java mit folgenden Parametern aufrufen:

```
wsdl2java -j2ee axis2 -wsdl ./wsdl/Consumer.wsdl
```

Diesmal generiert wsdl2java keinen JavaSource Ordner, sondern nur den bereits bekannten WebContent Ordner sowie die .overwrite und build.xml Dateien. Das liegt daran, dass wir im WSDL-File nichts definieren, was nicht bereits durch inkludierte Schemata abgeglichen wäre, da unser Web Service nur primitiv gepublishete Nachrichten empfangen soll. Wir wollen jedoch dennoch eine benutzerspezifische Capability implementieren, um zu zeigen, dass die SOAP-Nachrichten, die der Service empfängt auch korrekt ausgewertet werden können. Dazu müssen wir per Hand die Datei MaCapability.java im von uns zu erstellenden Verzeichnis workDir/Consumer/JavaSource/org/apache/ws/muse/fora/wsnconsumer/ erstellen. Die Datei an sich ist dabei dann von der Grundstuktur ähnlich aufgebaut, wie auch unsere benutzerspezifische Capability des ersten Web Services. Wir erstellen also folgende Datei:

```
package org.apache.ws.muse.fopra.wsnconsumer;

import java.util.Date;
import org.apache.muse.core.AbstractCapability;
import org.apache.muse.ws.addressing.soap.SoapFault;
import org.apache.muse.ws.notification.NotificationConsumer;
```

```
import org.apache.muse.ws.notification.NotificationMessage;
import org.apache.muse.ws.notification.NotificationMessageListener;
import org.apache.muse.ws.notification.WsnConstants;

public class MyCapability extends AbstractCapability
                          implements NotificationMessageListener
{
 private GUI gui;
 public void initialize() throws SoapFault {
  super.initialize();
  gui = new GUI();
 }

 public void initializeCompleted() throws SoapFault {
  super.initializeCompleted();
  NotificationConsumer wsn =
     (NotificationConsumer)getResource().getCapability(WsnConstants.CONSUMER_URI);
  wsn.addMessageListener(this);
 }

 public boolean accepts(NotificationMessage message) {
  return true;
 }

 public void process(NotificationMessage message) {
  String from = message.getProducerReference().getAddress().toString();
  String topic = message.getTopic().toString();
  String what =
    message.toString().split("<wsnt:Message>")[1].split("\">")[1].split("<")[0];
  gui.tell("Received Message from: " + from + "\nTime: " + new Date() +
           "\nTopic: " + topic + "\nMessage: " + what + "\n");
 }
}
```

Die Klasse erbt von AbstractCapaility und implementiert das Interface NotificationMessageListener. Während der Initialisierung der Capability wollen wir auch ein kleines GUI erstellen, auf dem die erhaltenen Nachrichten grafisch aufbereitet dargestellt werden. Diese Klasse müssen wir selbstverständlich auch erstellen. Doch dazu später mehr. Als nächstes benötigen wir die initializeCompleted()-Methode, in der wir einen WSN Consumer als uns selbst definieren und daran einen MessageListener in Form der Benutzerspezifischen Capability hinzufügen.
Da wir das NotificationMessageListener Interface implementieren, müssen wir die beiden dort definierten Methoden acceppts und process konkret implementieren. accepts dient dabei nur der Überprüfung, ob unsere Capability überhaupt Nachrichten entgegennimmt. Interessanter hingegen ist die process Methode. Diese Methode wird aufgerufen, sobald eine Nachricht empfangen wurde. Hier drin befindet sich also die eigentliche Logik, die der Web Service

ausführen soll, wenn er eine Nachricht aus einem Subscribe-Publish-Mechanismus erhält. Wir teilen die erhaltene Nachricht dazu in den Absender, den Topic und die eigentliche Nachricht auf. Diese drei Sachen geben wir dann mittels der tell-Methode, die unser GUI innehat, auf dem GUI aus, so dass wir einfach die erhaltenen Nachrichten mitlesen können. Die GUI-Klasse sieht dabei wie folgt aus:

```
package org.apache.ws.muse.fopra.wsnconsumer;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.TextArea;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class GUI extends JFrame {

 private JPanel infopanel;
 private JPanel outputpanel;
 private JLabel desc;
 private TextArea ret;

 public GUI() {
  super("Consumer Watcher");
  setMinimumSize(new Dimension(600,400));
  setResizable(true);
  setSize(600,400);
  setLocation(400,200);
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  outputpanel = new JPanel(new BorderLayout(2,1));
  desc = new JLabel("Output:");
  ret = new TextArea("");
  outputpanel.add(BorderLayout.NORTH, desc);
  outputpanel.add(ret);
  ret.setEditable(false);
  getContentPane().add(BorderLayout.CENTER, outputpanel);
  pack();
  setVisible(true);
 }
 public void tell(String what) {
  ret.insert(what + "\n", ret.getText().length());
 }
}
```

Diese einfach Klasse erzeugt ledilich ein kleines Fenster, das folgendermaßen aussieht: Mittels der tell-Methode wird das Ausgabe-Fenster weiter befüllt.

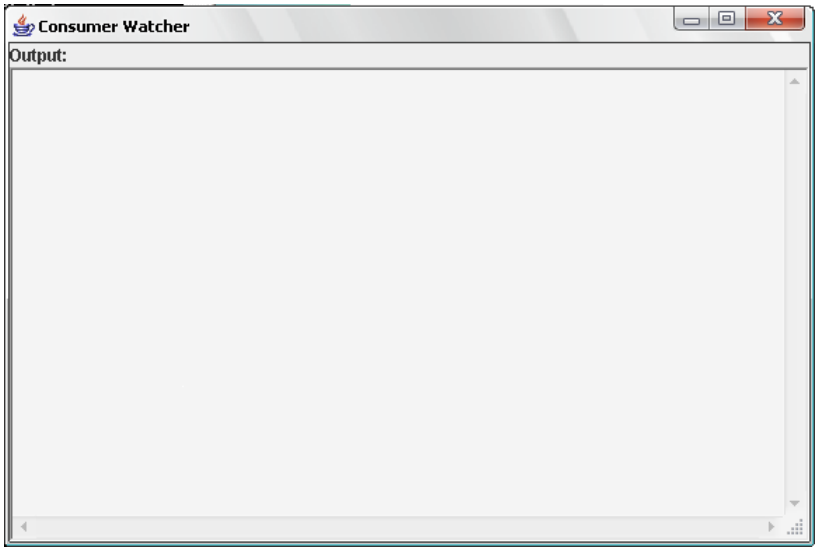

Abbildung 5.1: Consumer GUI

Da wsdl2java leider keine MyCapability angelegt hat, ist es notwendig, die Capability per Hand der Verwaltung durch Muse hinzuzufügen. Dazu müssen wir die Datei workDir/Consumer/WebContent/WEB-INF/classes/muse.xml bearbeiten und unsere benutzerspezifische Capability unter der bereits vorhandenen NotificationConsumer Capability, die Muse für jeden NoticicationConsumer-Service implementiert, hinzufügen mit folgendem Code:

```
<desc:capability xmlns:desc="http://ws.apache.org/muse/descriptor">
 <desc:capability-uri>
   http://ws.apache.org/muse/fopra/wsnconsumer
 </desc:capability-uri>
 <desc:java-capability-class>
   org.apache.ws.muse.fopra.wsnconsumer.MyCapability
 </desc:java-capability-class>
</desc:capability>
```

5.7 Kompilierung des WSN-Consumers

Nun haben wir die Arbeit an diesem Web Service bereits abgeschlossen und können mittels dem Aufruf von *ant* im workDir/Consumer Verzeichnis das WAR-File generieren, das die gewünschte Consumer.war Datei anlegt, mit der wir folgenden Unterkapitel weiterarbeiten.

5.8 Deployment in eine Tomcat Umgebung

Im Installationsverzeichnis von Tomcat finden wie den Unterodner webapps. Dorthin kopieren wir beide zuvor erstellten WAR Archive, also FoPra.war und Consumer.war. Tomcat wird

dann mit Hilfe der startup.bat/sh im /bin Ordner gestartet und entpackt automatisch das WAR-Archiv und führt das Deployment selbstständig durch. Auch unterstützt Tomcat Hot-Deployment. Das bedeutet, WAR Archive können, wenn niemand darauf zugreift, während des Betriebes entfernt oder hinzugefügt werden. Dies geht meist gut, doch hin und wieder kommt es dabei auch zu einem Fehler. So werden beispielsweise beim Entfernen eines WAR Archives die dazugehörigen Dateien nicht alle gelöscht oder aber der Zugriff auf Capabilities eines neuen WAR Archives erfolgt nicht korrekt, sondern es gibt Runtime Exceptions. Dies lässt sich aber durch einen Neustart des Tomcats beheben. Somit empfiehlt sich generell auf Hot Deployment zu verzichten falls möglich, und lieber per shutdown.sh/bat und startup.sh/bat den Server zu beenden und neuzustarten und gegebenenfalls die WAR-Archive und die dazugehörigen deployten Dateien vor dem erneuten Starten manuell zu löschen.
Wir können nun überprüfen ob unser Deployment erfolgreich war, ob also Tomcat das Muse/Axis2 Projekt korrekt verarbeiten konnte. Dazu überprüfen wir zuerst, ob die beiden Services korrekt installiert wurden, indem wir einen beliebigen Broswer starten und uns dort auf http://localhost:8080 beziehungsweise anstatt localhost die IP, auf dem Tomcat läuft verbinden. Port 8080 ist dabei der Standard-Port von Tomcat, der jedoch in der Konfigurationsdatei auch geändert werden kann. Dabei gilt eventuell die Firewall anzupassen. Es sollte nun die Apache Tomcat Oberfläche erscheinen.

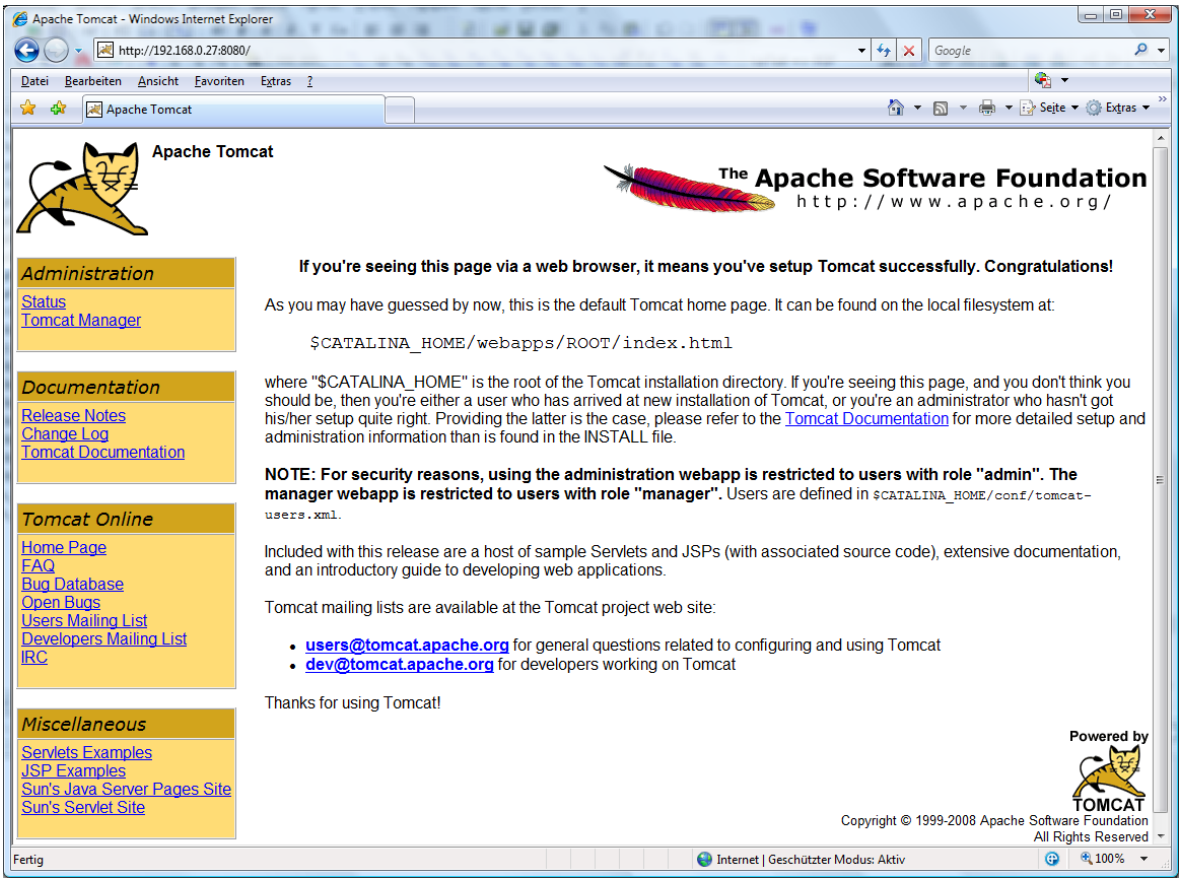

Abbildung 5.2: Tomcat Web-Oberfläche

Mit einem Klick auf „Tomcat Manager“ und Eingabe des Manager-Passworts, das wir zu Beginn unserer Arbeit festgelegt haben, gelangen wir zur Übersicht der aktuell installierten Projekte. Dort sind neben einigen Standard-Tomcat-Tools auch unsere beiden Web Services gelistet. Der Haupt Web Service nennt sich FoPra, der kleine WSN-Consumer-Web-Service nennt sich einfach nur Consumer. Durch einen einfachen Klick darauf wird die Axis2 Umgebung gestartet und wir können nun wieder zwischen mehreren Optionen wählen. Mit einem Klick auf „Services“ sehen wir die Dienste, die von unseren Web Services angeboten werden, sowie deren EPRs.

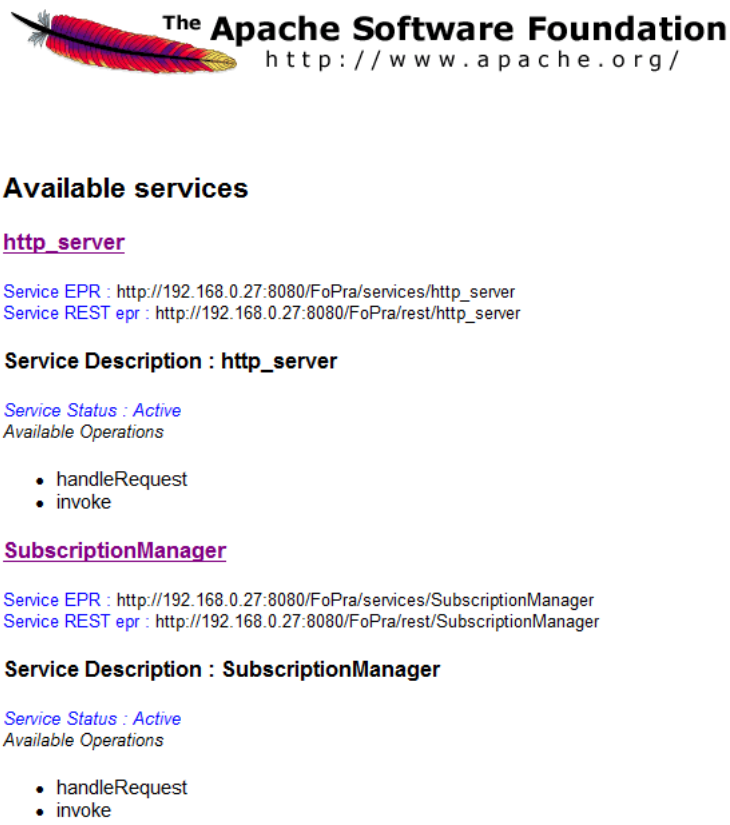

Abbildung 5.3: Tomcat FoPra Web Service: angebotene Dienste

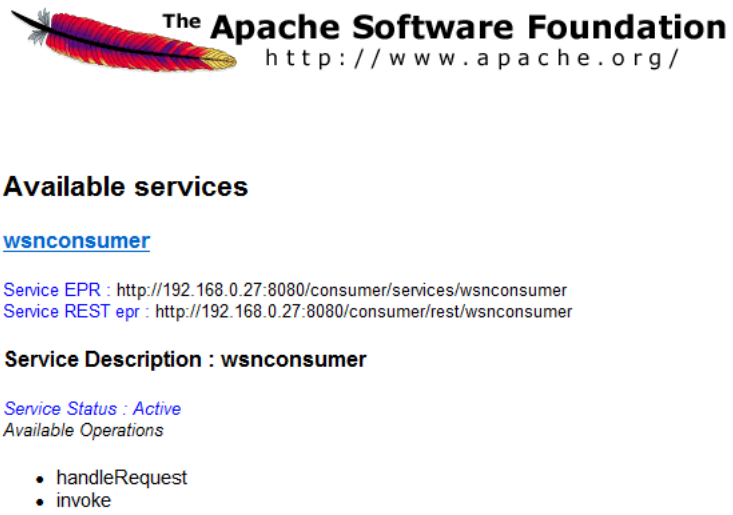

Abbildung 5.4: Tomcat Consumer Web Service: angebotene Dienste

Wir sehen also, dass der FoPra Web Service sowohl einen Dienst bietet, der sich „http_server“ nennt, als auch einen Dienst, der „SubscriptionManager“ heißt. Ersterer verwaltet unseren Apache HTTP Web Server, zweiterer dient der Verwaltung der Subscriptions im Sinne von WSN.
Unser Consumer Web Service bietet den Dienst wsnconsumer, der uns als WSN-Consumer dient zur Nachrichten-Entgegennahme.
Mit einem Klick auf „Validate“ können wir noch bei jedem Web Service überprüfen, ob die Axis2 Bibliotheken vorhanden sind. Jedoch ist das Validieren nicht sehr aussagekräftig, da nur auf das Vorhandensein der Bibliotheken geprüft wird und keinerlei Funktion. Außerdem erhalten wir dort noch viele Informationen zur verwendeten Java-Runtime, mit der Tomcat arbeitet.
Wir haben nun einen kleinen Blick auf die Web-Oberfläche von Apache Tomcat geworfen und gesehen, dass beide von uns entwickelten Web Services fertig deployed und einsatzbereit sind.

5.9 Entwicklung des Web Service Testers

Als nächstes wollen wir natürlich die deployten Web Services testen. Diese unterhalten sich untereinander und mit einem sogenannten Client-Proxy mittels SOAP-Nachrichten. Apache Muse bietet uns die Möglichkeit mittels wsdl2java eine simple Klassenstruktur zu erstellen, die Java-Methoden-Aufrufe in SOAP-Nachrichten konvertiert und diese an den zugehörigen Web Service schickt. Diese Java-Anwendung wird von Muse Client-Proxy genannt.
Wir erstellen dazu zunächst einen neuen Ordner in unserem workDir namens client. Darin rufen wir folgenden Befehl auf:

```
wsdl2java -proxy -wsdl ../FoPra/wsdl/fopra.wsdl
```

Damit wird folgende Ordnerstruktur erstellt:

- /src
  Enthält den Quellcode unseres Client-Packages.
- /lib
  Enthält Jar-Archive, die von Muse implementiert sind.
- build.xml
  Ein Ant-Script zu Kompilierung.
- .overwrite
  Die obligatorische .overwrite Datei, die alle Dateien gelistet hat, die nicht per wsdl2java überschrieben werden dürfen.

Für uns interessant ist nun natürlich der /src Ordner und die darin enthaltenen Klassen und Interfaces.
Am Ende des Package Pfades sehen wir eine Java Klasse namens http_serverProxy, die das Interface http_server implementiert. Wir implementieren nun im selben Ordern, wo auch http_serverProxy.java und http_server.java liegen eine neue Java Klasse namens tester.java. Diese Klasse soll eine main-Methode enthalten, so dass sie auch ausführbar ist und uns dann die Möglichkeit liefert, mit unserem FoPra-Web Service zu kommunizieren, der dann den Apache HTTP Web Server managet, wie wir oben implementiert haben, sowie Subscriptions entgegennimmt.
Da der Umfang des Testers sehr groß ist, was aber vor allem daraus resultiert, dass der Code, der nötig zur GUI-Erstellung ist, sehr umfangreich ist, wird im Folgenden nur auf die relevanten Teile eingegangen. In der Main-Methode wird ledlich eine Instanz von sich selbst erzeugt. Der Konstruktor kümmert sich dann um den Aufbau des gesamten GUI. So werden hier alle Panels und Buttons angelegt. Die Buttons bekommen alle actionListeners zugewiesen, die dann die Methode perform aufrufen mit Buttonspezifischen Parametern. Die Methode perform ist dafür zuständig, dass wir mit unserem Web Service kommunizieren können. Je nach Button wird eine andere action übergeben, an Hand derer die perform Methode zwischen den Cases switched. Alle GUI-spezifischen Sachen wurden in folgendem Listing entfernt. Außerdem sind Variablendeklarationen nicht vorhanden, da diese bereits an anderer Stelle der Klasse durchgeführt wurden. Ebenso werden sinnbildhaft nur wenige Cases dargestellt, die anderen sind analog. Daher ist folgender Code nur als Pseudo Code anzusehen.

```
private void perform(int action, String arg) {
 switch (action) {
  case 1: //setup
   URI address;
   address = URI.create(uri.getText());
   epr = new EndpointReference(address);
   http = new http_serverProxy(epr);
   URI consumerAddress;
   consumerAddress = URI.create(uri_consumer.getText());
   epr_consumer = new EndpointReference(consumerAddress);
  break;
  case 2: //start
   try {
```

```
    http.start();
    tell("Start: OK");
   }
   catch (Throwable error)
   {
    tell("Start: Failed");
    err(error);
   }
  break;
  case 4: //getServerName
   try {
    tell("Servername: " + http.getServerName());
   }
   catch (Throwable error) {
    tell("get Servername: Failed");
    err(error);
   }
  break;
  case 5: //setServerName
   try {
    http.updateServerName(arg);
    tell("Updated Servername to: " + arg);
   }
   catch (Throwable error) {
    tell("Update Servername: Failed");
    err(error);
   }
  break;
  case 11: //subscribeConnections
   try {
    http.subscribe(epr_consumer, new TopicFilter(new QName("ConnectionsTopic")), null);
   }
   catch (Throwable error) {
    tell("Subscription: Failed");
    err(error);
   }
  break;
  default:
  break;
 }
}
```

Wir sehen also, dass wir beim Klick auf Setup ledlich die Instanzen erzeugen, die an die EPRs gebunden sind. Die Instanz unserers FoPra Web Service Proxys nennt sich http. Auf diesem Proxy werden nun alle Funktionen aufgerufen, wie zum Beispiel http.start() beim Klick auf den Start-Button oder http.subscribe(...) zur WSN-Subscription. Der Proxy kümmert sich dann mittels Muse um die Generierung und Versendung der SOAP-Messages.

Bei jedem Button-Klick wird mittels der Methode tell beziehungsweise err in den Output-Frame des GUI geprintet. Diese Methoden sind hier nicht gelistet, da sie wiederum sehr GUI-lastig sind und nichts mit der Funktionalität der Web Services zu tun haben.
Nun müssen wir unser Package noch kompilieren. Damit das Jar-Archiv nachher auch ausführbar ist, müssen wir im build.xml File bei der Kompilierung noch etwas hinzufügen. Im Target „java“ gibt es das Element <jar> mit dem Unterelement <fileset>. Auf diese Ebene fügen wir noch einen Eintrag zur Generierung der Manifest-Datei hinzu, so dass dieser Teil der build.xml-Datei folgendermaßen aussieht:

```
<jar destfile="${JAR_FILE}">
 <fileset dir="${JAVA_DEST_DIR}">
  <include name="**/*.class"/>
 </fileset>
 <manifest>
  <attribute name="Main-Class" value="org.apache.ws.muse.fopra.httpd.tester"/>
  <attribute name="Class-Path" value="lib/muse-core-2.2.0.jar
  lib/muse-util-2.2.0.jar lib/muse-util-qname-2.2.0.jar
  lib/muse-util-xml-2.2.0.jar lib/muse-wsa-soap-2.2.0.jar
  lib/muse-wsdm-muws-adv-api-2.2.0.jar lib/muse-wsdm-muws-adv-impl-2.2.0.jar
  lib/muse-wsdm-muws-api-2.2.0.jar lib/muse-wsdm-muws-impl-2.2.0.jar
  lib/muse-wsdm-wef-api-2.2.0.jar lib/muse-wsdm-wef-impl-2.2.0.jar
  lib/muse-wsn-api-2.2.0.jar lib/muse-wsn-impl-2.2.0.jar
  lib/muse-wsrf-api-2.2.0.jar lib/muse-wsrf-impl-2.2.0.jar
  lib/muse-wsrf-rmd-2.2.0.jar lib/muse-wsx-api-2.2.0.jar
  lib/muse-wsx-impl-2.2.0.jar lib/wsdl4j-1.6.1.jar lib/xalan-2.7.0.jar
  lib/xercesImpl-2.8.1.jar lib/xml-apis-1.3.03.jar"/>
 </manifest>
</jar>
```

Damit ist nun gewährleistet, dass eine korrekte Manifest-Datei im client.jar File angelegt wird, wenn wir mit *ant* kompilieren. Es wird die ausführbare Klasse mit der Main-Methode definiert, sowie der Class-Path festgelegt, damit alle Muse-Bibliotheken gefunden werden können.
Als nächstes rufen wir nun *ant* auf und generierieren ein client.jar file, das eine Manifest-Datei und unsere Klassen http_serverProxy.java, http_server.java und tester.java enthält.

5.10 Testen der WS Resources

Wir haben nun alles fertig entwickelt und können mit dem Testen unserer Web Services beginnen. Dazu ist es selbstverständlich nötig, dass Tomcat läuft, was, wie weiter oben auch bereits erläutert, durch ein simples startup.sh/.bat im Tomcat/bin ordner funktioniert. Tomcat ist immer sehr redselig und beschreibt beim Start bereits ausführlich das Deyploment unserer beiden Web Services gefolgt von einer Information, wie lange das Starten gedauert hat. Unter Windows wird eine eigene Tomcat-Konsole gestartet, wo man life alle von Tomcat ausgegebenen Informationen, sowie erhaltene und gesendete SOAP-Nachrichten mitverfolgen kann. Unter Linux scheint Tomcat auf den ersten Blick stumm. Nicht einmal so etwas wie „Erfolgreich gestartet“ bekommt der Benutzer zu sehen. unter /logs/catalina.out jedoch

findet sich ein Logfile, das genau den selben Inhalt enthält, wie in der Windows-Konsole ausgegeben wird. Somit ist erstmal keine Life-Verfolgung des Outputs möglich unter Linux. Erst mit weiteren Tools kann man diese catalina.out-Datei life bei jeder Änderung sich anzeigen lassen.
Als nächstes starten wir unser Test-Programm, indem wir im workDir/client Verzeichnis einfach

```
java -jar client.jar
```

aufrufen.
Das Programm bietet eine intuitives GUI, das wie im folgenden Bild dargestellt, aussieht. Wir können die EPRs unserer beiden Web Services festlegen und mittels dem Button Setup

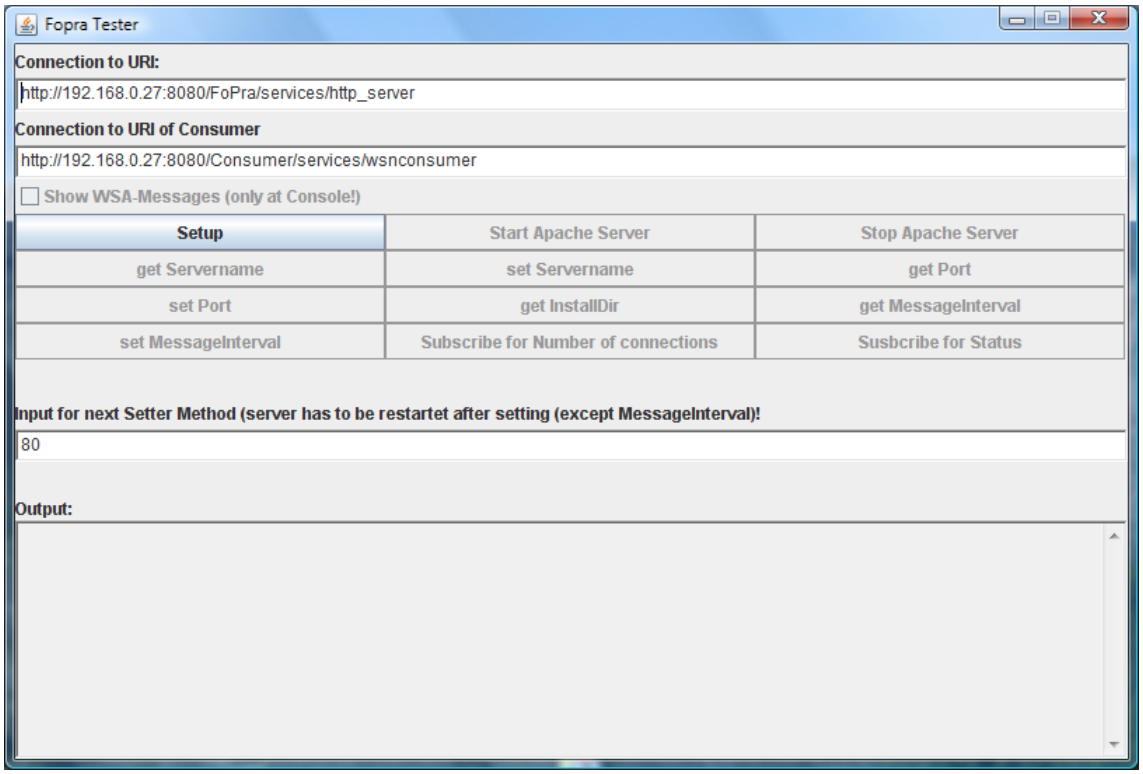

Abbildung 5.5: GUI des Testers

diese für unsere zukünfitgen Aktionen konfigurieren. Die anderen Buttons sind dabei selbsterklärend. Vor jedem Set muss in der „Input for next Setter“-Zeile natürlich ein korrekter Wert eingegeben werden, der dann übermittelt wird. Die Ausgaben erfolgen zum Teil in der grafischen Ausgabe, als Teil des GUI, zum anderen Teil direkt in der Konsole, von der aus das Jar-Archiv gestartet wurde. SOAP/WSA-Nachrichten können nur direkt in der Konsole ausgegeben werden, da diese direkt von einer Klasse aus einem Muse-Jar generiert werden. Alles andere sehen wir im GUI im Output-Panel. Der Haken bei „WSA“ aktiviert beziehungsweise deaktivert die Ausgabe das WSA-Nachrichten in der Konsole. Die Nachrichten entsprechen dabei genau den SOAP-Requests und -Responses.
Wir geben also zuerst die beiden URIs unserer beiden EPRs ein. Diese könenn wir, falls wir

sie vergessen haben sollten, mittels der Tomcat-Web-Oberfläche, wie bereits oben beschrieben, nachlesen.
Nach Eingabe der URIs klicken wir auf Setup und alle anderen Buttons werden benutzbar. Setup initialisiert dabei den Client-Proxy und gibt alle anderen Buttons frei. Unser, vom GUI im Hintergrund instantiierter http_serverProxy, ist der eigentliche Kommunikator. Das GUI dient lediglich zur intuitiven Ausführung der Befehle und zur grafischen Aufbereitung. Als nächstes können wir zum Beispiel einige Gets ausführen, um den konfigurierten Namen oder Port unsereres gemanageten Apache HTTP Web Servers zu erfahren. Beim ersten Aufruf auf eine Capability des Web Services, wir diese initialisiert, was wir sehr gut im Tomcat-Logfenster beziehungsweise im catalina.out-File beobachten können. Wenn der Haken bei „Show WSA-Messages“ gesetzt ist, bekommen wir auch die genauen WSA-Messages in der Konsole zu sehen. Diese schauen bei einer Abfrage des Servernamens wie folgt aus:

```
[CLIENT TRACE] SOAP envelope contents (outgoing):

<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
 <soap:Header>
  <wsa:To xmlns:wsa="http://www.w3.org/2005/08/addressing">
     http://192.168.0.27:8080/FoPra/services/http_server</wsa:To>
  <wsa:Action xmlns:wsa="http://www.w3.org/2005/08/addressing">
     http://docs.oasis-open.org/wsrf/rpw-2/GetResourceProperty
        /GetResourcePropertyRequest
  </wsa:Action>
  <wsa:MessageID xmlns:wsa="http://www.w3.org/2005/08/addressing">
     uuid:e3543701-05d3-f61b-f4bb-8c8d6188d48d
  </wsa:MessageID>
  <wsa:From xmlns:wsa="http://www.w3.org/2005/08/addressing">
   <wsa:Address>http://www.w3.org/2005/08/addressing/role/anonymous</wsa:Address>
  </wsa:From>
 </soap:Header>
 <soap:Body>
  <wsrf-rp:GetResourceProperty xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/rp-2"
    xmlns:pfx2="http://ws.apache.org/muse/fopra/httpd"> pfx2:ServerName
  </wsrf-rp:GetResourceProperty>
 </soap:Body>
</soap:Envelope>

[CLIENT TRACE] SOAP envelope contents (incoming):

<soapenv:Envelope xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
   xmlns:wsa="http://www.w3.org/2005/08/addressing">
 <soapenv:Header>
  <wsa:To>http://www.w3.org/2005/08/addressing/anonymous</wsa:To>
  <wsa:ReplyTo>
   <wsa:Address>http://www.w3.org/2005/08/addressing/none</wsa:Address>
  </wsa:ReplyTo>
  <wsa:MessageID>urn:uuid:F783C8C625658AEB0512211357305834</wsa:MessageID>
```

```
  <wsa:Action>
     http://docs.oasis-open.org/wsrf/rpw-2/GetResourceProperty
        /GetResourcePropertyRequest
  </wsa:Action>
  <wsa:RelatesTo wsa:RelationshipType="http://www.w3.org/2005/08/addressing/reply">
     uuid:e3543701-05d3-f61b-f4bb-8c8d6188d48d
  </wsa:RelatesTo>
 </soapenv:Header>
 <soapenv:Body>
  <wsrf-rp:GetResourcePropertyResponse
     xmlns:tns="http://axis2.platform.core.muse.apache.org"
     xmlns:wsrf-rp="http://docs.oasis-open.org/wsrf/rp-2">
   <pfx2:ServerName xmlns:pfx2="http://ws.apache.org/muse/fopra/httpd">
      FoPraServer
   </pfx2:ServerName>
  </wsrf-rp:GetResourcePropertyResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Auf der Tomcat-Seite sind die eingehenden Nachrichten, also die von hier ausgehenden, auch zu sehen.

Unser GUI interpretiert die eingehendenden Messages und gibt diese im Output-Feld aus: Sollten wir uns jedoch bei der Eingabe der URI unseres Haupt Web Services vertippt haben,

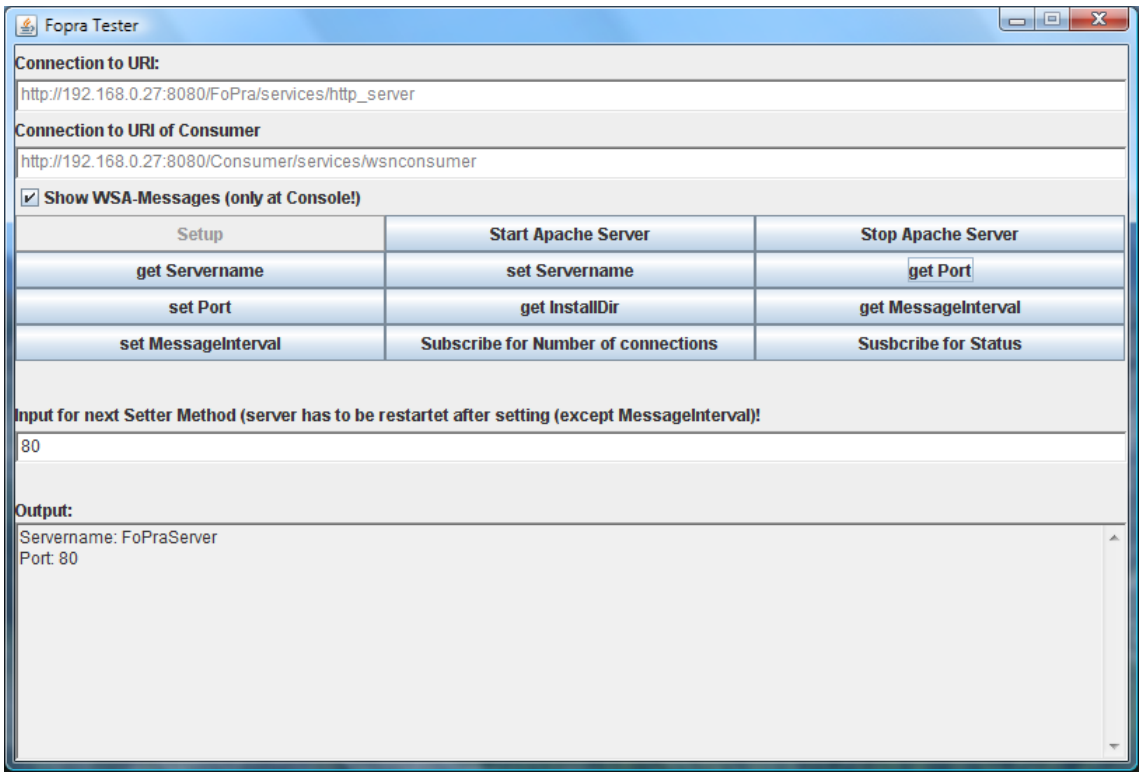

Abbildung 5.6: GUI des Testers nach einem Aufruf von Setup, gefolgt von get Servername und get Port

so wird dabei eine Fehlermeldung in der GUI ausgebgeben. Diese könnten zum Beispiel wie folgt aussehen:

```
ERROR: org.apache.muse.ws.addressing.soap.SoapFault:
   Service not found operation terminated !!
```

Diese Meldung besagt, dass wir zwar eine WS Resource adressiert haben, die aber den von uns eingegeben Service nicht kennt. Andererseits könnte aber auch zum Beispiel folgende Fehlermeldung auftreten:

```
ERROR: org.apache.muse.ws.addressing.soap.SoapFault:
   Connection refused: connect
```

Dies bedeutet, dass an der von uns eingegebenen URI keine WS Resource erreichbar ist. Entweder haben wir uns vertippt, oder eine Firewall blockiert die Nachrichten.
Zu beachten ist dabei, dass solch eine Meldung nicht beim Klick auf Setup auftritt, da dort nur die Logik initialisiert wird, sondern erst beim Durchführen einer konkreten Operation wie zum Beispiel get Servername.
Doch gehen wir nun davon aus, dass wir den Web Service korrekt erreichen können. Wir wollen als nächstes den Servernamen auf „UnserErsterServer“ ändern. Dazu geben wir „UnserTestServer“ bei „Input for next Setter“ ein und klicken anschließend auf „set Servername.

Ein erfolgreiches Update sehen wir wiederum im Output-Panel. Dort sollte zu lesen sein: „Updated Servername to: UnserTestServer“. Den Port können wir ebenso ändern.
Natürlich ist bei allen Set-Methoden darauf zu achten, dass eine valide Eingabe statt findet. Wir haben nun unseren Server fertig konfiguriert und möchten ihn mit einem Klick auf „Start Apache Server“ starten. Ein „Start: OK“ in der Ausgabe bestätigt uns den erfolgreichen Start. Alle zukünftigen Set-Methoden, außer dem Setzen des MessageIntervals benötigen einen Restart des Servers, bevor sie in Kraft treten.
Wir können die Funktionalität des Servers überprüfen, indem wir uns mit einem beliebigen Browser auf die IP des Rechners, auf dem der HTTP Web Server läuft und dem von uns konfiguierten Port verbinden. Mittels get Port haben wir zuvor schon gesehen, auf welchem Port der Server aktuell lauscht. Der Apache HTTP Web Server gibt dabei standarmäßig eine primitive, aber eindeutige Web Seite aus:

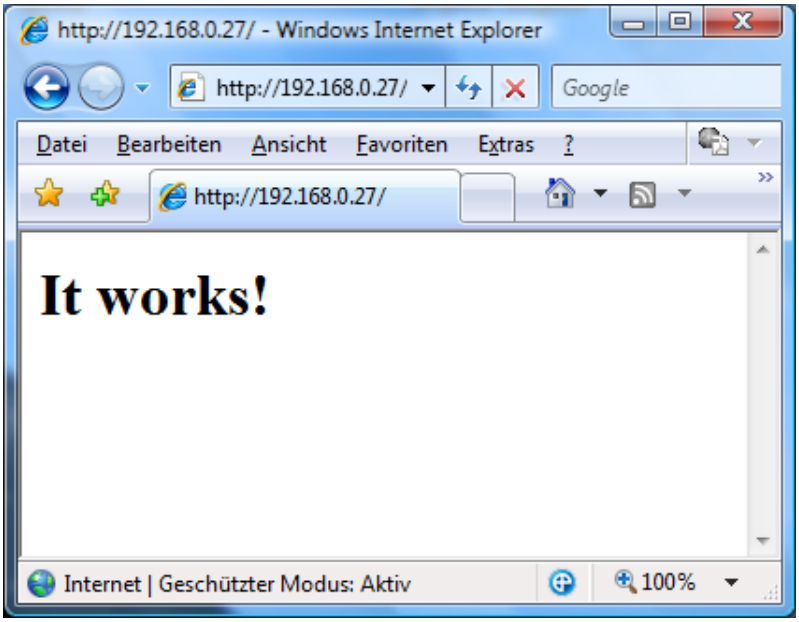

Abbildung 5.7: It works!

Bis jetzt haben wir nur mit unserem Haupt Web Service kommuniziert, also mit der URI an oberer Stelle. Da dieser Web Service auch ein Nachrichten Producer ist, kann er Anmeldungen von anderen Web Services entgegennehmen, die bei bestimmten Ereignissen benachrichtigt werden möchten. Dazu benötigen wir nun unseren zweiten kleinen Web Service, der unter der zweiten oben eingegebenen URI erreichbar sein sollte.
Wir möchten diesen Web Service nun bei unserem Hauptservice anmelden, damit dieser im Takt des von uns definierten und jederzeit änderbaren MessageIntervals die Anzahl der Verbindungen zum gemanageten HTTP Web Server unserem zweiten Web Service, dem WSN-Consumer mitteilt. Dazu klicken wir auf „subscribe for Number of connections“. Unter Windows können wir im Tomcat-Fenster die WSA-Messages life verfolgen, die beide Web Services nun miteinander austauschen. Unter Linux hilft leider nur ein Blick in das catalina.out Logfile.
Unser Consumer-Service hat jedoch auch eine von uns, wie weiter oben beschrieben, im-

plementierte Capability, die ein kleines GUI erzeugt, in dem die eingehenden Nachrichten angezeigt werden. Dieses GUI öffnet sich nun auf dem Rechner auf dem unser Consumer Service anzufinden ist, also dort, wo Tomcat läuft. Das GUI meldet dabei, von wem eine Nachricht zu welchem Topic zu welcher Uhrzeit empfangen wurde und insbesondere wie der Inhalt der Nachricht lautet. Standarmäßig, wenn wir das MessageInterval nicht geändert habem wird alle 10 Sekunden einen Nachricht gepublished. Nach 20 Sekunden sieht unser Consumer Watcher GUI also folgendermaßen aus:

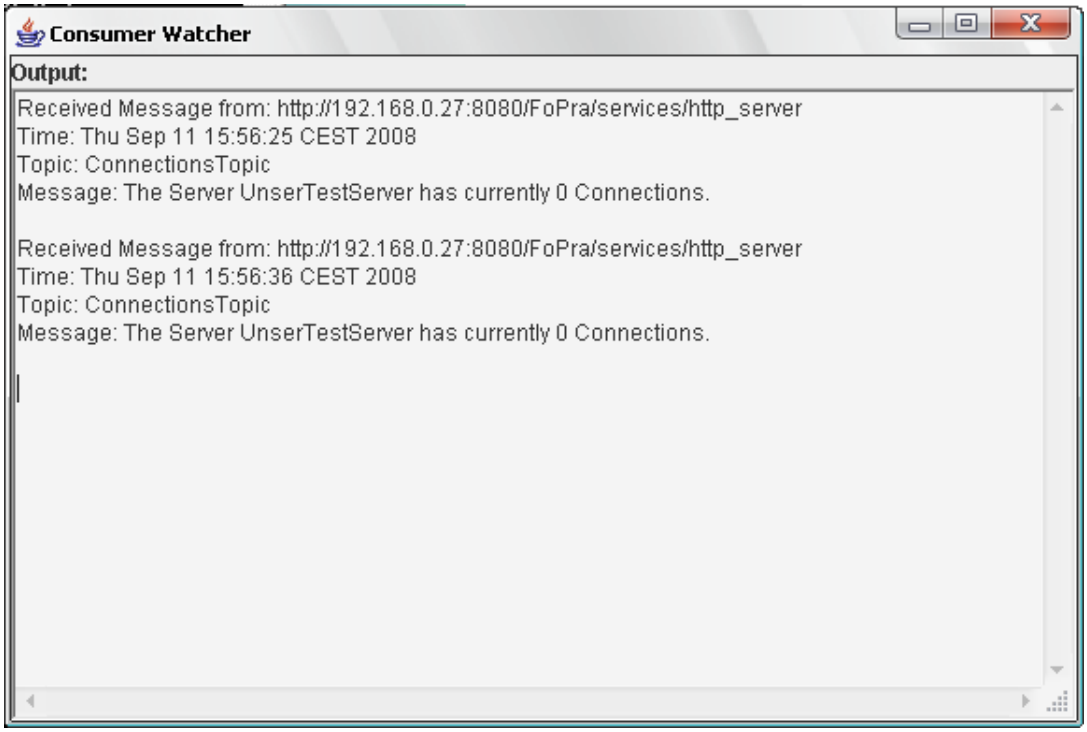

Abbildung 5.8: Der Consumer Watcher

Wir sehen, das aktuell keine Verbindungen zu unserem gemanageten HTTP Web Server aufgebaut sind. Wenn wir nun nochmal wie weiter oben eine Test-Verbindung zu unserem Server aufbauen, um ein schönes „It works!“ zu sehen, dann erkennt das unser Web Service und publishet eine entsprechende Meldung an den Consumer.

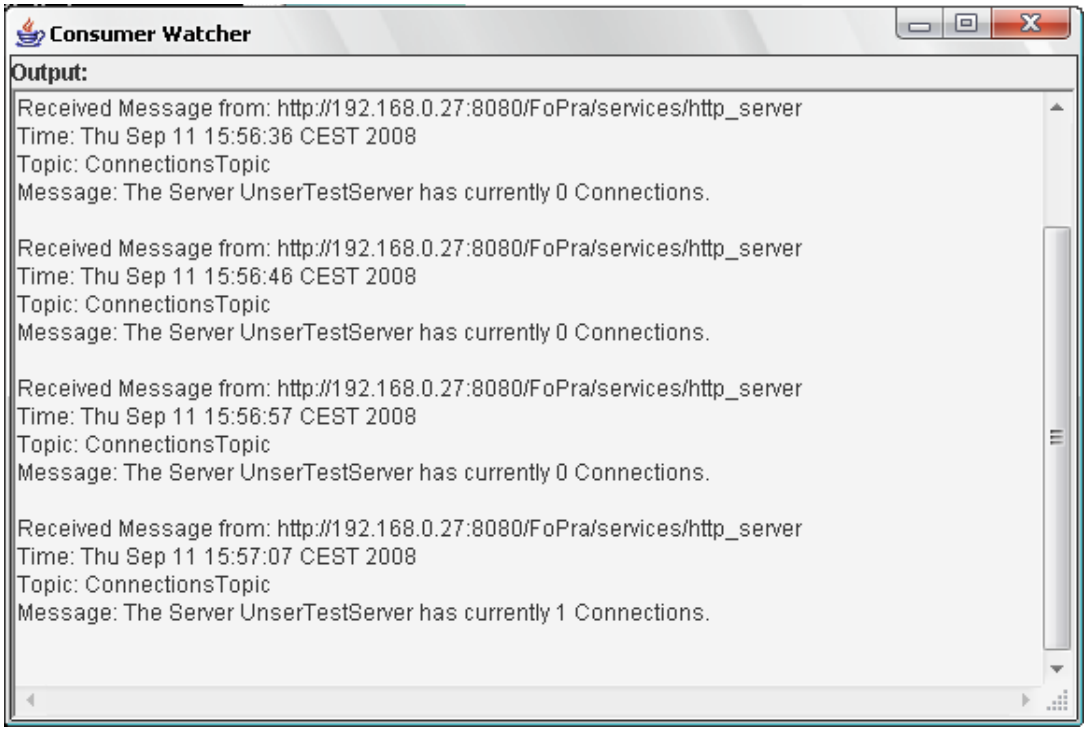

Abbildung 5.9: Der Consumer Watcher mit Anzeige, dass eine Verbindung zum gemanageten HTTP Web Server besteht.

Sollte die GUI nicht erschienen sein, so haben wir mit Sicherheit eine falsche URI für unseren Consumer eingegeben, so dass wir folglich beim Subscriben auch einen nicht existenten Web Service angemeldet haben. Die Subscription an sich ist natürlich erfolgreich gewesen, so dass unser Tester-Programm auch keine Fehlermeldung ausgeben kann. Der eigentliche Fehler tritt natürlich erst auf, sobald unser FoPra Web Service versucht, einen EPR an der falschen URI zu erreichen, der nicht existent ist. Diese Fehlermeldung sehen wir dann im Tomcat-log-Fenster beziehungsweise in der catalina.out-Datei. Es gibt dabei verschiedene Meldungen wie zum Beispiel: not a valid URI, URI is not absolute, oder aber auch erst auf den zweiten Blick zu erkennende falsche URIs: Axis beschwert sich über nicht geschlossene <HR>-Tags. Dies kommt davon, dass man zwar eine korrekte WS Resource adressiert hat, die aber keine Nachrichten annehmen kann. Dabei wird ein http 500 Fehler von Tomcat generiert und dieser dann als Response gewertet, der nicht interpretiert werden kann, da die http 500 Errorpage kein gültiges XML ist und somit ein Parse-Error auftritt, der dann als Beschwerde über ein nicht geschlossenes <HR>-Tag ausgegeben wird. Auch kann irgendwo im langen log-Output ein simples „null“ ohne weiteren Hinweis auftreten. Auch das kann eine Beschwerde über eine gepublishete Nachricht sein, die nicht angenommen wurde.
Gehen wir nun jedoch davon aus, dass die eingetippte URI korrekt ist. Dann möchten wir uns für einen zweiten Topic anmelden, nämlich einen Topic, der genau dann eine Message published, wenn der Web Server nicht mehr erreichbar ist, obwohl er es sein sollte. Das testen wir folgendermaßen:
Natürlich muss dazu der HTTP Web Server über unseren Web Service gestartet werden mittels einem Klick auf „Start Apache Server“, was wir ja bereits gemacht haben. Danach

subscriben wir unseren kleinen Consumer-Service für diesen Topic bei unserem großen FoPra-Service mittels dem Button „Subscribe for Status“. Nachdem die Subscription erfolgreich war, passiert erstmal gar nichts, außer dass unser FoPra Web Service im MessageInterval-Takt prüft, ob der Apache HTTP Web Server noch online ist. Sollten wir den Web Server per „Stop Apache Server“ anhalten, so wird die Überprüfung ebenfalls angehalten, da der Server ja absichtlich heruntergefahren wurde.
Wir müssen also ein Herunterfahren nun außerhalb unserer Web Service Logik erzwingen. Dazu müssen wir in das Installationsverzeichnis des Web Servers wechseln, welches wir mittels get InstallDir von unserem Web Service erfahren. In diesem Verzeichnis gibt es ein Unterverzeichnis namens bin, darin führen wir folgenden Befehl aus für ein Herunterfahren des Apache HTTP Web Servers:

```
httpd -k stop
```

Spätestens nach Ablauf eines MessageIntervals sollte nun unser FoPra Web Service eine Meldung an den Consumer Service schicken. Dieser sollte die Nachricht entgegennehmen und in seinem GUI lesbar darstellen.

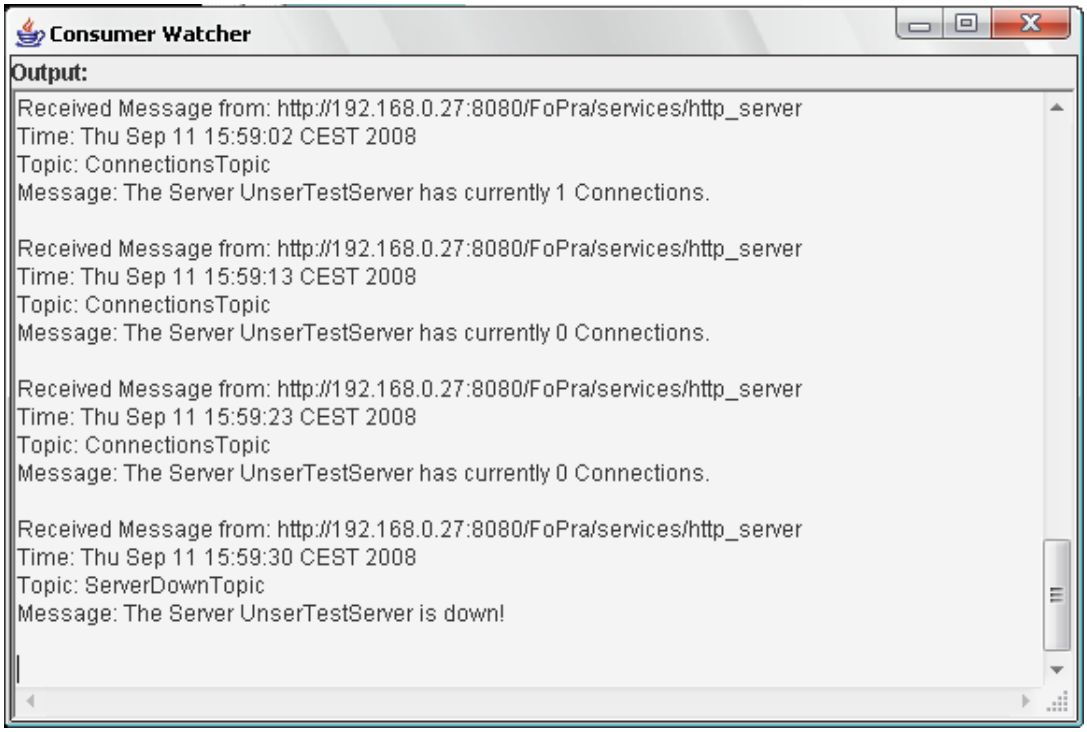

Abbildung 5.10: Der Consumer Watcher mit Anzeige, dass der Apache Web Server down ist.

Da der Consumer Service nun für beide Topics angemeldet ist, erhält er auch Nachrichten zu beiden Topics. So kommen nun im Abstand von MessageInterval je eine Nachricht zu den derzeitigen Verbindungen (die nach einiger Verzögerung auf Grund der noch offenen und nicht gültig beendeten etwaigen Verbindungen auf 0 absinken) und eine Nachricht, dass der Web Server nicht mehr online ist. Wir können nun mit einem simplen Klick auf „Start Apache Server“ versuchen, den Web Server wieder zu starten, was in unserem Fall natürlich

klappt, da er ja nicht hängt, sondern einfach zu Testzwecken beendet wurde. Sobald er wieder läuft, erhält der Consumer nur noch Nachrichten zu den derzeitigen Verbindungen.
Sollte es zu keinem Problem kommen, ist alles korrekt implementiert und wir können uns über unsere erste selbstenwickelte WSDM Management Anwendung freuen.

6 Zusammenfassung

6.1 Zusammenfassung der Arbeit

In Kapitel 5 haben wir uns ausführlich mit der Entwicklung zweier Web Services innerhalb einer Apache Muse Umgebung befasst. Diese beiden Web Services dienen uns dazu, einen Apache HTTP Web Server zu verwalten und zu konfigurieren. Der von uns entwickelte FoPra Web Service bietet dazu verschiedene Möglichkeiten im Rahmen der WSDM-, WSRF- und WSN-Spezifikationen. So ist es uns möglich, den Servernamen und dessen Port zu konfigurieren, sowie das Installationsverzeichnis abzufragen. Der Server kann gestartet und gestoppt werden. Außerdem kann unser zweiter von uns entwickelter Web Service Conusmer im Sinne eines WSN-Consumers bei unserem Fopra Web Service zu zwei verschiedenen Topics angemeldet, also subscribed werden. Der Fopra Service schickt in einem jederzeit konfigurierbaren Interval von jedem Topic ausgehend eine Nachricht an alle für diesen Topic angemeldeten Web Services. Einer der Topics verteilt Informationen über die Anzahl der gerade mit dem HTTP Web Server verbundenen Clients. Der andere Topic verschickt eine Nachricht, sobald der HTTP Web Server nicht mehr erreichbar sein sollte auf Grund eines Absturzes oder eines Abschaltens außerhalb des verwaltenden Web Services.
Diese beiden Web Services haben wir mit Hilfe von Apache Muse entwickelt und innerhalb einer Axis2 Umgebung als WAR-Files einem Tomcat-Server übergeben.
Danach haben wir einen Tester entwickelt und unsere beiden Web Services getestet.

6.2 Interpretation der Ergebnisse

Apache Muse bietet eine Java-Implementierung von WSDM, WSRF und WSN. Laut Muse muss man sich nicht großartig mit der Spezifikation auseinandersetzen. Diese Aussage kann ich nicht uneingeschränkt teilen. Ohne zu wissen, was die einzelnen Spezifikationen besagen ist es nahezu unmöglich sinnvolle Web Services zu definieren. Nach einer Einarbeitungszeit in die jeweiligen Spezifikationen, sowie Lernen der Grundlagen der Web Service Description Language (WSDL) ist es jedoch möglich relativ schnell einfache Web Services selbst zu designen. Das Muse beiliegende WSDL-Template, das bereits einen Großteil der WSDM-, WSRF- und WSN-Spezifikationen definiert, bietet eine gute Grundlage, um mächtige Web Services zu erstellen. Selbst entwickelte Capabilities sind jedoch wieder deutlich schwerer zu entwickeln und nur lösbar, wenn man WSDL zumindest in den Grundzügen verstanden hat und anwenden kann. Der größte Teil bei der Entwicklung der Services ist jedoch die konkrete Implementierung der Java Klassen der eigenen Capabilities. Die eigentliche Logik der Web Services ist dort anzutreffen und muss vollständig eigenhändig programmiert werden. Muse bietet also nur die Implementierung der Spezifikationen, also die grundlegenden Funktionen wie zum Beispiel: Abfragen von Properties, Verteilen von Nachrichten, Verwalten von Subscriptions und Empfangen von Nachrichten. Der Entwickler jedoch muss sich überlegen und implementieren, woher die Properties ausgelesen werden, wohin sie geschrieben werden, wel-

che Nachrichten akzeptiert werden, was bei Eintreffen einer Nachricht ausgeführt wird, wie eine Anmeldung konkret durchgeführt wird, wie Nachrichten empfangen und interpretiert werden und zu guter Letzt wie Nachrichten überhaupt versendet werden.
Muse bietet also wie gesagt nur ein rudimentäres Grundgerüst, generiert mittels wsdl2java, auf dem der Entwickler dann aufsetzt und die komplette Logik selbst implementieren muss.

6.3 Schwierigkeiten bei der Entwicklung

Während dieser Arbeit sind einige doch relativ große Probleme und Ungereimtheiten aufgetreten.
Muse liegt seit mehr als eineinhalb Jahren in Version 2.2 vor. Darauf basierend wird zwar weiterentwickelt, jedoch bekommt man Bugfixes nur relativ schwierig und diese sind meist zu solch konkreten Problemen, dass man sie momentan noch vernachlässigen kann. Muse 3.0 ist bereits ebenfalls seit über einem Jahr angekündigt und soll viele Neuerungen und Vereinfachungen mit sich bringen. Bis jetzt ist aber dazu noch kaum etwas zu lesen, da wohl noch eine weitere Version 2.3 dazwischen geplant ist, die aber noch nicht final erschienen ist und wohl nur die vielen kleinen Bugfixes seit Version 2.2 enthält.
Wie bereits zwischendurch erwähnt, setzt Muse mit Axis2 in einem Tomcat momentan zwingend Java 1.5 voraus und ist im aktuellen Stand nicht mit Java 1.6 lauffähig. Dies wird eigentlich nirgends erwähnt, zumindest war auf der offiziellen Seite von Muse dazu nichts zu finden. Einzig in einigen Diskussionsforen, auch dem offiziellen, liest man hin und wieder bei Problemen, man solle doch Java 1.5 versuchen. Meist hat das zur Lösung der Probleme geführt.
Da ein Web Service eigentlich stetig unter Weiterentwicklung steht, sei es durch Erweiterung des Funktionsumfangs oder durch Abänderung der logischen Implementierung, gibt es in dieser Hinsicht große Probleme, wenn der Web Service auf Muse basiert. Mittels wsdl2java werden ja bekanntlich die Java-Klassen der benutzerspezifischen Capabilities generiert, also die Capabilites, an der der Entwickler überhaupt nur arbeitet. Nehmen wir also an, der Web Service wurde bis zu einem bestimmten Punkt per WSDL beschrieben und danach mittels wsdl2java die Capabilites generiert. Danach hat der Entwickler die Logik implementiert. Soweit so gut - doch was ist, wenn nun der Funktionsumfang erweitert werden soll? Dazu muss zuerst das WSDL-Dokument abgeändert werden und die neuen Funktion beschrieben werden. Wenn man nun mittels wsd2ljava die Capabilites generieren will, wird dies nicht erledigt, da die Java-Klassen nicht überschrieben werden, da sie ja bereits eigenen Code enthalten. Also kann auf die neu-definierten Funktionen nicht zugegriffen werden. Wird jedoch das Überschreiben mittels -overwrite oder durch explizite Anpassung innerhalb der .overwrite Datei erzwungen, so haben die Capabilites zwar alle neuen und alten Funktionsgerüste, aber die alte Logik ist nicht mehr vorhanden. Der Entwickler wird also gezwungen, die Logik der alten Funktionen zuerst irgendwo zu sichern, danach mittels -overwrite eine Neugenerierung zu erzwingen um anschließend die alte Logik wieder in die neuen Klassen zu kopieren. Erst danach kann er den neuen Funktionen die neue Logik hinzufügen. Wieso kann Muse hier noch kein intelligentes wsdl2java bieten, das die alten Klassen um die neuen Funktionen erweitert?
Die Kompilierung des Projektes mittels ant scheint auf den ersten Blick sehr einfach und intuitiv. Doch gilt stehts zu beachten, dass hier ebenfalls mit Java 1.5 kompiliert wird, also darf der Capability-Quellcode selbstverständlich keine Java 1.6-only Funktionen verwenden.

Eine Kompilierung per ant mit Java 1.6 gelingt zwar, doch kommt es später im Einsatz innerhalb Tomcats wiederum zu Problemen. Daher gilt zu beachten, dass sowohl die Kompilierung als auch der Produktiveinsatz mittels einer Java 1.5 Umgebung durchgeführt wird. Tomcat bietet dazu jedoch von Haus aus bereits die Möglichkeit, die zu verwendende Java Umgebung beim Start festzulegen, falls mehrere Java Umgebungen installiert sein sollten. So ist es sogar möglich, eine eigene Java-Umgebung für Tomcat festzulegen, falls dies gewünscht ist.
Desweiteren, wie wir bei der Entwicklung des Consumer Service gesehen haben, generiert wsdl2java keine leeren Capabilites. Da wir aber dennoch eine benutzerspezifische Capability benötigten, um einen MessageListener dem Consumer hinzuzufügen, mussten wir uns selbst um die Entwicklung der Capability kümmern. Diesmal von Grund auf und nicht wie bei unserem FoPra Service, bei dem das Methoden-Gerüst mit Signaturen bereits von Muse generiert wurde, so dass wir direkt mit Entwicklung der Logik beginnen konnten.
Ähnliches Phänomen gab es bereits bei Entwicklung des FoPra Web Services. Dort wurden die von uns im WSDL-File definierten StartFailedFault und StopFailedFault nicht automatisch angelegt, sondern mussten manuel implementiert werden, obwohl diese im Grunde keine Funktionalität haben. Es wäre wohl ein Leichtes, diese Faults genauso wie auch die MyCapability Klasse zumindest als Skeleton anzulegen.
Ein großes anderes Problem sind Fehlermeldungen von Muse. Diese sind alles andere als gut. Es wird - wenn überhaupt - die Java-Fehlermeldung geworfen, manchmal nicht einmal diese, sondern nur ein einfaches „null“. Man weiß in diesem Fall weder woher, noch warum die Meldung geworfen wurde. Eine Fehlerbeschreibung wäre oft sinnvoll und hilfreich gewesen. Die in Kapitel 5 schon beschriebende Fehlermeldung, dass ein <HR>-Element nicht geschlossen wurde, ist nur ein Beispiel von vielen sinnfreien Fehlermeldungen die den eigentlichen Fehler gar nicht erst beschreiben oder nur einen Hinweis darauf geben. Fehlersuche in Muse-Umgebungen ist schwer und mühselig!

6.4 Eigene Worte und Fazit

Doch nach all den Hürden und Schikanen konnten am Ende doch zwei funtionierende Web Services implementiert werden, die sowohl WSDM-, WSRF- als auch WSN-Standards gerecht werden, wie wir mit dem selbst entwickelten Test-Tool festgestellt haben.
Während den letzen Monaten konnte ich jedoch feststellen, dass Muse noch in sehr kleinen Kinderschuhen steckt. Muse bietet zwar viele komfortable Funktionen und die Implementierung der Spezifikationen ist natürlich eine große und tolle Sache, doch in Hinsicht auf Entwicklerzugänglichkeit, vor allem in Hinblick auf intuitives Arbeiten und Entwickeln, fehlt es doch noch weit. Wie bereits im vorherigen Absatz beschrieben gibt es viele Dinge zu beachten und viele Probleme die auftreten, die man nicht erwarten würde.
Muse ist meiner Ansicht nach im aktuellen Stand nicht wirklich für den produktiven Einsatz geeignet. Vor allem in Hinblick darauf, dass die Spezifikationen sowieso so gut wie nicht im Einsatz sind. Auch unter dem Gesichtspunkt, dass WSDM und WSM (DMTF Web Services for Management-Spezifikation, siehe [DMT07]) über kurz oder lang zu einer gemeinsamen Spezifikation verschmelzen soll, die dann von Muse 3.0 implementiert werden soll, sollte man sich gut überlegen, nicht auf Muse 3.0 zu warten. Die meisten Foren-Besucher des offiziellen Entwicklerforums beschreiben meist nur Probleme, die zu Beginn auftreten, und Kinderkrankheiten.

Unter einem anderen Gesichtspunkt betrachtet bietet Muse jedoch etwas Neues, Individuelles. Die Arbeit der Muse-Entwickler ist auf jeden Fall grandios, auch der Support hilft gerne mal weiter, auch wenn dies nicht immer zur gewünschten Lösung führt.
Management wird derzeit meist noch über andere Mechanismen geregelt, wie zum Beispiel dem SNMP. Doch dieses Protokoll kann nicht mit der Mächtigkeit von Web Services mithalten. Und Web Services sind stark im Kommen, zwar noch nicht im Sinne von Management, doch überall anders werden sie eingesetzt.
Muse demonstiert interessante Möglichkeiten zu interessanten Spezifikationen. Doch genau diese werden sich ändern - genauso wie auch Muse.
Warten wir gespannt auf Version 3.0...

Abbildungsverzeichnis

Literaturverzeichnis

[Apa08a] Apache: *Muse*. http://ws.apache.org/muse, 2008.

[Apa08b] Apache: *Muse Tutorial*. http://ws.apache.org/muse/docs/2.2.0/tutorial/index.html, 2008.

[Apa08c] Apache: *Muse: wsdl2java Tool*. http://ws.apache.org/muse/docs/2.2.0/manual/tools/wsdl2java.html, 2008.

[DC04] Dave Chappell, Lily Liu: *Web Services Brokered Notification 1.2*. http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BrokeredNotification-1.2-draft-01.pdf, 2004.

[DMT07] DMTF: *WBEM: Web Services for Management*. http://www.dmtf.org/standards/wbem/wsman, 2007.

[Heg07] Hegering, H.-G.: *Netz- und Systemmanagement*. http://www.nm.ifi.lmu.de/teaching/Vorlesungen/2007ss/nsmgmt/skript/Kap01.pdf, 2007.

[OAS08a] OASIS: *Web Services Distributed Management (WSDM) TC*. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm, 2008.

[Oas08b] Oasis: *WS-MetadataExchange*. http://xml.coverpages.org/WS-MetadataExchange.pdf, 2008.

[SG04a] Steve Graham, Bryan Murray: *Web Services Base 2 Notification 1.2*. http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf, 2004.

[SG04b] Steve Graham, Jem Treadwell: *Web Services Resource Properties 1.2*. http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-04.pdf, 2004.

[ST04] Steve Tuecke, Lily Liu, Sam Meder: *Web Services Base Faults 1.2*, 2004.

[TM04] Tom Maguire, David Snelling: *Web Services Service Group 1.2*. http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ServiceGroup-1.2-draft-02.pdf, 2004.

[uI04] IBM, Globus Alliance und: *The WS-Resource Framework*, 2004.

[Vam04] Vambenepe, William: *Web Services Topics 1.2*. http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf, 2004.

[VB06] Vaughn Bullard, Bryan Murray, Kirk Wilson: *An Introduction to WSDM*. http://www.oasis-open.org/committees/download.php/16998/wsdm-1.0-intro-primer-cd-01.doc, 2006.

[Wee04] Weerawarana, Jeffrey Frey Steve Graham Karl Czajkowski Donald Ferguson Ian Foster Frank Leymann Tom Maguire Nataraj Nagaratnam Martin Nally Tony Storey Igor Sedukhin David Snelling Steve Tuecke William Vambenepe Sanjiva: *Web Services Resource Lifetime 1.1*. http://www.ibm.com/developerworks/library/ws-resource/ws-resourcelifetime.pdf, 2004.

[Wik08a] Wikipedia: *Apache Ant*. http://de.wikipedia.org/wiki/Ant, 2008.

[Wik08b] Wikipedia: *Apache Axis2*. http://de.wikipedia.org/wiki/Axis2, 2008.

[Wik08c] Wikipedia: *Apache HTTP Server*. http://de.wikipedia.org/wiki/Apache_HTTP_Server, 2008.

[Wik08d] Wikipedia: *Apache Tomcat*. http://de.wikipedia.org/wiki/Apache_Tomcat, 2008.

[Wik08e] Wikipedia: *SOAP*. http://de.wikipedia.org/wiki/SOAP, 2008.

[Wik08f] Wikipedia: *Web Service*. http://de.wikipedia.org/wiki/Web_Service, 2008.

[Wik08g] Wikipedia: *Web Services Description Language*. http://de.wikipedia.org/wiki/WSDL, 2008.

[Win04] Winkler, Don Box Erik Christensen Francisco Curbera Donald Ferguson Jeffrey Frey Marc Hadley Chris Kaler David Langworthy Frank Leymann Brad Lovering Steve Lucco Steve Millet Nirmal Mukhi Mark Nottingham David Orchard John Shewchuk Eugène Sindambiwe Tony Storey Sanjiva Weerawarana Steve: *Web Services Addressing (WS-Addressing)*. http://www.w3.org/Submission/ws-addressing/, 2004.