



## Auf einer Wellenlänge

# Das Open service (-oriented architecture) Gateway interface (OsoaGi)

Matthias Füller, Willi Nüßer

Die Entwicklung von Softwarekomponenten auf Basis des Komponenten-Frameworks OSGi ist heutzutage eine etablierte Technologie. Ebenso ist bei verteilten Anwendungen die serviceorientierte Architektur (SOA) seit langem eine Blaupause für eine mögliche Architektur. Ein genaueres Betrachten der grundsätzlich unterschiedlichen Ansätze zeigt viele Gemeinsamkeiten, sodass eine Kombination nicht abwegig erscheint. Eine Möglichkeit der Integration von OSGi-Komponenten in eine SOA wird in der neuen OSGi-Spezifikation 4.2 beschrieben. Dieser Artikel führt in diese Gedanken ein und zeigt eine exemplarische Implementierung.

Die OSGi Alliance ist ein Industriekonsortium, das unter anderem ein Basisframework für eine komponentenbasierte Java-Entwicklung spezifiziert. Es ist derzeit Gegenstand eines regelrechten Medienrummels. In vielen Java-Anwendungen und -Frameworks wird die Umstellung auf OSGi erwogen [JOnAS,ServiceMix]. Aus Java-Schnittstellen (soweit vorhanden) werden Services und die ehemaligen jar-Dateien mutieren dank spezieller Meta-Informationen schnell zu OSGi-Bundles, wie Komponenten von der OSGi genannt werden. Durch diese geringen Anforderungen erweist sich der Einstieg in OSGi für einen Java-Entwickler als äußerst einfach. Trotz (oder gerade wegen) dieser Einfachheit ist eine saubere Komponententwicklung möglich. Bei guter Trennung von OSGi- und anderem Java-Code ist der Einsatz von OSGi-Bundles in normalen Java-Umgebungen auch weiterhin gegeben.

Genauere Details zu OSGi und seinem Aufbau finden sich auf der OSGi-Seite [OSGi] oder bei den einzelnen OSGi-Framework-Implementierungen [Felix,Equinox].

SOA auf der anderen Seite ist als Architekturkonzept für verteilte Systeme zu verstehen und beschreibt Dienste (Services) als lose gekoppelte Entitäten. Diese Services sind über definierte und standardisierte Schnittstellen zu erreichen. Obwohl eine einheitliche Definition in der Literatur fehlt, lassen sich aber folgende typische Merkmale einer SOA erkennen [W3C,sdm05,Heut06]:

- ▼ **Interoperabilität:** Verwendung von offenen sowie plattformunabhängigen Standards.
- ▼ **Schnittstellenorientierung:** Die Schnittstelle beschreibt die Service-Funktionalität. Die eigentliche Implementierung bleibt verdeckt.
- ▼ **Grob granulare Schnittstellen:** Schnittstellen mit wenigen Operationen und ggf. großen und komplexen Nachrichten, um Kommunikations-Overhead zu minimieren.
- ▼ **Message-orientiert:** Zur Kommunikation zwischen Consumer und Provider werden erweiterbare Nachrichten ausgetauscht. Dies erlaubt die Erweiterung von Services, ohne dass bestehende Consumer angepasst werden müssen.
- ▼ **Service-Registry:** In einer Registry können Services hinterlegt und gesucht werden. Damit ist eine lose Kopplung von Services möglich.



Der Kerngedanke hinter der SOA ist die Wiederverwendung von bestehenden Services und deren Funktionalität. Auch wenn der Vergleich eines Komponenten-Frameworks wie OSGi und eines Architekturkonzeptes auf dem ersten Blick einem Vergleich von Äpfeln mit Birnen gleichkommen mag, finden sich doch bei näherer Betrachtung folgende Schnittmengen:

- ▼ **Serviceorientierung:** OSGi und SOA sehen Funktionen als Services. Nur die Schnittstelle ist relevant. Wie diese Funktionalität implementiert ist, interessiert nicht.
- ▼ **Service-Registry:** Beide Varianten beschreiben eine Registry, in der Informationen zu Services abgelegt und gesucht werden können.

Bei den anderen SOA-Merkmalen Granularität, Message-Orientierung und Interoperabilität ist die Beziehung etwas weniger offensichtlich. Die Verwendung *grob granularer Schnittstellen* stellt dabei sicherlich das kleinste Problem dar, da es als eine Frage des Anwendungsdesigns anzusehen ist und damit ebenso in OSGi-Services Verwendung finden kann. Die Message-Orientierung kann durch die Wahl eines passenden Kommunikationsprotokolls ebenfalls in einer OSGi-Umgebung erreicht werden.

Doch bei der *Interoperabilität* driften SOA und OSGi auseinander, da sich hier die unterschiedlichen Ziele von SOA und OSGi bemerkbar machen. Eine SOA ist für die Kommunikation zwischen verteilten und unterschiedlichen Systemen gedacht, OSGi für die Verwaltung und Kommunikation zwischen den Komponenten innerhalb einer Java-Anwendung. Aber auch Java-Anwendungen müssen häufig mit der Außenwelt kommunizieren. Je nach Implementierung kann es dabei allerdings zu einer starken Verflechtung zwischen eigentlicher Anwendung und dem Kommunikations-Framework kommen. Ein Austausch des Kommunikationsprotokolls wird erschwert.

Elegant wäre zweifelsohne die Möglichkeit, entfernte Dienstschnittstellen stets nur als lokale OSGi-Services anzusprechen und mit ihnen innerhalb der lokalen OSGi-Umgebung zu kommunizieren. Die Consumer dieser Services könnten – oder besser: müssten – nicht unterscheiden, ob sie mit einem lokalen Service oder über einen Proxy mit einem entfernten Service kommunizieren. Ähnliche Szenarien wer-

den im *OSGi 4.2 Early Draft* im Kapitel „RFC 119 – Distributed OSGi“ [OSGi4.2] beschrieben. Im Folgenden beschreiben wir diese Szenarien und übertragen sie dann auf eine SOA-OSGi-Kommunikation.

## Distributed OSGi in OSGi 4.2

Die Early-Draft-Spezifikation von OSGi 4.2 beschreibt im Kapitel „Distributed OSGi“ mögliche Services und Komponenten für die Kommunikation zwischen verteilten OSGi-Umgebungen, aber auch mit anderen nicht weiter spezifizierten Systemen. Wenn die Verbindung über allgemeine Protokolle wie z. B. SOAP oder andere Webservices-Standards geschieht, brauchen diese auch keine Java-Anwendungen zu sein. Dieser Artikel spricht in diesem Zusammenhang deshalb allgemein vom Service-Export, wenn ein lokaler Service für die Außenwelt zugänglich gemacht wird, und vom Service-Import, wenn ein externer Service lokal verfügbar wird.

Die Kommunikation mit den einzelnen Endpunkten soll für den OSGi-basierten Partner möglichst transparent ablaufen. Damit ist sichergestellt, dass Entwickler, die mit der OSGi-Entwicklung vertraut sind, problemlos eine Kommunikation zwischen verschiedenen Komponenten herstellen können, ohne dass sie genauere Kenntnisse über den Ort der Komponenten oder das verwendete Protokoll besitzen. Die OSGi-Spezifikation definiert keine neuen Protokolle oder Datenformate. Sie spezifiziert lediglich Services, Klassen und Meta-Daten für die Nutzung in einer späteren Implementierung. Die Spezifikation definiert dabei einige Anforderungen, die eingehalten werden sollten. Die wichtigsten sind:

- ▼ *Keine Abhängigkeiten vom Kommunikationsprotokoll:* Für einen OSGi-Service, sei es ein Remote-Service-Consumer oder ein -Provider, darf es keine Abhängigkeiten vom verwendeten Kommunikationsprotokoll geben. Einzig die OSGi-Middleware ist für die Kommunikation zwischen den verteilten Partnern zuständig.
- ▼ *Auswahl der Remote-Services:* Die Auswahl der zu exportierenden Services geschieht entweder direkt über einen Export des Dienstes mithilfe des speziellen Discovery-Service, oder aber über bestimmte Service-Propertyts. Im letzteren Fall werden zu exportierende Services mit diesen Propertyts markiert und können damit durch bestimmte Listener automatisch exportieren werden. Dadurch wird eine Abhängigkeit der funktionalen Services von der technischen Lösung vermieden.
- ▼ *Transparenter Service-Zugriff:* Für einen Service-Consumer muss die Verwendung des Service vollkommen transparent sein. Für ihn ist der Zugriff auf einen lokalen Service formal identisch mit dem Zugriff auf einen entfernten Service.

Durch die Einhaltung dieser Anforderungen ist ein Austausch des Kommunikationsprotokolls ohne Änderungen bei den Service-Providern und -Clients möglich. Zur Einführung in diese Spezifikation wird ein Blick auf die definierten Services und deren Funktionalitäten geworfen, um dann beispielhaft die exemplarischen Stellen einer möglichen Implementierung zu zeigen. Die in diesem Artikel gezeigte Implementierung einer Kommunikation über Webservices kann dank der eingehaltenen Anforderungen leicht auf ein anderes Protokoll übertragen werden.

## Von einem Entwurf zum Prototypen

Abbildung 1 zeigt den Aufbau und den Ablauf der Remote-Service-Funktionalitäten, wie sie im Early Draft beschrieben

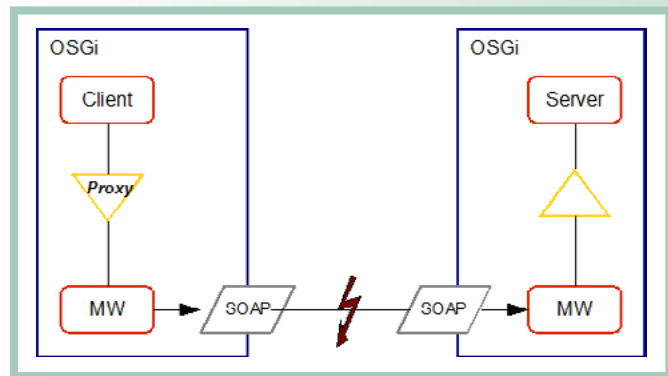


Abb. 1: Ex- und Import von OSGi-Services (nach [OSGi4.2])

sind. Beim Export eines lokalen OSGi-Service sorgt die Implementierung (Middleware) für die Erreichbarkeit des lokalen Service von außen. Die Middleware bietet entsprechende Schnittstellen für das jeweilige Kommunikationsprotokoll und setzt Protokoll-Aufrufe in entsprechende Java-Aufrufe an den Service um.

Bei einem Import eines externen Service in die lokale OSGi-Umgebung nimmt die Middleware die Anfragen als Service-Proxy entgegen und leitet diese an den entsprechenden externen Service weiter. Dies kann, wie in der Abbildung zu sehen, eine andere OSGi-Komponente oder aber ein Service in der SOA sein. Zudem kann die Implementierung eine entsprechende externe Service-Registry durchsuchen und benötigte externe Services somit in die Umgebung importieren.

## Das Tor zwischen OSGi und der SOA-Welt

Als wichtigste Schnittstelle für die Kommunikation mit externen Systemen definiert die Spezifikation das **Discovery**-Interface, welches teilweise in Listing 1 zu sehen ist.

```
public interface Discovery {
    ...
    Collection findService(String interfaceName, String filter);

    void findService(String interfaceName, String filter,
        ServiceListener callback);

    ServiceEndpointDescription publish(Map javaInterfacesAndVersions,
        Map javaInterfacesAndEndpointInterfaces,
        Map properties);
    ...
    void unpublish(
        ServiceEndpointDescription serviceEndpointDescription);
    ...
}
```

Listing 1: Discovery-Interface, Ausschnitt

Diese Schnittstelle beschreibt Methoden zum Exportieren lokaler Services und zum Suchen entfernter Services. So ermöglicht die Methode **publish** den Export des in den Parametern angegebenen Service. Die Implementierung, die exemplarisch weiter unten beschrieben wird, sucht den entsprechenden lokalen Service aus der OSGi-Umgebung und stellt diesen extern zur Verfügung. Bei Bedarf ist hier auch eine automatische Registrierung in einer entsprechenden Registry möglich. Somit können andere externe Partner diesen Service leichter finden und ihn entsprechend verwenden.

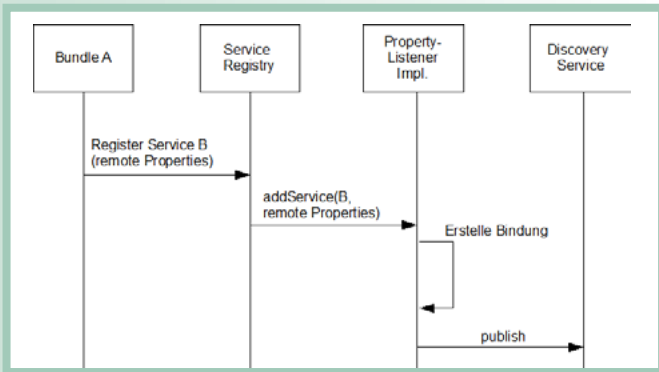


Abb. 2: Export eines OSGi-Service über Service-Propertys (nach [OSGi4.2])

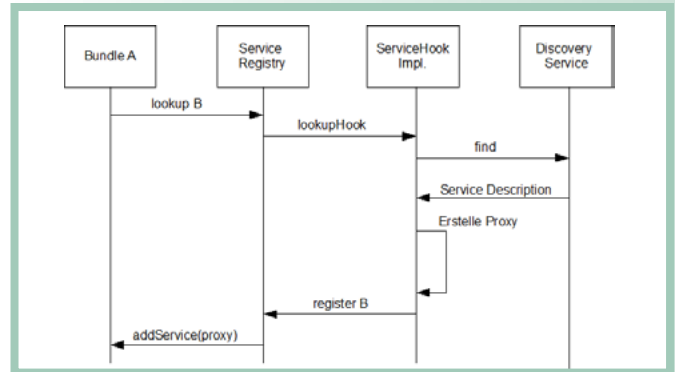


Abb. 3: Import eines entfernten Service in die OSGi-Umgebung (nach [OSGi4.2])

Allerdings ist der Weg des direkten, fest im Quellcode vorgegebenen Exportierens über diese Methode nicht sehr flexibel. Aus diesem Grund definiert das Early Draft das Property `osgi.remote.interfaces`, über das ein Service in der OSGi-Registry kenntlich gemacht werden kann. Ein Property-Listener nimmt dieses Property wahr und exportiert diesen Service daraufhin über den *Discovery-Service* (s. Abb. 2).

Der Import eines externen Service in die lokale OSGi-Umgebung erweist sich als ein wenig komplexer. Zunächst muss die entfernte Registry nach dem gewünschten Service durchsucht werden. Ist dieser gefunden, muss die Implementierung dafür sorgen, dass dieser entfernte Service auch lokal vorhanden ist. Sie erstellt einen Proxy und meldet diesen in der lokalen OSGi-Umgebung an. Der Proxy agiert nun als lokaler Stellvertreter in der OSGi-Umgebung. Bei einem internen Methoden-Aufruf an den Proxy müssen die Parameter verpackt (marshalling) und an den entfernten Service gesendet werden. Die darauf folgende Antwort ist wieder zu entpacken und an den Aufrufer als normaler Return-Wert zurückzugeben. Dieser Ablauf folgt damit dem bekannten Vorgehen von RPC, Corba usw.

Die Service-Suche erfolgt ebenfalls über das *Discovery-Interface*. Die Methode `findService` ermöglicht die Suche nach entfernten Services. Über das zurückgegebene *ServiceDescription*-Objekt ist der Ort des entfernten Service bekannt und die Erstellung eines Proxys nicht mehr schwierig.

Natürlich ist es wünschenswert, wenn andere lokale Services bei ihrer Service-Suche wie gewohnt die normale OSGi-Registry abfragen können und keine Kenntnisse und damit Abhängigkeiten zu den Remote-Service-Funktionalitäten besitzen. Die Suche nach entfernten Services muss also parallel zur lokalen Suche erfolgen. Die unten vorgestellte Implementierung klinkt sich hierfür als *ServiceHook* für die Service-Lookups in die OSGi-Registry ein. Der *ServiceHook* ist ebenfalls eine in OSGi 4.2 [OSGi4.2] neu definierte Möglichkeit, sich in der Service-Registry als *Listener* für bestimmte Ereignisse anzumelden.

Abbildung 3 verdeutlicht dabei den Ablauf für einen *ServiceHook* bei der Remote-Service-Suche. Bei einem Service-Lookup bekommt der Listener dies mitgeteilt. Ist der Service noch nicht lokal vorhanden, kann der Listener die entfernte Registry nach entsprechenden Services durchsuchen. Bei einer erfolgreichen Suche wird ein Proxy erstellt und in der OSGi-Registry angemeldet. Ein erneuter Lookup oder z. B. ein *Service-Tracker* [OSGi06] findet anschließend den Proxy-Service und gibt die Service-Referenz des Proxys an den Consumer, der mit dem entfernten Service nun normal arbeiten kann.

### Implementierung für eine Webservice-Kommunikation

Die bisherige Darstellung gibt die Möglichkeiten des Importes und Exportes von Services auf Service-Ebene wieder, wie sie in OSGi 4.2 beschrieben sind. Im Folgenden wird exemplarisch gezeigt, wie die vorgestellten Möglichkeiten relativ einfach implementiert werden können. Als Basis für die Webservice-basierte Kommunikation dient hierbei das Webservice-Framework Apache CXF [CXF]. Durch einen integrierten Jetty [Jetty] kann es als Stand-Alone Service fungieren und ist nicht auf Servlet-Container oder Applikationsserver angewiesen.

In Listing 2 ist eine beispielhafte Implementierung der `publish`-Methode des *Discovery*-Interfaces zu sehen. Der erste Parameter beinhaltet die zu exportierenden Service-Namen als Key sowie die Version als Value. In dieser beispielhaften Implementierung dient nur der Name als Kriterium, mit dem die jeweilige Service-Implementierung in der OSGi-Registry gesucht wird. Filter können hier natürlich die Auswahl spezifizieren. Als zweites wird das Java-Interface des zu exportierenden Services geladen. In einer normalen Java-Umgebung könnte dies direkt über `Class.forName()` erfolgen. In der OSGi-Umgebung mit dem separaten Bundle-Classloader muss das Implementierungs-Bundle aber Zugriff auf das entsprechende Java-Package haben. Da ein Anpassen des Bundle-Manifestes für jeden neuen Service-Im- und -Export nicht akzeptabel ist, muss eine andere Lösung verwendet werden. Hier liegt es nahe, die bereits vorhandene Implementierung der Service-Schnittstelle zu verwenden. Ihr *ClassLoader* lädt das benötigte Interface. Ist das Service-Interface bestimmt, erfolgt der Export z. B. mit Hilfe der *JaxWsServerFactoryBean*-Klasse [JaxWS].

```
public ApacheCXFDIScoveryImpl implements Discovery {
    ...
    ServiceEndpointDescription publish(Map javaInterfacesAndVersions,
    Map javaInterfacesAndEndpointInterfaces, Map properties) {
        ServiceEndpointDescriptionImpl description =
            new ServiceEndpointDescriptionImpl();
        for (Object key : javaInterfacesAndVersions.keySet()) {
            String interfacename = (String) key;
            description.addInterfaceName(interfacename);
            /* add more description */
            ServiceReference serviceReference =
                Activator.getBundleContext().getServiceReference(
                    interfacename);
            Object service = Activator.getBundleContext().getService(
                serviceReference);
            Class serviceclass = service.getClass()
                .getClassLoader().loadClass(interfacename);
            JaxWsServerFactoryBean svrFactory =
```

```

        new JaxWsServerFactoryBean();
    svrFactory.setServiceClass(serviceclass);
    svrFactory.setAddress("http://" + name + ":" + port + "/"
        + serviceclass.getSimpleName());
    svrFactory.setServiceBean(service);
    server = svrFactory.create();
    }
    return description;
}
...}

```

Listing 2: Export eines Service mittels JaxWsServerFactoryBean

Der Import eines Service erfolgt, wie beschrieben, über einen `ServiceRegistryHook`. Dies triggert einen Aufruf der `find`-Methode des `FindHook`-Interfaces bei jeder Anfrage der Service-Registry an. Listing 3 zeigt die Klasse `FindHookListener`, welche das `FindHook`-Interface implementiert. Anhand der übergebenden Parameter der `find`-Methode wird ein `ServiceDescription`-Objekt erstellt und an den `Discovery`-Service als asynchrone Anfrage übergeben. Findet dieser einen entsprechenden Service, meldet er sich über die Methode `serviceAvailable` des `ServiceListener`-Interfaces, die vom `ServiceAvailableListener`-Objekt bereitgestellt wird. Diese Methode muss zunächst das eigentliche Service-Interface laden. Mit diesem Interface und weiteren Parametern kann die `JaxWsProxyFactoryBean`-Klasse einen Proxy erstellen. Dieser wird anschließend mit den entsprechenden Propertyts in der lokalen OSGi-Registry angemeldet. Der Service ist nun in der lokalen OSGi-Umgebung verfügbar und kann von anderen Services transparent genutzt werden.

```

public class FindHookListener implements FindHook {
    ...
    Discovery discovery;
    ...
    void find(BundleContext context, String name, String filter
        boolean allServices, Collection references) {
        ServiceDescription desc = new ServiceDescription();
        /* create ServiceDescription */
        discovery.findService(desc, null,
            new ServiceAvailableListener(context));
    }
    ...
}

public class ServiceAvailableListener implements ServiceListener {
    ...
    void serviceAvailable(ServiceDescription desc) {
        Class serviceclass=
            context.getBundleClassLoader().load(desc.getInterfaceName());
        JaxWsProxyFactoryBean factory = new JaxWsProxyFactoryBean();
        factory.setServiceClass(serviceclass);
        factory.setAddress(desc.getProperty("remote.url"));
        factory.setWsdURL(desc.getProperty("remote.interface"));
        Object client = factory.getClientFactoryBean().create();
        Properties props = new Properties();
        /* copy properties from ServiceDescription */
        ServiceRegistration reg = Activator.getBundleContext().
            registerService(desc.getInterfaceName(), client, props);
        /* store registration */
    }
}

```

Listing 3: Import eines Service mittels JaxWsProxyFactoryBean

## Fazit

Die im OSGi Early Draft 4.2 beschriebenen Funktionalitäten erlauben die Kombination von SOA und OSGi auf einfache Art und Weise. Der Artikel stellte die relevanten Services des Drafts

zunächst auf Schnittstellen-Ebene vor. Anschließend verdeutlichen Auszüge eine potentielle Implementierung mittels des Apache CXF-Frameworks. Aufgrund der generischen Schnittstellen des Early Drafts ist eine Erweiterung für andere Protokolle möglich, ohne dass dafür die eigentlichen Services verändert werden müssten.

Da OSGi 4.2 sich noch in einer frühen Standardisierungsphase befindet, können sich die Schnittstellen sicherlich noch etwas verändern. Es ist aber jetzt schon erkennbar, dass mit dem Standard 4.2 OSGi einen wichtigen Schritt Richtung SOA machen und damit für neue Anwendungsszenarien geöffnet wird.

## Literatur und Links

- [CXF] Apache CXF: An Open Source Service Framework, <http://cxf.apache.org/>
- [Equinox] OSGi-Framework Equinox, [www.eclipse.org/equinox/](http://www.eclipse.org/equinox/)
- [Felix] OSGi-Framework Apache Felix, <http://felix.apache.org/>
- [Heut06] R. Heutschi, Ch. Legner, H. Österle, Serviceorientierte Architekturen: Vom Konzept zum Einsatz in der Praxis, DW 2006 - Integration, Informationslogistik und Architektur, LNI Proceedings 90, Köllen Verlag, 2006, <http://www.alexandria.unisg.ch/EXPORT/DL/32282.pdf>
- [JaxWS] Apache CXF User Guide – JaxWS, <http://cwiki.apache.org/CXF200C/jax-ws.html>
- [Jetty] Jetty, <http://www.mortbay.org/jetty/>
- [JOnAS] Java Open Application Server, <http://jonas.objectweb.org/>
- [OSGi] OSGi™ – The Dynamic Module System for Java™, <http://www.osgi.org/>
- [OSGi06] OSGi Service Platform, Service Compendium, 2006, <http://www.osgi.org>
- [OSGi4.2] OSGi Service Platform Release 4 – Version 4.2 – Early Draft 2, <http://www.osgi.org/download/osgi-4.2-early-draft.pdf>
- [sdm05] J.-P. Richter, Th. George, T. Gugel, Th. Heimann, H. Lange, Th. Möllers, Technology Guide SOA, sd&m, 2005, [http://www.de.capgemini-sdm.com/web4archiv/objects/download/pdf/2/soa\\_technologyguide.pdf](http://www.de.capgemini-sdm.com/web4archiv/objects/download/pdf/2/soa_technologyguide.pdf)
- [ServiceMix] Open-Source-ESB Apache ServiceMix, <http://servicemix.apache.org/>
- [W3C] Web Services Architecture, W3C Working Group Note 11.2.2004, <http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/wsa.pdf>



**Matthias Füller** ist Diplom-Informatiker und wissenschaftlicher Mitarbeiter an der FHDW Paderborn. Dort konzipiert und implementiert er in F&E-Projekten OSGi- und SOA-basierte Anwendungen. E-Mail: [matthias.fueller@gmx.net](mailto:matthias.fueller@gmx.net).



**Dr. Willi Nüßer** ist seit 2002 Professor für angewandte Informatik an der Fachhochschule der Wirtschaft (FHDW) in Paderborn und Inhaber der Heinz Nixdorf Stiftungsprofessur. Er war von 1996 bis 2002 bei der SAP AG u. a. im LinuxLab in Walldorf tätig. Seit 2002 beschäftigt er sich in Theorie und Praxis mit der Webservice-Technologie. E-Mail: [wilhelm.nuesser@fhwdw.de](mailto:wilhelm.nuesser@fhwdw.de).